Pattern Recognition 43 (2010) 1306–1319

Contents lists available at ScienceDirect

Pattern Recognition

journal homepage: www.elsevier.com/locate/pr

# A new clustering algorithm for coordinate-free data

Alejo Hausner

*University of New Hampshire, Durham, NH 603-862-1237, USA*

ARTICLE INFO

*Article history:*
Received 18 July 2008
Received in revised form
1 August 2009
Accepted 28 October 2009

*MSC:*
05C15
62H30
54E35

*Keywords:*
Cluster analysis
Graph coloring
Metric space
Partition

ABSTRACT

This paper presents the colored farthest-neighbor graph (CFNG), a new method for finding clusters of similar objects. The method is useful because it works for both objects with coordinates and for objects without coordinates. The only requirement is that the distance between any two objects be computable. In other words, the objects must belong to a metric space. The CFNG uses graph coloring to improve on an existing technique by Rovetta and Masulli. Just as with their technique, it uses recursive partitioning to build a hierarchy of clusters. In recursive partitioning, clusters are sometimes split prematurely, and one of the contributions of this paper is a way to reduce the occurrence of such premature splits, which also result when other partition methods are used to find clusters.

## 1. Introduction

This paper presents a new method for cluster analysis: the colored farthest-neighbor graph (CFNG). Cluster analysis studies the problem of finding groups of similar objects. Given a set of objects, this problem amounts to splitting the set into clusters or groups, such that the objects in each group are more similar to each other than to the objects in other groups.

A basic assumption of cluster analysis is that little is known about the data initially. The rules or characteristics that will define a group are not known *a priori*. Since clusters are identified based on similarity between objects, often the only assumption made in cluster analysis is that, given any two objects in the data, we can compute a positive number which expresses their degree of similarity. This degree of similarity is usually expressed as a distance: if two objects are very similar the distance between them is small, while two less-similar objects will have a larger distance between them.

Knowing the degree of similarity between objects does not reveal what caused clusters of objects to occur in the first place. We do not expect to learn everything there is to know about the data. Most often, identifying clusters in the data is simply a *first step* to discovering useful information about the data itself.

For example, the data may be genetic samples: here, each object is a version of the same gene, found in several different individuals of the same species, or from individuals in several related species. The information in each object is a sequence of DNA base pairs. The distance between two such genes is defined as the number of places where the base pairs are different. Such differences are due to mutations, and more mutations imply that more generations have passed since the two individuals descended from a hypothetical common ancestor. Conversely, fewer mutations means the two genes are more similar, and that they descended from a more recent common ancestor. With this definition of similarity, cluster analysis aims at finding groups of related individuals. Notice that the analysis may not yield any more information beyond discovering such groups. Actually discovering geographic, historical, environmental or other causes for the observed groups remains a task for a specialist in the domain. As stated above, cluster analysis simply provides a first stage in knowledge discovery.

### 1.1. Metric spaces and embedding

If we can compute degrees of similarity between objects, we are stating that we know the objects belong to a *metric space*. However, sometimes we also know that the objects are *embedded*. Before proceeding, let us define these two terms more carefully.

A metric space consists of a set of objects $S$, and a function $d(x,y)$ called the *metric* which takes two objects in the set and

*E-mail addresses:* ah@cs.unh.edu, hausner@dwc.edu.
*URLS:* http://www.cs.unh.edu/~ah, http://faculty.dwc.edu/hausner.

0031-3203/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.patcog.2009.10.018

yields a positive real number. This number is the distance between the two objects. If two objects $x$ and $y$ are similar, their distance $d(x,y)$ will be small, and if the objects are very different, it will be large.

The metric function $d(x,y)$ usually has the following properties:

1. $d(x,y) > 0$ iff $x \neq y$ (non-negativity).
2. $d(x,y) = d(y,x)$ (symmetry).
3. $d(x,y) + d(y,z) > d(x,z)$ (triangle inequality).

The CFNG method works even for metrics $g(x,y)$ that do not obey the triangle inequality; such metrics are called *semimetrics*.

A set of objects is said to be embedded if each object has coordinates. For example, 2D points on the plane and 3D points in space are embedded; the number of coordinates attached to each object is the *dimension* of the space. Coordinates give the objects position, and from them, distances can be obtained. Not all objects are embedded. For example, words are sequences of characters, and web pages can be identified as collections of words, and neither of these has natural coordinates. This means that words and web pages do *not* have position, *i.e.*, are not embedded.

In two dimensions, the distance between two points can be computed using Pythagoras' theorem, and this theorem generalizes to the Euclidean distance used in higher dimensions. This implies that embedded objects also belong to a metric space. However, the converse does not hold: many interesting data sets exist which are in a metric space, but are not embedded. This distinction is important for the CFNG method. The method's only requirement is that the objects being clustered belong to a metric space. This differs from many popular methods, which rely on embeddedness: they often need to create a synthetic object that represents the "center" of a cluster, even though this center is often not itself an object in the original input set. For non-embedded data, such approaches cannot be used. They must be adapted in ways that render the methods less efficient.

## 2. Background and related work

Before presenting the CGNG method in detail, it is useful to survey popular approaches to cluster analysis. This survey will necessarily be incomplete: cluster analysis is a very well-researched area. In fact, early work predates the invention of digital computers. Many excellent reviews of clustering techniques have been written, including a recent book by Mirkin [21], and papers by Jain et al. [12] and Berkhin [4]. We cannot cover all previous work in the brief overview below. Hence we will focus on major techniques, and on specific methods that relate to the CFNG method. The interested reader is encouraged to read the reviews given above to obtain a broader outlook.

Popular methods for cluster analysis can be grouped into three main categories: $k$-means, agglomeration hierarchies, and partition hierarchies. The CFNG method, presented later in the paper, builds a partition hierarchy. Before proceeding further, we will briefly describe the three classes of clustering methods, to better understand where the CFNG method fits in. The first class is $k$-means.

### 2.1. *k-Means*

Given a set of objects, the $k$-means method finds clusters by initially guessing $k$ initial cluster centers (see Fig. 1a, b, where $k=2$). The initial cluster centers are chosen arbitrarily, usually randomly. The position of each center is then repeatedly adjusted by classifying and averaging. For each iteration, the objects are first classified: each object is matched to the one center it is closest to (Fig. 1c). Then, the objects are averaged: for each center, the average position of all its matching objects is obtained (a total of $k$ means are obtained, hence the method's name). Then, each center is moved to its corresponding mean position (Fig. 1d). Classification and averaging are repeated, until the centers stop moving (Fig. 1f), at which time the final classification of objects groups them into $k$ different clusters.

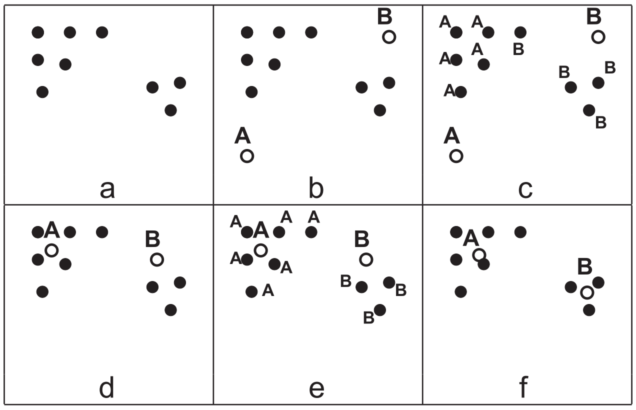


**Fig. 1.** The $k$-means algorithm: (a) shows a set of points on the plane; (b) two randomly placed cluster centers are chosen, and then (c) each point is given the label of its nearest cluster center. (d) The average (mean, or centroid) of each class of points is computed, and the each center is moved to its mean, at which time (e) the points are labeled once more, (f) centroids are re-computed, and the centers moved again. The algorithm has converged.

The $k$-means method was first described by McQueen [18]. Running $m$ iterations on a set of $n$ objects costs $\Theta(mkn)$ time, so the method is fast. However, its simplicity leads to some limitations. First of all, it requires that $k$ be equal to the number of clusters actually present in the data, a quantity that is by definition unknown at first. In other words, the $k$-means method simply assumes that there are $k$ clusters, and then tries to estimate their *positions*. The iterative anomalous cluster [20] is a modification of $k$-means that avoids this assumption. This method begins with a single cluster, then adds new ones, each centered at the object farthest from existing cluster centers. However, such approaches require multiple iterations, and do not scale well to large data sets.

Another limitation is that the positions thus found depend on the initial choice of center positions: a different initial random choice for the $k$ positions will often yield a different final set of cluster centers. This problem can be mitigated by re-running the algorithm several times, and stopping when the same set of centers is found several times. Unfortunately such a solution also loses some of the speed advantage of the $k$-means method.

For our purposes, a more serious limitation is that $k$-means requires embedded data: using an average to compute the center of a group of objects can only be done if the objects have coordinates. For non-embedded objects, this problem can be overcome by finding a *medioid object* instead of *center position*. The $k$-medioids or partitioning around medioids (PAM) approach was introduced by Kaufman and Rousseeuw [16]. For a set of objects in a metric space, the medioid is the object in the set that is nearest the "middle" of the set: it is the object $x$ with the smallest total distance $\sum d(x,y)$ to all other objects $y$ in the set. For a set of $n$ objects, computing this medioid costs $\Theta(n^2)$ time. As a result, PAM is too slow for large data sets. Even for non-embedded data such as strings and graphs, it is possible to obtain a "mean" object for a set of objects, but the problem of obtaining such a representative is known to be NP-hard [13].

### 2.2. Agglomeration hierarchy

$k$-Means uses partitioning: it divides a large set of objects into $k$ smaller subsets. Agglomeration works in the opposite direction: it groups small subsets into larger sets. Fig. 2a shows agglomeration working on seven points on the plane (**a**–**g**). At each stage in the algorithm, a set of entities is considered. Each entity is either an original object, or a cluster of objects. Each cluster may be represented by a center point, located at the average position of the two entities that constitute it, although such synthetic points are not always used. Each step of agglomeration finds the closest two entities (be they clusters or points), removes them from the set, merges them into a cluster, and adds this new entity to the set. Thus, after each step, the set has one less entity. At the end, a single entity remains, representing a cluster that contains all the original points. In Fig. 2b, **a** and **c** are the two closest points, and are replaced by a new cluster, centered at **1**. In Fig. 2c, **e** and **g** are joined, in Fig. 2c, **f** and **1** are joined, and so on.

The process builds a binary tree. When two entities are joined, a new tree node is created, representing the new cluster. Its children are the two entities that were joined. The shape of the tree reflects the structure of the clusters in the original data: clusters are represented by subtrees, and objects that were close together in the original set tend to end up near each other in the tree. For example, the tree shown in Fig. 2g has two main subtrees: the leaf nodes of the left subtree {**a**,**c**,**d**,**f**} make up one main cluster, and those in the right {**b**,**e**,**g**} make up another. Within the left subtree, {**a**,**c**,**f**} form a subcluster, which can be identified as a subsubtree.

Unlike the simple $k$-means method, agglomeration does not require that the number of clusters be known *a priori*: the cluster structure of the data is usually reflected in the structure of the tree itself. However, agglomeration is slow. If there are $n$ objects, agglomeration will need $\Theta(n^3)$ time to run because each merge requires $\Theta(n^2)$ distance calculations, and $n$ merges are needed. Thus agglomeration is usually applied only to relatively small data sets.

The key step in building an agglomeration hierarchy is computing the distance between two clusters. Four definitions for this inter-cluster distance are popular: single-link, complete-link, average-link, and minimum-variance. In the single-link approach, first presented by Sneath and Sokal [23], the inter-cluster distance is obtained by computing the distances between every pair of objects $(a, b)$, with $a$ in one cluster and $b$ in the other. The shortest distance thus found is used. The complete-link approach by King [17] also compares all the pairwise distances, but chooses the largest one instead. Complete-link clustering yields more compact clusters, but may miss clusters made up of objects arranged in chains or shells. The average-link distance simply averages the pairwise distances. Ward's minimum-variance approach [14] computes the distance between clusters by considering the mean-square-error between objects and a cluster's centroid. The difference in this error, before and after two clusters are merged, defines the distance between the clusters.

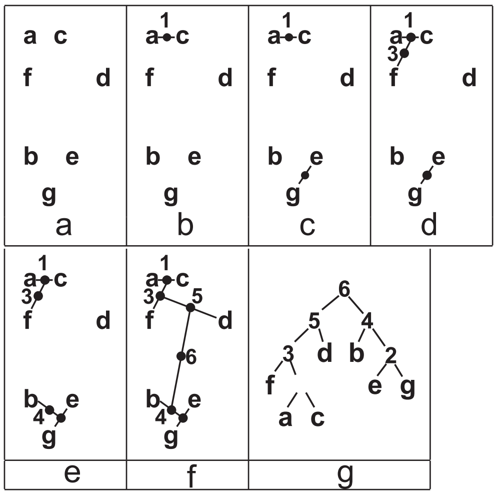


**Fig. 2.** Agglomeration hierarchy: (a) Starting with a set of seven points (a through g) on the plane, (b) the closest two points are joined (into cluster center point 1) and removed, leaving six entities. Then (c) through (f) each time, the next closest pair of points or clusters is joined into a new cluster (center points 2–6), until a single entity (a single cluster) remains. (g) The final hierarchy that is obtained.

### 2.3. Partition hierarchy

Partition-based hierarchies can be built faster than agglomeration hierarchies, although most current partition methods only work on embedded data. The CFNG provides an alternative solution: it is partition-based, but works on non-embedded data. Before considering our method, we must look at partition hierarchies.

Agglomerative methods build a tree from the bottom up, merging objects (tree leaves) and then clusters (internal nodes), until the final merge creates the root of the tree. In contrast, partition-based methods build a hierarchy from the top down. Here we will describe Bentley's kd-tree [2], a popular and simple approach that works by sorting and splitting.

Given a set of points (Fig. 3a), the kd-tree algorithm initially sorts them using one coordinate, and then splits the sorted list. In Fig. 3b, the initial sort is by $y$ coordinate, yielding the list (**g**, **b**, **e**, **f**, **d**, **a**, **c**) of points from bottom to top. This list is then split at the median. Two subsets result, and the sorting and splitting is repeated on both of them recursively. However, the second sort and split is along a different coordinate ($x$ in Fig. 3c). The subdivision continues recursively, each time along a different coordinate. It produces ever smaller subsets, until a minimum subset size (possibly a single object) is reached (Fig. 3d). The smallest subsets become leaves of the tree, and other tree nodes above them correspond to larger subsets which were split. The root node of the tree corresponds to the entire input data set (Fig. 3e).

Similar to agglomeration, partition often produces a tree whose structure reflects the clusters in the data. In our example, each subtree of the tree has leaf objects that make up clusters in the original data. However, partitioning is faster than agglomeration, because sorting is used, and the total time cost of building the kd-tree is $\Theta(n\log^2 n)$. Despite this advantage, the partition method relies on coordinates. Without them, sorting and splitting would not be possible.

The kd-tree was originally proposed as a means to accelerate spatial search, not clustering. Other methods that create spatial

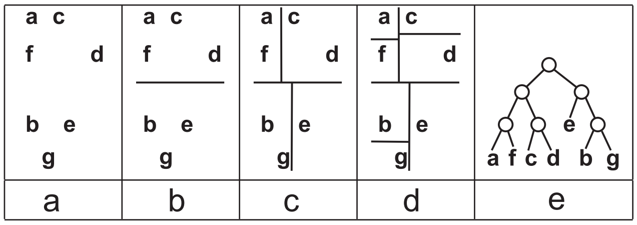


**Fig. 3.** kd-Tree partition hierarchy: (a) initial objects; (b) objects are sorted by $y$ coordinate, and split at the median; (c) both halves sorted on $x$ coordinate, and split along median; (d) median splits on $y$-axis again; (e) the tree created from the partitions. Circle tree nodes represent partition lines.

search structures for points can also be applied to cluster analysis. Finkel and Bentley's quadtree [8] is similar to the kd-tree, but partition is always at the halfway point between extrema in each dimension, and not at the median of the data points' coordinates. In a variant presented by Mirkin [20] which is specifically aimed at clustering, each partition is effected using $k$-means with $k=2$. This tends to better adapt to the data, but needs multiple runs for each partition to ensure a good pair of means is found, a limitation of $k$-means that was mentioned above. Guttman's R-tree [11] is another adaptive spatial data structure, which groups objects hierarchically "bottom up" instead of splitting space, and its structure also adapts to clusters in the data. R-trees and their variants are attractive because they can handle dynamic data: insertion and deletion are $O(\log n)$- time operations. In contrast, partitional hierarchies such as the kd-tree are not dynamic: if any data object changes, the whole hierarchy may have to be re-built from scratch.

For most partitional hierarchies like the kd-tree, each split is *binary*: the set of points is split into two parts. This may be appropriate if the set does indeed contain two clusters. If the data contain more than two clusters, each part in the split may contain several clusters. Since each part will be split again recursively, these clusters will be isolated by later splits in the process, and will end up as subtrees further down in the hierarchy (this assumes that the splits do obey the data's cluster structure, which does not always happen, as will be discussed below in Section 4). As a result, if we consider the points in Fig. 8a, and draw a partitional hierarchy as shown in Fig. 8f, we deprive the reader of the visual clues needed to recognize which subtrees correspond to actual clusters in the data. These clues can be introduced by drawing the same tree as a *dendrogram* [19], in which the lengths of the lines joining a node to its two child nodes are proportional to the distance between the items in one child subtree and those in the other. In Fig. 8f these lines are horizontal. This can be seen in Fig. 8g: short lines leading to leaf nodes 14 and 15 reflect the close proximity between points 14 and 15, while longer lines joining leaf node 10 the 14–15 subtree reflect the greater distance from point 10 to the two-point group of 14 and 15.

Partitional hierarchies have not often been applied to coordinate-free data. One of the few exceptions is Rovetta and Masulli's *shared farthest neighbor* (SFN) approach [22]. Their method identifies clusters by computing all inter-object distances in order to find, for each object, the other object most distant from it (its "farthest neighbor"). Any objects which share the same farthest neighbor are known to be distant from a common object, and are assumed to be close together. They are grouped into a subset, and subdivided recursively by finding SFNs within the subset. Notice that each partition is not necessarily binary: Fig. 5c shows an initial partition into *three* subsets.

The method which will be described below provides a generalization of the SFN algorithm. It is similar to SFN because it works on non-embedded data, but differs in that each partition is binary, just like a kd-tree. It builds a hierarchy with a time cost whose complexity lies between that of agglomeration and coordinate-based partition.

A key step in our algorithm is the construction of the farthest-neighbor graph. This is a graph where each object is joined to the other object farthest from it. Much research has been devoted to methods for building *nearest*-neighbor graphs, including efficient algorithms for Delaunay triangulations [9]. However, little work has been done on the farthest-neighbor problem. One related work is the rotating-caliper algorithm for computing the diameter of a point set [6]. Unfortunately, that algorithm is restricted to Cartesian spaces, where objects are embedded, so it cannot be used to speed up the CFNG method when applied to non-embedded data.

### 2.4. Contributions

The two main contributions of this paper are:

1. A new method for cluster analysis of objects in a metric space. The method does not require coordinates. It uses graph coloring to improve on and generalize an existing technique by Rovetta and Masulli [22], which relies on grouping objects that share a farthest neighbor.
2. A way to improve the quality of partitions. Binary partition can sometimes cut up a cluster of objects that should remain together. Such undesirable splits occur not only in the CFNG method, but in many other methods that use a "top-down" approach to find clusters, methods such as the 2-means cluster analysis and median-split approaches. At the cost of extra computation, the improvement yields a hierarchy where clusters of objects are more likely to remain intact.

## 3. Approach

The CFNG method is a technique for finding clusters, which builds a partition hierarchy. The key step in building the hierarchy is splitting one set of objects into two subsets. This split is achieved in two steps. First, the objects are used to obtain their *farthest-neighbor graph* (FNG). Then, this graph's vertices are colored, using just two colors. Finally, the vertices are separated into two subsets, one for each color. Objects in each subset tend to be near each other, and far from objects in the other subset. In other words, this process tends to produce good partitions. We apply this split to each subset again, and keep splitting into smaller and smaller subsets until single objects are obtained.

The following sections describe each of these steps. Section 3.1 details how the FNG is built, Section 3.2 details how it is colored, and Section 3.3 details how the partition hierarchy is built.

### 3.1. Graph construction

A *graph* is a set of objects, along with a set of links between some of the objects. Each object in the graph is called a *vertex*, and each link is called an *edge*. The set of vertices is usually denoted $V$, while $E$ is the set of edges.

The *farthest-neighbor* graph (FNG) can be built for any set of objects in a metric space. This graph consists of the objects (which will be vertices of the graph), and edges between the objects. In this graph, each edge connects an object to the object most distant from it: its *farthest neighbor*. Fig. 4a shows a set of points the plane, and Fig. 4b shows their FNG, with lines joining most distant points (the graph edges). In a sense, the word "neighbor" is somewhat misleading in this context, since it suggests nearness and not distance, but it is in common use, so we will use it.

#### 3.1.1. Cost of building the FNG

How much time is needed to build the FNG? The time will depend on the number of objects being analyzed (the *cardinality* of $V$, denoted $|V|$). It will also depend on the cost of computing $g(x,y)$ which depends on the *kind* of data being analyzed. Computing the distance between two 3D points is quick and simple, but computing the edit distance between two strings is quadratic in the lengths of the strings. If the objects being clustered are graphs, then this cost may be even greater: computing the edit distance between two graphs is known to be NP-hard [10]. Having made the reader aware of this issue, the discussion that follows will focus on the dependence on $|V|$ alone.

The FNG's vertices can be grouped into two classes: *extreme* and *interior*. For each vertex $v \in V$, let $f(v)$ be its farthest neighbor.

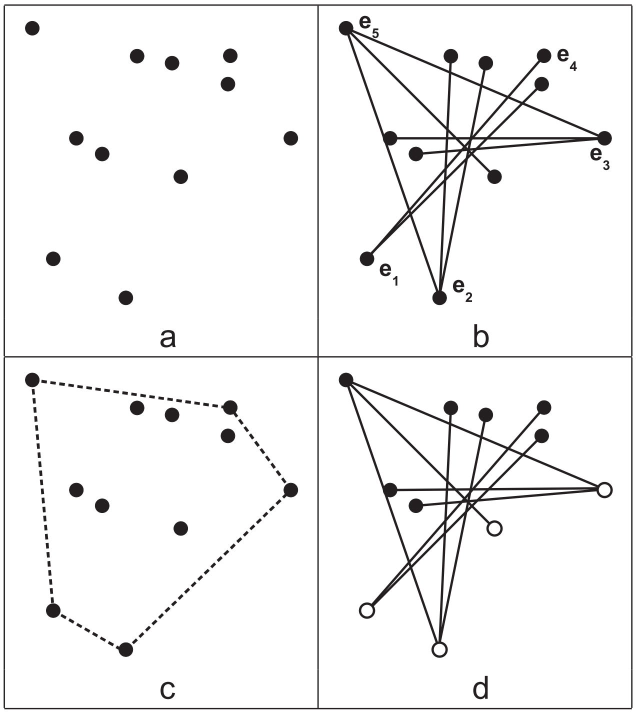


**Fig. 4.** (a) A set of points on the plane, and (b) their farthest-neighbor graph, with edges in the graph shown as solid lines; (c) the convex hull of the same points (dashed lines). Notice that each edge has one or both endpoints on the convex hull. (d) One possible 2-coloring of this farthest-neighbor graph. Notice the separation between black vertices above and the white vertices below.

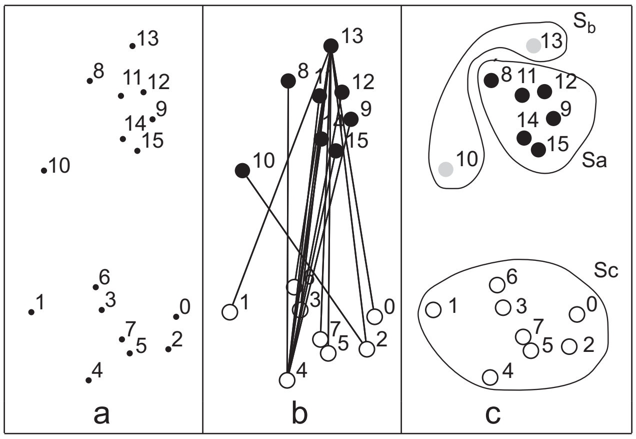


**Fig. 5.** (a) 16 random points on the plane, and (b) their colored FNG. (c) The SFN algorithm splits these points into *three* groups, not two.

Any such $f(v)$ is extreme: in Fig. 4b, points $e_1$ through $e_5$ are extreme. The set $F \subset V$ of extreme vertices consists of all such $f(v)$ (i.e., it is the range of the function $f$). The remaining vertices (which are not farthest neighbors to any other vertex) comprise the set $I = V - F$ of interior vertices.

It is important to note that, if $e = f(v)$, it does follow that $v = f(e)$. This can be seen in Fig. 5b: point 2 is point 10's farthest neighbor, but not vice versa (point 2's farthest neighbor is point 13). In other words, the "farthest-neighbor" relation is not symmetric. As a result, the FNG is a *directed* graph: its edges are oriented, and could be drawn as arrows, not simply as straight line segments.

In some cases, the extreme vertices $F$ can be found easily. Suppose the objects are points in ordinary Cartesian space $R^d$ (where $d$ is the dimension of the space). Then it can be shown that for every edge of the FNG, at least one endpoint lies on the convex hull of the points (see the Appendix for a proof). In other words, for this case, $F$ is a subset of the hull vertices. On the other hand, if the objects have no coordinates, and we only know that they belong to a metric space, little will be known *a priori* about the extreme set $F$.

A simple algorithm can be used to build the FNG. It proceeds by testing each vertex $v_i$ against every other candidate vertex $c_j$ that might be $v_i$'s farthest neighbor. When the farthest candidate $c_{max}$ is found, an edge is added to the FNG. Thus, in general the algorithm's running time will be in $\Theta(|V| \times |C|)$, where $C = \{c_j\}$ is the set of possible candidates. The speed of this algorithm will depend on what is known about the set of candidates $C$. If the objects have no coordinates, we must use a *brute-force algorithm* that tests *all* other vertices: $C = V$; the running time of this brute-force algorithm will be in $\Theta(|V|^2)$. However, if the objects are points in a Cartesian space, we need only to let $C = \text{hull}(V)$. This can result in a much faster algorithm. For instance, it is known that, for $|V|$ randomly placed points, distributed uniformly in a hypercube of $R^d$, the number of points lying on the convex hull is $O(\log^{d-1}|V|)$ [3], which is much less than $|V|$ for large $|V|$. These hull points can be found in time $O(|V| \times |\text{hull}(V)|)$ using the gift-wrapping algorithm [5], so the overall cost of building the FNG for embedded data of dimension $d$ is $O(|V| \times \log^{d-1}|V|)$.

Of course, even if the objects are Cartesian points, they may not be randomly distributed, and their hull may be very large. A simple example consists of 2D points arranged at equal intervals on a circle: every point will be on the hull, and furthermore every point will also be extreme. In this case the algorithm takes $\Theta(|V|^2)$ time. This behavior will also arise with high-dimensional data such as genetic sequences, for which the brute force algorithm must be used. Since its time cost is quadratic, it is not suitable for very large data sets. The CFNG method is best suited to data sets with low cardinality, similar to like Rovetta and Masuli's [22] method.

Nevertheless, in many cases the set $F$ of extreme objects makes up a small subset of the vertices $V$: usually, most vertices are interior, not extreme. Thus it would be good to be able to discover the set $F$ more quickly than in $\Theta(|V|^2)$ time. Solving this problem remains a useful avenue for further research.

## 3.2. Vertex coloring

In any graph, the vertices can be colored so that the two endpoints of each edge receive different colors. To color a graph, vertices do not receive actual visible colors. The word "color" here simply means a unique label. Thus a proper vertex coloring of a graph assigns each vertex a label, such that if any two vertices are adjacent (i.e., are linked by an edge), they will get different labels. The most well-known problem in graph coloring is the four-color theorem. It states that any planar graph (having the same structure as, say, a political map of countries) can be colored with at most four colors [1]. This means that, using just four colors, each country (each vertex) can be colored so that any two countries sharing a border (two vertices joined by an edge) have different colors. If $k$ different labels are used in a coloring, the graph is said to be *k-colored*. The four-color theorem, then, says that planar graphs can always be 4-colored.

Once the FNG is built, the method uses vertex coloring to partition the objects into two well-separated subsets. For an arbitrary graph, a coloring can be found quickly with a simple *greedy algorithm*, which starts with an uncolored graph, and visits the vertices in some order. When it visits a vertex, it examines the colors of its neighbors. If none of the neighbors have been colored, it chooses a color for the vertex arbitrarily, but if some have been colored, it chooses a color that does not occur among the neighbors.

Such a greedy algorithm will always succeed, but potentially may use too many colors. If we demand that the number of colors used be small, the problem is much harder. Simply determining whether $k$ colors are sufficient for a proper coloring is known to be an NP-complete problem [15]. Fortunately, this will not occur in our case. The FNG is a graph with special properties: a *forest*.

### 3.2.1. The FNG is a forest

A forest is a collection of *trees*. A tree, in turn, is a minimally connected graph: in a tree, any two vertices are connected by a single path (a sequence of consecutive edges). A tree can contain no cycle (no path that returns to its starting vertex), for if it did, any two vertices on the cycle would have two paths connecting them, not just a single path.

To see why the FNG must be a forest, we first show that, if the FNG contains a cycle, the objects in the cycle must be equidistant. To see why this must be so, suppose the FNG contains the three-cycle $A \rightarrow B \rightarrow C \rightarrow A$, as shown in Fig. 6a. By this we mean that $B$ is $A$'s farthest neighbor, $C$ is $B$'s farthest neighbor, and $A$ is $C$'s. Hence $d(A,B) \geq d(B,C)$ and $d(B,C) \geq d(C,A)$ and $d(C,A) \geq d(A,B)$. Since the distance metric is symmetric ($\forall x,y, d(x,y) = d(y,x)$), these three inequalities can only hold if $d(A,B) = d(B,C) = d(A,C)$, in other words, if the three objects are mutually equidistant.

Assume that the brute-force algorithm from Section 3.1.1 is used to build the FNG, and assume further that the objects are always examined in the same order. Also, suppose that when an object has several equidistant farthest neighbors, the first such neighbor found is used, and the remaining ones are ignored. In our example, suppose the points just mentioned occur in the order $\cdots A \cdots B \cdots C \cdots$. Then, when searching for $A$'s furthest neighbor, $B$ will be found first, and $C$ will be ignored, (i.e., $B = f(A)$). Similarly, $A = f(B)$ and $A = f(C)$. As a result, the FNG will be built to contain the edges shown in Fig. 6b, and will not contain a cycle. Since the FNG contains no cycles, it must be a forest. As we will show next, forests can always be colored using just two colors.

### 3.2.2. Trees are 2-colorable

We can prove that, for any tree, some 2-coloring exists. To see why this is possible, first convert the tree into a *rooted* tree, where one vertex is privileged as the root vertex, and other vertices are arranged in levels subordinate to the root vertex. Each vertex is at some tree depth: the root is at depth zero, and the depth of every other vertex is its distance from the root. Then, color nodes by alternating colors with increasing depth: the root (level 0) gets black, its children (level 1) get white, its grandchildren (level 2) get black, and so on. This is possible because nodes at the same level are never joined by an edge. This property is illustrated in Fig. 7.

This 2-coloring will also be found by the greedy algorithm described above in Section 3.2. We choose an arbitrary starting vertex, color it white, and then process the remaining vertices in depth-first-search (DFS) order. To process a vertex, the algorithm simply colors it black or white, whatever color is different from the color of its neighboring vertices.

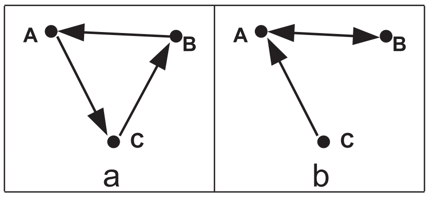


**Fig. 6.** (a) If the FNG contains a cycle, the objects in the cycle must be equidistant, but (b) the FNG built by the brute-force algorithm does not contain cycles.

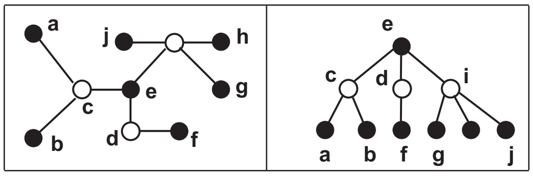


**Fig. 7.** (left) A 2-coloring of a tree, and (right) the same tree redrawn with node **e** as root: nodes at the same level have the same color, and colors alternate black/white by level.

As long as the graph is a single tree, it will be connected, and DFS traversal will visit all the vertices, regardless of starting vertex. However, the FNG often consists of several disconnected trees (as in Fig. 4b), so DFS must be performed repeatedly, once for each tree. To visit all the trees, the greedy coloring algorithm modifies the usual DFS traversal, which uses a stack. Whenever DFS traversal finishes a tree, the stack will be empty; at that point, the algorithm seeds the stack with a vertex from an unvisited tree, and restarts. Note that the vertices *must* be visited in DFS order. Other processing orders may lead to graph colorings that use three or more colors, even though just two colors are necessary.

### 3.3. Building the hierarchy

Now that we have described algorithms for building the FNG and for coloring it, we will show how to combine these algorithms to build a hierarchy suitable for cluster analysis. The hierarchy is built using Algorithm 1. Given a set of objects in a metric space, the method first builds their FNG using the brute-force $\Theta(|V|^2)$ algorithm described above in Section 3.1.1, then 2-colors the vertices of the FNG using the modified DFS algorithm described above in Section 3.2. The set of objects is then partitioned by color; one subset contains all the white vertices of the FNG, and the other subset all the black vertices. The algorithm then recursively builds two partition hierarchies, one for each of the two subsets. One of these hierarchies becomes the left subtree, and the other the right subtree.

**Algorithm 1.** Recursive construction of partition hierarchy.

```
Data: $S$ is a set of objects in a metric space
Result: returns a binary tree of objects
BuildTreeRecursively($S$)
  if $|S| = 1$ then
    return leaf node containing $S$
  else
    $(S_1, S_2) \Leftarrow$ split($S$)
    $n \Leftarrow$ a new tree node
    $n.left \Leftarrow$ BuildTreeRecursively($S_1$)
    $n.right \Leftarrow$ BuildTreeRecursively($S_2$)
    return $n$
  end
split($S$)
  construct $F$, the FNG for the set $S$
  color vertices in $F$
  $B \Leftarrow$ black vertices of $F$
  $W \Leftarrow$ white vertices of $F$
  return($B$,$W$)
```

### 3.3.1. Properties of the hierarchy

An example of the partition algorithm's progress appears in Fig. 8. Fig. 8a shows an initial set of 16 points on the plane,

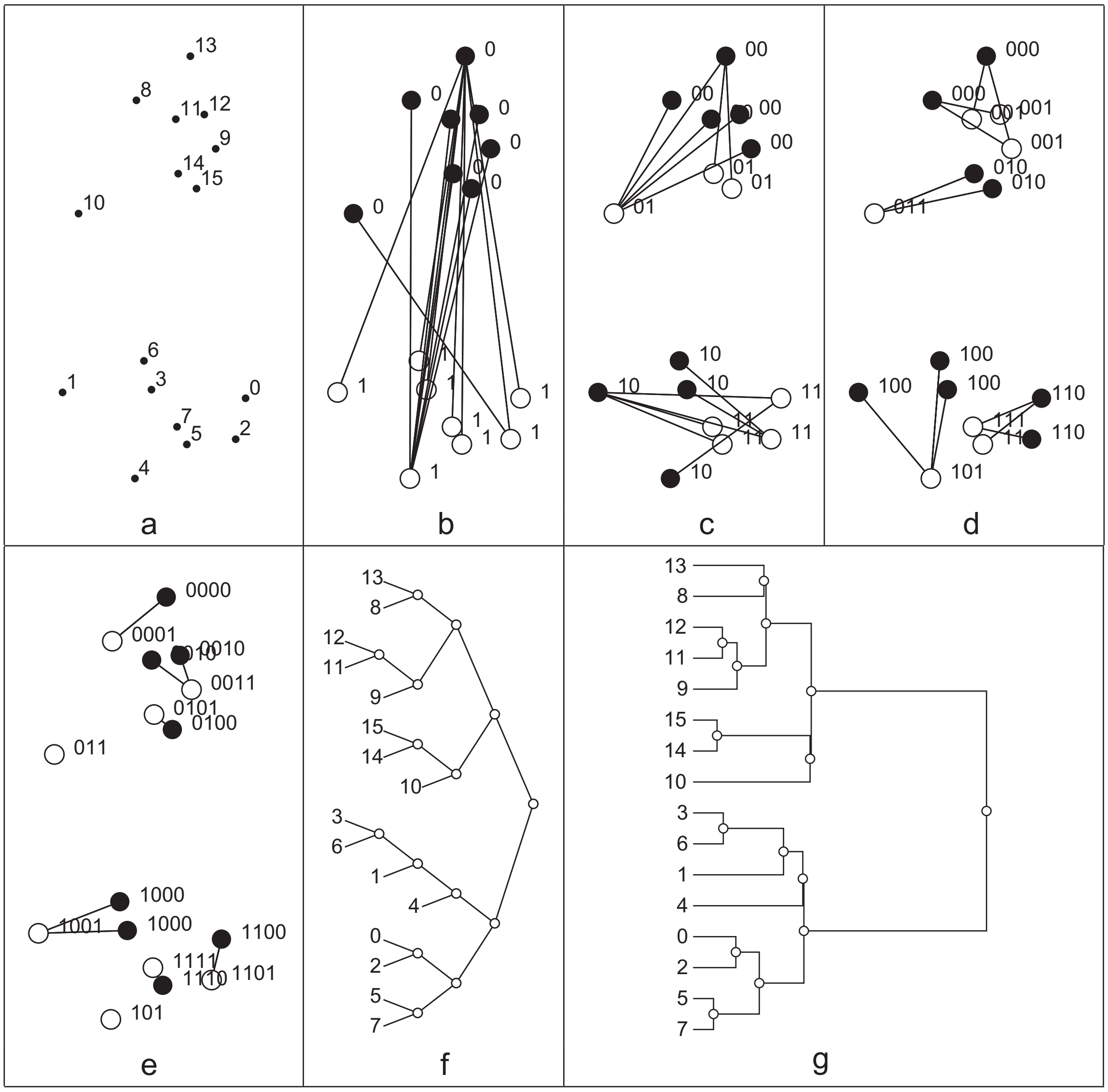


**Fig. 8.** (a) 16 random points on the plane. (b) The first partition step, showing the colored FNG, with black and white graph vertices, labeled 0 and 1, respectively. (c) The black points are partitioned again by building the FNG and coloring them, and so are the white points; four subsets are obtained. Each vertex gets a two-digit label, showing its coloring history: 00 = black, then black, 01 = black, then white, etc. (d) The subsets are partitioned again, and (e) again, and a 0 or 1 is appended to each point's label to show its coloring history. (f) Two more partitions (not shown) produce a hierarchy, with the original points at the leaves. (g) The same tree as a *dendrogram*, in which branch lengths encode the separation between subtree clusters.

randomly generated around two cluster centers. The first partition (Fig. 8b) forms the two main subtrees of the hierarchy, and subsequent partitions (Figs. 8c–e) create descendant subtrees. The resulting hierarchy appears in Fig. 8f, rotated so that higher nodes and the tree root are on the right side, while lower nodes and leaves are on the left.

Notice that the tree structure (Fig. 8f) embodies the clustering properties of the original objects, on many scales. On the large scale, the obvious upper cluster (points 8–15) and lower cluster (points 0–7) are each located on the two main subtrees. Simply removing the nodes nearest the root will break the tree into subtrees that embody the main clusters in the data. On a fine scale, nearby objects end up nearby on the tree. For example, points 14 and 15 are nearest neighbors, and their corresponding tree nodes are also nearby on the tree; in fact, they are siblings. Moreover, relatively isolated points can be considered to form clusters in their own right (e.g., points 1, 4, and 10), and their corresponding tree nodes are on isolated single-leaf branches of the tree.

Another interesting property of the tree is worth mentioning. As the objects are repeatedly colored, we can construct a multi-digit label for each object, by appending a 0 or 1 digit each time. When an object node is colored black, we append a 0, and when white, a 1. For example, point 10 is first colored black, then white, then white again: its final label is **011**. Similarly, point 13's final label is **0000**, and point 7's is **1111**. After all the objects have been thus labeled, they can be sorted lexicographically ("alphabetically") using their labels. The resulting order tends to sort points physically in space. On the tree in Fig. 8f, the leaf nodes have been arranged from top to bottom by their sorted labels. Notice that they end up nearly (but not exactly) sorted by *y* coordinate.

### 3.4. Shared farthest neighbors

The CFNG method shares many characteristics with the "shared farthest neighbors" (SFN) algorithm by Rovetta and Masulli [22]. However, it improves the SFN algorithm in several ways. The CFNG algorithm always yields a binary partition of objects into two subsets, whereas the number of subsets obtained by a partition in the SFN algorithm can vary. The SFN algorithm can easily split a cluster where no natural partition is necessary, while the CFNG often avoids such splits.

This can be seen in Fig. 5, which compares the results of the two algorithms on the same set of random 2D points. Fig. 5b shows the first split in our CFNG algorithm, while Fig. 5c shows the partition made by the SFN algorithm. Recall that the SFN algorithm groups any points that share the same farthest neighbor. Since all points in the bottom cluster (0–7) share point 13 as their farthest neighbor, the SFN algorithm groups them into one cluster (shown in white). However, in the top cluster, points 10 and 13 in the top cluster share point 2 as their farthest neighbor (and are shown in gray), while the others share point 4 as their farthest neighbor (and are shown black). In other words, the SFN algorithm unnecessarily splits the upper cluster into two subsets: $S_a = \{8, 9, 11, 12, 14, 15\}$ and $S_b = \{10, 13\}$, yielding a total of three clusters.

Why does this happen? Because the SFN algorithm does not actually build a graph, and hence cannot use all the information held in the FNG. It is possible to use the FNG to determine that the two subsets $S_a$ and $S_b$ belong together, by considering paths that connect points in one subset with those in the other. For example, there is a path from point 10 in $S_a$ to point 8 in $S_b$: $(10 \rightarrow 2 \rightarrow 13 \rightarrow 4 \rightarrow 8)$. When the CFNG algorithm colors the graph, points 10, 13, and 8 can all receive the same color, because they are not joined by edges. Notice that this path from 10 to 8 has an even number of edges, and that, in a tree, any two vertices separated by such an even-length path can be colored the same. Such even-length paths can be found from any vertex in $S_a$ to any in $S_b$, so the CFNG method colors $S_a$ and $S_b$ the same. Thus the CFNG method does not split the top cluster.

The above discussion gives an example where the SFN algorithm creates splits which are avoided by the CFNG algorithm. It is easy to find other examples where this also occurs. However, it does not prove that the CFNG algorithm *always* avoids such splits. The next section will discuss such splits in more detail.

## 4. Hasty splits

In many methods that build a partition hierarchy, the resulting hierarchy may be of poor quality. Sometimes groups of objects that make up an obvious cluster end up separated on far-apart subtrees, whereas a good hierarchy would group such objects together into the same subtree. This occurs for partitions that split sets at the median, and for the *k*-means algorithm. The CFNG algorithm can suffer from this problem too.

Fig. 9 illustrates the problem. It shows a set of 2D points, clearly grouped into three clusters, $A = \{0, 1, 2\}$, $B = \{3, 4, 5, 6\}$, and $C = \{7, 8, 9\}$. Fig. 9a shows the first split made by the CFNG algorithm. Notice that not all points in $B$ get the same color: 6 is white, but 3, 4, and 5 are black. As a result, the first split joins part of $B$ with cluster $A$ above (into $S_1$), and part of it with cluster $C$ below (into $S_2$). Recall that the first split partitions objects into the two main subtrees of the hierarchy. Because cluster $B$ was cut by the first split, some of its members end up in one subtree, and some in the other. In other words, the structure of the tree does not reflect the cluster structure of the data. This mismatch appears in Fig. 9b: point 6 ends up in subtree $S_2$ at the bottom, not in subtree $S_1$ which contains its nearest neighbors 3, 4, and 5.

Even if the first split had not cut center cluster $B$ in two, that cluster would eventually have been split into smaller parts. The reason that $B$'s elements end up in two different subtrees is that $B$ was split too early. We shall call such splits *hasty*.

Why does this happen? The ultimate reason is that the binary partition is blind: each pass of the algorithm splits a set into two parts, regardless of whether or not such a split is suitable for the data in question. It must be emphasized that the problem is not binary partition itself: a set of objects can be partitioned into two subsets in many ways, and some splits do obey the cluster structure of the data. For instance, a partition of the points in Fig. 9a into the two subsets $\{0, 1, 2, 3, 4, 5, 6\}$ and $\{7, 8, 9\}$ does indeed obey the cluster structure. However, making this particular partition requires knowing which points comprise each cluster. This knowledge is by definition not available *a priori*, since the algorithm's goal is to discover the clusters in the first place.

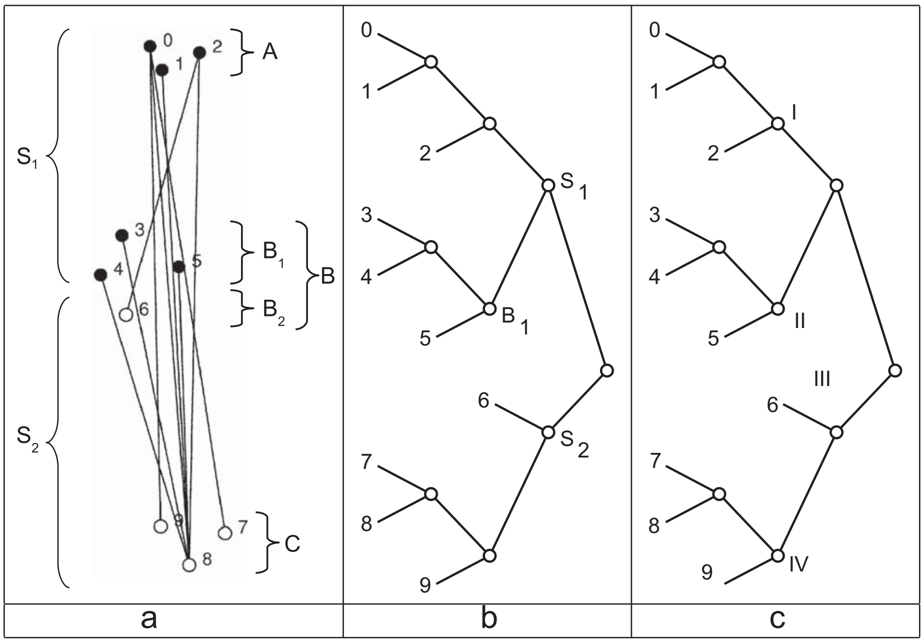


**Fig. 9.** (a) A, B, and C are three clusters of points on the plane; their FNG is shown. After the first partition step, the coloring algorithm splits $B$ into subclusters $B_1$ and $B_2$. (b) The final tree obtained after hierarchical coloring. Notice that point 6 ends up on a separate branch of the tree from other points in its cluster. (c) Subtrees I–V are homogeneous (i.e., they contain only points from the same cluster).

### 4.1. Overcoming hasty splits

Just as a given set can be split in many ways, there are many ways to improve the quality of the splits. Several strategies were studied:

1. *Cure*: In this approach, initially the partitional hierarchy is built completely. In it, each subtree represents cluster of objects. Then, hasty splits are identified, and corrected. A correction involves finding two subtrees that are far apart in the tree, but close in distance (such as the leaf node 6 and the subtree $B_1$ in Fig. 9b) and bringing them together to be sibling tree nodes.
2. *Prevention*: Instead of allowing the recursive partition process to proceed to completion, this approach tries to ensure that each split does not cut clusters unecessarily. If a split is determined to be of poor quality, it is adjusted before the two subsets are partitioned further recursively.
3. *Hybrid*: The approaches above are either very costly or ineffective. As a compromise, this last approach allows a few levels of splitting to proceed. With these splits, the quality of the splits can be better evaluated. If a hasty split is found, clusters are re-grouped, and split again.

The "cure" approach is applied at the end, after all the recursion has proceeded to its deepest level. On the other hand, the "prevention" approach does not allow recursion to proceed until a good split is found. Finally, the "hybrid" compromise involves advancing recursively a few steps, and occasionally backtracking. In the next subsections, we will discuss these three approaches in more detail.

#### 4.1.1. Cure

To overcome these hasty cluster splits, they must first be detected. Recall that each internal node in the hierarchy represents a cluster at some level: higher nodes represent large clusters, and lower nodes represent subclusters. Due to hasty splits, two nodes which represent clusters near each other in space can end up located far from each other on the tree.

To define this problem more precisely, we need ways to measure the distance between two clusters $A$ and $B$: $d_t(A,B)$ is the tree distance, defined as the number of tree links (parent and child pointers) that separate the two tree nodes corresponding to $A$ and $B$, while $d_m(A,B)$, is the usual metric distance (e.g., single-link or complete-link) between two clusters of objects.

Three clusters are needed to decide fairly whether objects have been grouped badly. Let $A$ be a cluster, $N$ be a cluster close to $A$ in the tree (e.g., its sibling), and $F$ be a cluster far from $A$ in the tree (a "distant relative"). Let $\tau = d_t(A,N)/d_t(F,N)$ be a ratio of tree distances. Given our assumptions, we expect $\tau$ to be small. Now, let $\mu = d_m(A,N)/d_m(F,N)$ be the corresponding ratio of metric distances. If $\mu$ is large, then $N$ belongs near to $F$, not to $A$, and hence it should also be close to $F$ in the tree.

Actually finding three badly grouped clusters requires that all possible triplets of clusters $(A,N,F)$ be examined, looking for cases where $\tau$ is small and $\mu$ is large. Such an approach will be very time-consuming: computing a single inter-cluster metric distance $d_m(F,N)$ will cost $O(n^2)$ time, and if the data set contains $n$ objects, then the tree will contain $O(n)$ clusters. Hence testing all triples will cost $O(n^5)$ time! We might restrict the search to cases where $A$ and $N$ are *very* close in the tree (e.g., siblings), thus restricting the search to pairs of clusters $A$ and $F$, not triples, but the resulting time cost is still in $O(n^4)$.

Another problem is that many instances may be found where $\mu$ is large. For instance, in the tree shown on Fig. 9b, point 6 is "estranged" from cluster $B_1 = \{3,4,5\}$, but its $\mu$ ratio will also be large for several subclusters within $B_1$. It is not clear how the tree should be repaired once the hasty split has been detected.

#### 4.1.2. Prevention

Because a "cure" is expensive, we also consider "prevention". In this approach, a split is examined before further subdivision, and hopefully hasty splits may be prevented.

Suppose a set $S$ of objects has been split into two disjoint subsets $S_1$ and $S_2$. The hastiness of this split may be detected by comparing single-link distances to complete-link distances. The single-link (nearest pair) distance $d_s(S_1,S_2)$ between $S_1$ and $S_2$ will be relatively small after a hasty split, compared to the complete-link (farthest pair) distance $d_c(S_1,S_2)$. In Fig. 9a, points $4 \in S_1$ and $6 \in S_2$ are the nearest pair between $S_1$ and $S_2$. In the same figure, points $0 \in S_1$ and $8 \in S_2$ are the farthest pair. We can suspect a hasty split when

$$d_s(S_1,S_2) \ll d_c(S_1,S_2). \qquad (1)$$

The single- and complete-link distances can also be used to correct a hasty split (these distances are defined above in Section 2.2). Eq. (1) assumes that the nearest pair of objects belongs to a hastily split subcluster. By computing the single-link distance $d_s(S_1,S_2)$, we will identify these two objects. Conversely, the farthest two objects (identified by computing the complete-link distance $d_c(S_1,S_2)$) will belong to subclusters that were probably not cut inappropriately.

These four objects can then be used to correct the hasty split. They can be used as "seeds" to build new subsets, as follows. First create four subsets, each containing a single object: one of the seeds. Then, test the remaining objects, and identify the seed closest to it. Add the object to that seed's subset. This yields four subsets. Finally, merge the two subsets whose seeds were the nearest pair into one of the other subsets, yielding a total of two subsets. For example, in Fig. 9a, 4 and 6 are the nearest-pair seeds, while 0 and 8 are the farthest pair. Testing the remaining objects yields the four subsets $\{5,6\}$, $\{3,4\}$, $\{0,1,2\}$, and $\{7,8,9\}$. Merging the first three subsets yields two subsets $\{0,1,2,3,4,5,6\}$ and $\{7,8,9\}$. This is a partition that does not split the middle cluster $B$ hastily.

### 4.2. Hybrid

Unfortunately, the prevention strategy just described is not foolproof. Fig. 10 shows an example where prevention fails. The original input set clearly contains three clusters of points. Moreover, building and coloring the FNG yields a hasty split: the top-left and right-most clusters are both cut in two. The prevention algorithm will find the nearest-pair seeds 3 and 4, and the farthest-pair seeds 4 and 8. Notice that point 4 belongs to both a farthest- and a nearest-pair. Grouping onto nearest seeds yields three subsets: $\{0,1,3\}$, $\{4,5\}$, and $\{2,6,7,8\}$. No two of these subsets can be merged without cutting one the three clusters of points. This example shows that the situation shown in Fig. 9, which motivated the prevention method, does not capture all the ways that hasty splits can occur.

The true problem is this: in a colored-FNG partition, each subset consists of objects that are all known to be far away from some extreme objects; however, knowing that objects in a group are all distant from some object tells us nothing about the mutual distances of objects *within* that group. The objects might all form a compact cluster, or might be widely dispersed.

This example makes it clear that the "prevention" strategy has problems. On the other hand, the "cure" strategy can detect arbitrary hasty splits, if we compare many tree nodes, and find "lost siblings". Unfortunately, a huge number of such comparisons may be required.

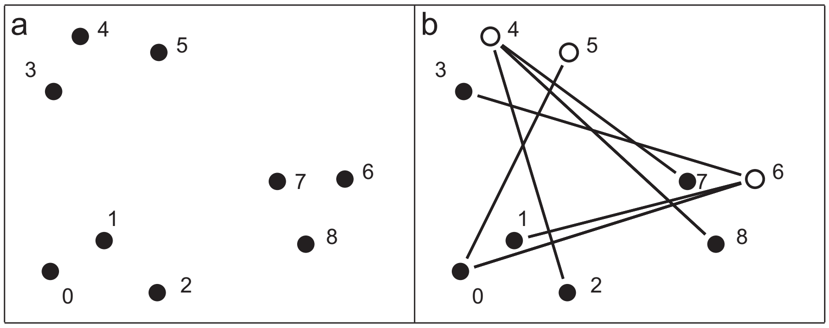


**Fig. 10.** (a) Three clusters of points, and (b) their colored FNG. The resulting partition is hasty, but this fact cannot be easily detected.

The only way to properly assess the quality of a split, is to split each subset further, and compare the distances between the smaller subsets thus obtained. Unfortunately, this kind of recursive splitting cannot be allowed to proceed too far, lest a huge number of small subsets is produced. The number of inter-subset distances that need to be evaluated also will be huge, as discussed above.

A compromise is to evaluate the quality of a split by performing ONE more split: the two subsets are further split into four smaller subsets. Then, distances between these four subsets can be obtained, and obvious hasty splits detected. If a set $S$ is split into subsets $A$ and $B$, then $A$ is split into ($c$ $d$) and $B$ into ($e$ $f$), comparisons between $c$, $d$, $e$, and $f$ can detect obvious hasty splits. If, for instance, we learn that ($c$, $d$, and $f$) belong together, the initial split can be revised, into two subsets $A' = c \cup d \cup f$ and $B' = e$.

This idea is embodied in a hybrid approach, and appears as Algorithm 2. When a subset is split into two, the two parts are themselves split again, and the four resulting subsets are tentatively pushed on a stack. Subsets are then popped from the stack to test for hasty splits. If the subsets indicate that the previous split was hasty, then they re-grouped into two subsets, and are discarded. As subsets are examined and split, a partition hierarchy is built, in a manner similar to that shown in Algorithm 1.

**Algorithm 2.** Stack-based hierarchy construction.

```
Data: $S$ is a set of objects in a metric space
BuildTreeWithStack($S$)
  clear stack
  $(A,B) = \text{split}(S)$
  push($A$), push($B$)
  $root \Leftarrow$ a new tree node
  $root.(left,right) \Leftarrow (A,B)$
  while stack is not empty do
    $B \Leftarrow \text{pop}()$
    $A \Leftarrow \text{pop}()$
    if $|A| \neq 1$ or $|B| \neq 1$ then
      $(c,d) \Leftarrow \text{split}(A)$
      $(e,f) \Leftarrow \text{split}(B)$
      if one of c, d, e, f is in the wrong subset then
        adjust $A$ and $B$ (merge one of $c,d,e,f$ into $A$ or $B$)
        $(c,d) \Leftarrow \text{split}(A)$
        $(e,f) \Leftarrow \text{split}(B)$
      end
      $A.(left,right) \Leftarrow (c,d)$
      $B.(left,right) \Leftarrow (e,f)$
      push($c$), push($d$), push($e$), push($f$)
    end
  end
  return root
```

## 5. Implementation details

In this section, we focus on some details of the implementation of the CFNG algorithm. The algorithm's input is a set of metric-space objects, and its output is a binary tree. In this tree, each node's two children are clusters of nearby objects. Leaf nodes store single objects.

The input objects are initially placed in a linked list, and this list is re-arranged with each partition. Within the list, a cluster of objects corresponds to a contiguous sublist: the first and last nodes in the sublist delimit the cluster. Objects are processed in a way that closely resembles quicksort, which also partitions a list of objects into two contiguous sublists, and then processes each sublist recursively.

Given a cluster of objects, the algorithm builds their FNG, and colors it. The objects are assumed to have no coordinates, so no convex hull or other similar acceleration structure can be obtained. Hence, the FNG must be built by brute force.

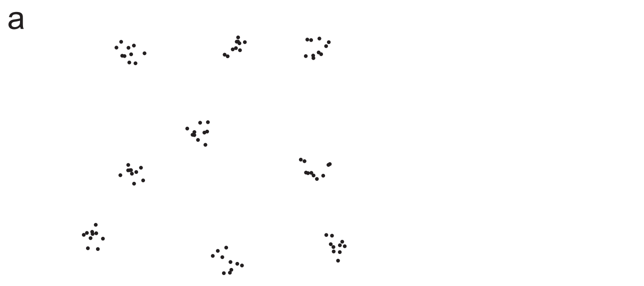
Once the FNG has been colored, the objects in the sublist are partitioned by color. This can be done with a single pass through the sublist. As a result, the objects are re-arranged into two smaller contiguous sublists: objects colored 0 are moved to the first sublist, and objects colored 1 to the second one.

As mentioned above in Section 3.3.1, each object has a multi-digit label that records all the 0 and 1 digits in its coloring history. After the hierarchy has been built, the objects can be sorted by label. One advantage of keeping all the objects in a single contiguous list is that this final sort is not needed. Because each sublist is partitioned with objects colored 0 coming before objects colored 1, the final multi-digit labels will naturally occur in increasing lexicographic order on the linked list. In other words, the sorting is performed at the same time that the hierarchy is being built. This is another way in which the CFNG algorithm resembles quicksort.

The objects could have been stored in an array instead, but using a linked list facilitates tree surgery during the correction of hasty splits. When a node in the tree is moved next to the other node that has been deemed its "natural" sibling, the objects in the node's cluster must also be moved in the list. If an array had been used, this operation would cost $O(n)$ time. However, with a linked list the same result is achieved by changing a few pointers to re-splice the list.

## 6. Results

This section presents an evaluation of the cluster analyses produced by the CFNG algorithm. Evaluating the quality of a cluster analysis algorithm is not straightforward. Bottom-up (agglomerative) methods produce different kinds of hierarchies from top-down (partitional) methods, and the two cannot be compared easily. However, we can use *a priori* information to



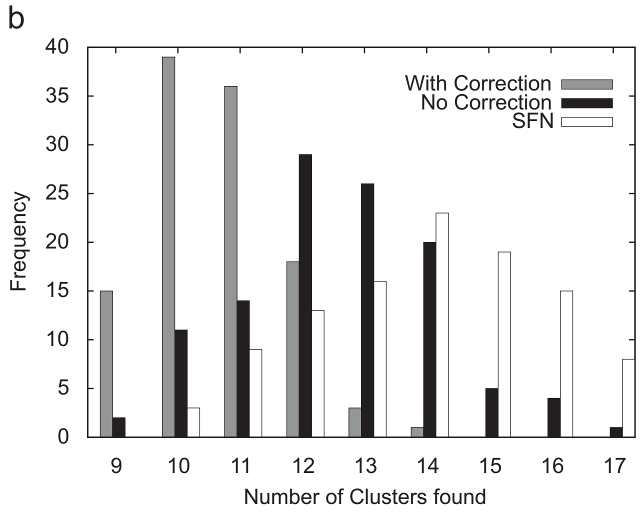


**Fig. 11.** (a) A typical synthetic test case, with nine clusters of 2D points. (b) Distribution of the number of clusters detected by the CFNG algorithm when applied to 112 test cases, with and without correcting for hasty splits. The number of clusters detected by the SFN algorithm is shown for comparison.

determine the effectiveness of a cluster analysis algorithm. We choose to use this approach.

### 6.1. A priori evaluation

To evaluate the quality of the hierarchies produced by the CFNG algorithm, we can generate a known set of clusters of objects, produce a hierarchy, and then compare the clusters found by the hierarchy with those known to exist *a priori*. That is the approach we followed.

As an experiment, we generated nine randomly placed centers on the unit square, and generated a cluster of 10 points randomly distributed around each center. We then fed this data into our algorithm, and produced a hierarchy of clusters. With some effort, the hierarchy can be inspected visually to determine whether each original cluster is grouped into its own independent subtree.

However, we wished to run this experiment many times, and the problem then arose as to how to identify subtrees that correspond to known clusters. The solution we chose was to identify *homogeneous* subtrees. A homogeneous subtree is one for which all the leaves belong to the same known cluster. For example, consider the tree shown in Fig. 9c. It has *four* homogeneous subtrees: I, II, III, and IV. Subtree I contains points 0, 1, and 2, which all belong to the top cluster. Similarly, subtree II contains 3, 4, and 5, which belong to the middle cluster. Subtree III contains the single point 6 from the middle cluster, and finally subtree IV's points also all belong to the bottom cluster. Had a hasty split not occurred, the tree would contain only *three* homogeneous subtrees, matching the three clusters in the data.

Homogeneous subtrees can be identified with a simple algorithm: first label each leaf node with the ID of its *a priori* cluster. Then, propagate the IDs upwards: any node whose children both have the same ID receives that ID in turn. However, at any node whose children have different IDs, the upward propagation stops: such a node is not homogeneous, but its children are the roots of homogeneous subtrees.

The results of this experiment are illustrated in Fig. 11. Fig. 11a shows a typical input to the algorithm. The size of each cluster is kept small, and the centers of the clusters far apart, to ensure that clusters do not overlap. Fig. 11b shows the result of applying recursive Algorithm 1 (without hasty-split corrections) to 112 different inputs, each with nine clusters of 10 points each. This is a frequency plot of the number of clusters found (the number of homogeneous subtrees). Ideally, nine clusters should be found, although often more than nine are found, for the reason explained just above. For comparison, the number of clusters found while invoking the hybrid hasty-split-fix method (Algorithm 2) is also shown. Without this method, the algorithm finds more than nine clusters quite often. However, even with it, the algorithm still often tends to split one or two clusters hastily: the approach in Algorithm 2 is not foolproof, and some hasty splits still get through. Finally, Fig. 11b also shows the frequencies of the number of clusters found by the SFN algorithm. For reasons discussed in Section 3.4 above, the SFN algorithm is more prone to splitting natural clusters than the present CFNG algorithm. Not surprisingly, the plot shows that SFN finds even more clusters than the CFNG algorithm without the hasty-split-fix correction. We did not study the result of adopting the hasty-split-fix correction to the SFN algorithm. This was because too many cases would have to be considered: the SFN algorithm can split a cluster into more than two parts, so the hybrid algorithm would potentially have to examine a large number of subclusters (not just four) to see if a bad split had occurred. Given $n$ subclusters, $2^n$ possible groupings would have to be considered.

It is important to emphasize that the CFNG algorithm is not alone in producing hasty splits. Many other partition-based methods suffer from this problem, including the popular $k$-means algorithm. As discussed above, an initial split cannot possibly be perfect, because the full cluster structure of the data is not known at the outset.

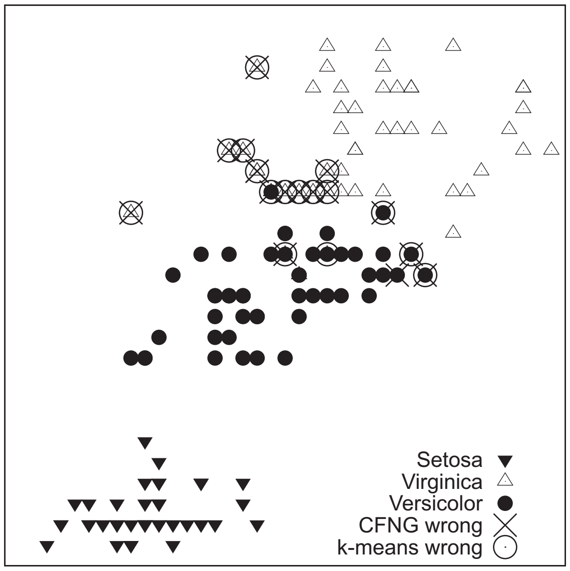


**Fig. 12.** Anderson's data set, showing three classes of iris specimens. Note that two of the three classes are not linearly separable; both $k$-means and the CFNG method misclassify some specimens near the boundary between the Versicolor and Virginica classes.

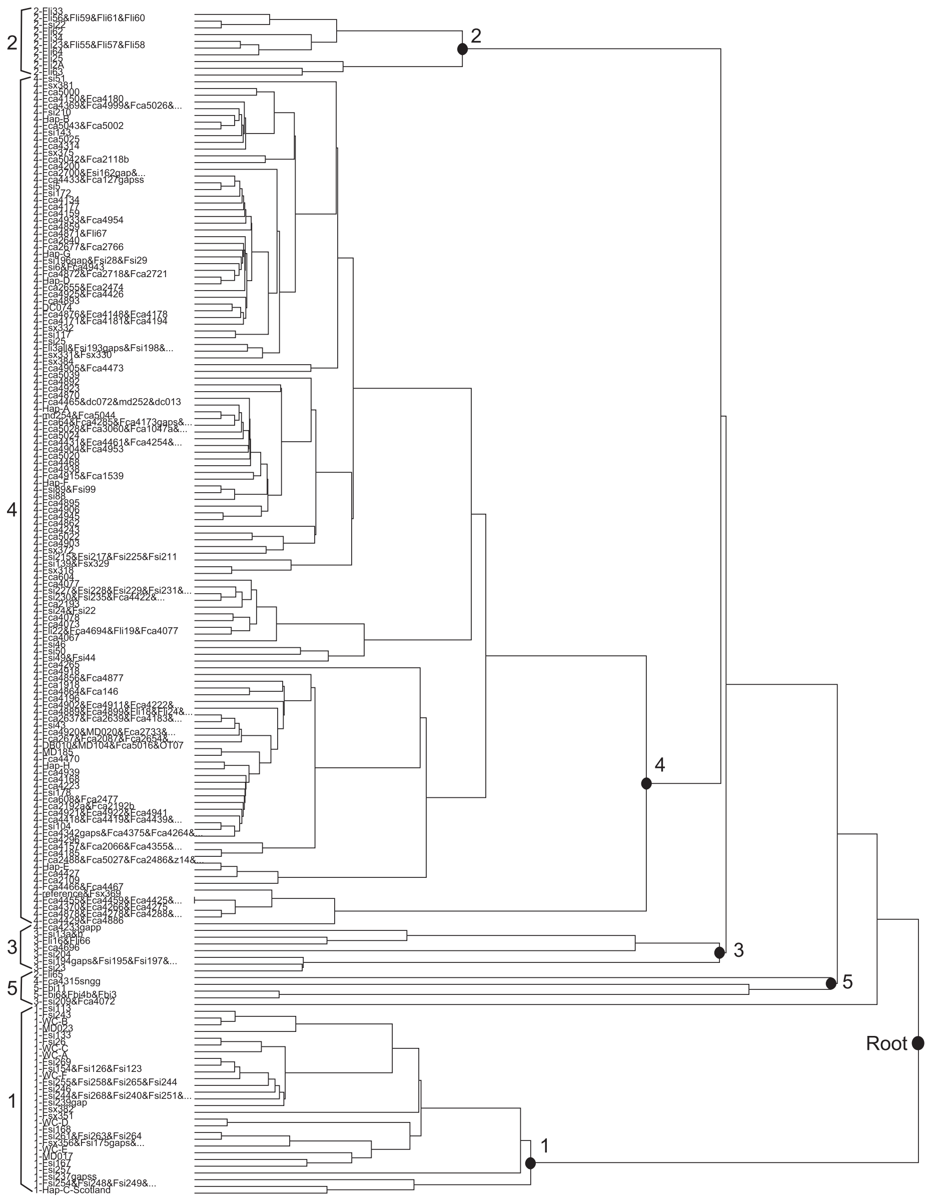


**Fig. 13.** Classification of genetic samples from five subspecies of domestic and wild cats.

## 6.2. Real-world data

Anderson's iris data set is a classic test for clustering algorithms. It consists of 150 sets of four measurements of lengths of flower parts (a four-dimensional data set). Three species of iris flowers (*virginica*, *setosa*, and *versicolor*) are present in the data, shown a 2D projection on Fig. 12. If the $k$-means algorithm is applied to this data, with $k = 3$, specimens from all three species are mostly classified correctly, but some specimens from the Versicolor and Virginica species are grouped with the wrong species (classification errors are marked with an X). If Algorithm 2 is used instead, and the partition hierarchy is truncated into three major subtrees, an almost identical classification is obtained, although some specimens are again mis-classified (they are marked with circles). The most likely reason for mis-classification is that the data itself is not linearly separable (no hyperplane can be placed to separate the data points for those two species). For embedded data such as this, the CFNG algorithm will tend to split clusters along a hyperplane (which is the perpendicular bisector of the line joining two extreme points), and the $k$-means algorithm has the same limitation.

Finally, the method was applied to non-embedded data. These data are a set of 176 DNA sequences, obtained from wild and domestic cats. These sequences were made available by Driscoll et al. in an article in Science [7].

Fig. 13 is a dendrogram, which shows the results of applying Algorithm 2 to the cat-genome data. In this figure, the internal tree nodes are shown on the right side, with the root right-most. The leaf nodes are shown on the left side, and identify each genetic sample. The first digit on the label of each leaf indicates the subspecies (clade) obtained in Driscoll et al.'s hierarchy. Each tree node represents a cluster, and the (average-link) distance between two sibling clusters is proportional to the $x$ coordinate of

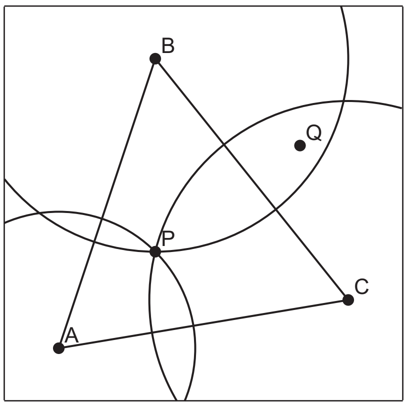


**Fig. 14.** If $P$ is inside a triangle, it cannot be farthest from another point.

their parent: parents more on the right side of the image have more widely separated children.

Five main subtrees are highlighted, corresponding to five main subtrees obtained by Driscoll et al. from the same DNA sequences, using an agglomerative hierarchy (their hierarchy is not shown). The roots of these subtrees are marked on the right side, and the corresponding leaf nodes are grouped on the left. If we use Driscoll et al.'s groupings as the known clusters, we find that subtrees 1, 2, and 4 are homogeneous (our classification matches theirs exactly). However, subtree 3 in our result contains one extraneous item, and subtree 5 contains three extraneous items. The non-homogeneity of these subtrees can be seen visually in the dendrogram: although these trees have very few leaves, their roots are far on the right, indicating that the differences between these leaf items are substantial.

The differences between our tree and the one built by Driscoll et al. may arise because two different approaches were used: our approach is divisive, while theirs is agglomerative. Moreover, the authors point out that their agglomerative hierarchy may not be perfect: they report that another marker (mitochondrial DNA), which was not used in the agglomerative hierarchy, places a few specimens in clades different from those found by agglomeration.

## 7. Conclusion

We have presented CFNG, a new algorithm for cluster analysis of objects in a metric space. The algorithm improves on an existing method. We have also presented a technique for improving the quality of the splits obtained in partition hierarchies. The latter technique is beneficial not only in our method, but in any method that builds a hierarchy of objects top-down. The brute-force method used to build the FNG runs in $O(n^2)$ time, which is too slow for large data sets. This suggests an important avenue for future research.

### 7.1. Future work

For objects without coordinates, the graph-building algorithm might be accelerated with a construction analogous to the convex hull. Looking at Fig. 4c, we can see that the hull points, for some reason, tend have many neighbors. In other words, many "interior" points have the same common farthest neighbor. Perhaps this definition may be used as an alternative to the convex hull: hull objects in a metric space might be defined as those objects that have multiple neighbors in the FNG. Hopefully, objects with that property make up a small subset of all the objects. If these hull objects could be identified quickly (in less than $O(n^2)$ time), then graph-building algorithm might be sped up substantially.

## Appendix A. Hull property of the FNG

For two-dimensional objects, all extreme objects (points) must belong to the input set's convex hull. To prove this, we need the following lemma:

**Lemma 1.** *If a point $P$ is inside a triangle and $Q$ is any other point, then at least one of the triangle's vertices is farther from $Q$ than $P$ is.*

**Proof.** Suppose the triangle has vertices $A$, $B$, and $C$, and let $P$ be inside the triangle. Build three circles: $C_A$, $C_B$, and $C_C$, centered at $A$, $B$, and $C$, respectively, and all passing through $P$. The three lozenge-shaped intersections $C_A \cap C_B$, $C_A \cap C_C$, and $C_B \cap C_B$ must be mutually disjoint, because $P$ is inside the triangle (see Fig. 14). Hence the intersection of all three circles $C_A \cap C_B \cap C_C$ has zero area. Thus, if $Q$ is any other point, it must be outside at least one circle. If, say, $Q$ is outside circle $C_A$, then $|A-Q| > |P-Q|$. □

With this lemma, we can establish our claim:

**Proposition 1.** *If $S$ is a set of two-dimensional points, and $f \in S$ is the farthest from some $q \in S$, then $f$ must belong to the convex hull of $S$.*

**Proof.** A point $h$ is in the convex hull of $S$ iff it is inside no other triangle $(A,B,C)$, for all $A,B,C \in S$. Suppose $f$ is *not* a hull vertex. Then it is inside some triangle. Then, by the lemma, one of $A$, $B$, or $C$ must be farther from $q$ than $f$ is from $q$. Thus $f$ is not farthest $q$, as was originally stated. This is a contradiction. □

The proposition extends naturally to three and higher dimensions. In higher dimensions, the triangle is generalized to a simplex. Notice that this proof requires that the objects belong to a Cartesian space. For non-embedded objects, the concept of a convex hull is inappropriate, because it depends on convexity, which in turn is defined in terms of linear interpolation between objects, and this requires coordinates.

## References

[1] K. Appel, W. Haken, J. Koch, Every planar map is four colorable, J. Math. 21 (1977) 439–567.
[2] J.L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (9) (1975) 509–517.
[3] J.L. Bentley, H.T. Kung, M. Schkolnick, C.D. Thompson, On the average number of maxima in a set of vectors and applications, J. ACM 25 (1978) 536–543.
[4] P. Berkhin, Survey of clustering data mining techniques, Technical Report, Accrue Software, San Jose, CA, 2002.
[5] D.R. Chand, S.S. Kapur, An algorithm for convex polytopes, J. ACM 7 (1970) 78–86.
[6] D. Eppstein, Updating widths and maximum spanning trees using the rotating caliper graph, Technical Report 93-18, UC Irvine CS Department, 1993.
[7] C.A. Driscoll, et al., The near eastern origin of cat domestication, Science 317 (2007) 519–523.
[8] R. Finkel, J.L. Bentley, Quad trees: a data structure for retrieval on composite keys, Acta Inf. 4 (1) (1974) 1–9.
[9] S. Fortune, A sweepline algorithm for Voronoi diagrams, Algorithmica 2 (1987) 153–174.
[10] X. Gao, B. Xiao, D. Tao, X. Li, A survey of graph edit distance, Pattern Anal. Appl. (2009), doi:10.1007/s10044-008-0141-y.
[11] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: 1984 ACM SIGMOD International Conference on Management of Data, 1984, pp. 47–57.
[12] A.K. Jain, M.N. Murty, P.J. Flynn, Data clustering: a review, ACM Computing Surveys 31 (3) (1999) 264–323.
[13] X. Jiang, H. Bunke, J. Csirik, Median strings: a review, in: M. Last, A. Kandel, H. Bunke (Eds.), Data Mining in Time Series Databases, World Scientific, Singapore, 2004, pp. 173–192.

[14] J.H. Ward Jr., Hierarchical grouping to optimize an objective function, J. Am. Stat. Assoc. 58 (1963) 236–244.
[15] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller, J.W. Thatcher (Eds.), Complexity of Computer Computations, Plenum, New York, 1972.
[16] L. Kaufman, P. Rousseeuw, Finding Groups in Data: An Introduction to Cluster Analysis, Wiley, New York, 1990.
[17] B. King, Step-wise clustering procedures, J. Am. Stat. Assoc. 69 (1967) 86–101.
[18] J.B. MacQueen, Some methods for classification and analysis of multivariate observations, in: 5th Berkeley Symposium on Mathematical Statistics and Probability, vol. 1, University of California Press, Berkeley, CA, 1967, pp. 281–297.
[19] E. Mayr, E.G. Linsley, R.L. Usinger, Methods and Principles of Systematic Zoology, McGraw-Hill, New York, 1953.
[20] B. Mirkin, Concept learning and feature selection based on square-error clustering, Mach. Learn. 35 (1999) 25–40.
[21] B. Mirkin, Clustering for Data Mining, Chapman & Hall, Boca Raton, FL, 2005.
[22] S. Rovetta, F. Masulli, Shared farthest neighbor approach to clustering of high dimensionality, low cardinality data, Pattern Recognition 39 (12) (2006) 2415–2425.
[23] P.H.A. Sneath, R.R. Sokal, Numerical Taxonomy, Freeman, London, UK, 1973.

**About the Author**—ALEJO HAUSNER holds a B.Sc. and an M.Sc. in Physics from McGill University, an M.Sc. in Computer Science from Queen's University, and a Ph.D. in Computer Science from Princeton University. His research interests range from computer graphics through computational geometry and cluster analysis. He is currently an Associate Professor at Daniel Webster College.