Multivariate Behavioral Research

Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/hmbr20

An Efficient Metric Combinatorial Algorithm for Fitting Additive Trees

James E. Corter

Available online: 10 Jun 2010

To cite this article: James E. Corter (1998): An Efficient Metric Combinatorial Algorithm for Fitting Additive Trees, Multivariate Behavioral Research, 33:2, 249-271

To link to this article: http://dx.doi.org/10.1207/s15327906mbr3302_3

Multivariate Behavioral Research, *33* (2), 249-271

An Efficient Metric Combinatorial Algorithm for Fitting Additive Trees

James E. Corter
Teachers College, Columbia University

A new combinatorial algorithm for fitting additive trees to proximity data is described. This algorithm, termed the "generalized triples" or *GT* method, proceeds by examining all triples of objects *x,y,u* in relation to the remaining set of objects to be clustered. For a given focal object, say *x*, the algorithm determines whether *y* or *u* is *x*'s nearest neighbor using estimates derived from the distances of these objects to each other and the saved sums of distances of these objects to the remaining objects in the set. The result is a basic computational loop that is approximately order(n^3). This idea is applied in a sequential agglomerative algorithm, with all pairs of objects that are mutual nearest neighbors (based on the above estimates) being joined at each stage. A simple version of the algorithm can be proven to find the correct solution if the dissimilarities matrix **D** actually satisfies the additive tree metric. The algorithm also works well on errorful data (i.e. data that cannot be modeled perfectly by an additive tree). A simulation study demonstrates that the *GT* algorithm works as effectively as the Sattath and Tversky algorithm (Corter, 1982; Sattath & Tversky, 1977) in terms of fit of the obtained solutions, and is faster for moderate- to large-sized data sets, especially in the presence of error. A second simulation study shows that the *GT* algorithm obtains comparable fits to De Soete's (1983) algorithm, with large savings in computation time.

Additive trees have proved to be useful representations for proximity data in the behavioral sciences. This usefulness has been demonstrated by a wide variety of applications to such problems as mental representation of concepts (e.g., Abdi, Barthélemy, & Luong, 1984), consumer choices (DeSarbo,

---

The author wishes to dedicate this work to the memory of Amos Tversky, whose intellectual accomplishments and unflagging enthusiasm for ideas were enormously inspirational to his students, friends, colleagues, and the larger research community.

Computer programs written in the PASCAL language implementing the *GT* and *STC* algorithms described in this article are available over the World-Wide Web. PASCAL source code for the programs "GTREE" (algorithm *GT*) and "ADDTREE" (algorithm *STC*) and documentation are maintained on the NetLib resource at Lucent Technologies' Bell Laboratories. See http://netlib.bell-labs.com/netlib/mds/index.html for information on obtaining files. For those without access to a PASCAL compiler, DOS-executable version of the programs may be available. See the author's home page at http://www.columbia.edu/~jec34 for current information on availability.

Portions of this research were presented at the annual meetings of the Classification Society of North America in East Lansing, MI, in June 1992, and in Washington, DC, in June 1997. Thanks to Geert De Soete for providing a copy of his LSADT program, and to Thomas Smith for his advice on compiling it for various PC configurations.

Manrai, & Manrai, 1993), perceptions of societal risks (Johnson & Tversky, 1984), organizational structure (e.g. Clinchy, 1990), mental organization of environments (Taylor & Tversky, 1992), and the historical relationships among languages and dialects (e.g. Corter, 1989). Additive trees, like their simpler counterpart, ultrametric trees, are typically applied to matrices of proximity data (i.e., similarities or dissimilarities), one familiar example being a correlation matrix.

The usefulness of additive trees for analyzing proximity matrices may be illustrated by the following example applications. In one study by Johnson and Tversky (1984), subjects were given a similarity rating task using 18 societal risks (for example, *terrorism, automobile accidents, lung cancer*) as stimuli. Subjects rated how similar they thought each pair of risks were. The additive tree solution (see Corter, 1996, p. 42) grouped the risks by overall similarity, revealing highly interpretable clusters of risks (for example, one cluster grouped all the illnesses, while another grouped the technological disasters *nuclear accident* and *toxic chemical spill*). The specific nature of these clusters may be used to infer what factors subjects take into account in comparing risks. As another example, Beller (1990) used additive trees (among other techniques) to analyze the relationships among subtest scores and among single test items in a scholastic aptitude test used to screen applicants to Hebrew University. In one analysis of single test items, she selected a representative 42-item subset of the entire pool of 230 items, then analyzed the correlations among pairs of the selected items. The additive tree solution for these data indicated two major groups of items: one including the items measuring knowledge, and the other including items related to problem-solving skills. Within each of these large groupings were more specific clusters of items (for example, a cluster comprised solely of the "numerical series" items). The additive tree solution shows not only which items cluster together, but also the relative dissimilarity of each of these groups of items to each other. Note that the decision to analyze only a subset of the 230 test items may very well have been dictated by limitations of the additive tree and multidimensional scaling software used in the analyses.

In general, two factors have inhibited wider use of additive trees in such research applications. First, additive tree algorithms have only recently become available on mass-distribution statistical packages, and therefore are less widely known than other techniques. Second, the algorithms that have been available to fit additive trees to data are relatively inefficient. For example, one algorithm widely used at present for fitting additive trees, the Sattath and Tversky (ST) algorithm (Sattath & Tversky, 1977; Corter, 1982), is a combinatorial algorithm that works in approximately order(n^4)log(n) time. This limits its usefulness with large data sets. Iterative

"mathematical programming" approaches for fitting additive trees (e.g. Carroll & Pruzansky, 1975; De Soete, 1983) are also quite costly computationally. For example, the most effective of several additive tree fitting algorithms described by Barthélemy and Guénoche (1991), the iterative "method of reduction", was more than an order of magnitude slower than the combinatorial algorithms they compared it with for their test cases.

Thus it seems that faster algorithms, whether combinatorial or based on iterative numerical methods, might spur wider use of additive trees as methods for analyzing proximities, both by enabling the analysis of larger data sets than can be handled by current algorithms and by encouraging wider implementation in mass-market statistical packages. Because the additive tree is a much less restrictive model than the ultrametric trees fit by "hierarchical clustering" methods and thus can accommodate a larger class of data structures, this greater availability would seem generally desirable from a data-analytic point of view (see Corter, 1996, for examples of data structures for which an additive tree seems more appropriate than an ultrametric). The present paper describes a new combinatorial algorithm that is theoretically faster than the widely used Sattath & Tversky algorithm by a factor of approximately order(n), making it a promising approach for the analysis of large data sets.

Some Properties of Additive Trees

In order for a matrix of distances to be representable by an additive tree, the distances must satisfy the so-called "additive tree inequality" or "four-point condition" (Buneman, 1971; Patrinos & Hakimi, 1972; Dobson, 1974 ; see Abdi, 1990, for a literature review of formal descriptions of additive trees and their properties). Letting x, y, u, and v denote objects in the set being analyzed, and $d(x,y)$ denote the dissimilarity or distance between objects x and y, this condition states that for every quadruple of objects in the tree $d(x,y)+d(u,v) \leq MAX[\, d(x,u)+d(y,v),\, d(x,v)+d(y,u)]$. This is equivalent to requiring that for any four objects there is *some* relabeling of the points as x,y,u,v such that

$$d(x,y)+d(u,v) \leq d(x,u)+d(y,v) = d(x,v)+d(y,u).$$

If the left-hand inequality is strict, then x and y are neighbors in the tree and so are u and v (Figure 1A, next page). If equality holds among all three sums, then the tree structure connecting these four objects is a singular tree (or "bush") with the leaf arcs for the four objects emanating from a single internal node (Figure 1B).

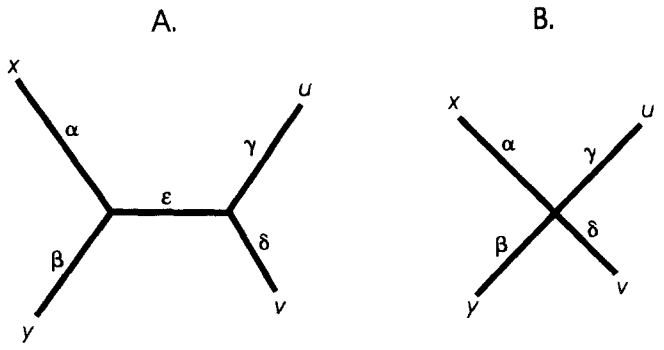

Figure 1
A. An unrooted additive tree on four objects. B. A singular tree.

The additive tree metric is a less restrictive condition than the ultrametric, which requires that for every triple of objects x,y,z, $d(x,y) \leq MAX\,[d(x,z), d(y,z)]$. In turn, the additive inequality implies the triangle inequality: $d(x,y) + d(y,z) \geq d(x,z)$ for all x,y,z. Thus the space of ultrametric distance matrices is nested within the space of additive tree matrices, which in turn is a subset of the set of matrices satisfying the axioms of a metric space, namely symmetry $[d(x,y) = d(y,x)]$, positivity $[d(x,y) > d(x,x) = 0]$, and the triangle inequality (see Critchley, 1986; Furnas, 1988).

Previous Algorithms for Fitting Additive Trees to Errorful Data

Algorithms proposed for fitting additive trees to data have generally been of two types: (a) combinatorial algorithms and (b) iterative numerical methods based on "mathematical programming" approaches (Arabie & Hubert, 1992; DeSoete & Carroll, 1996), though recently a different approach has been taken by Hubert and Arabie (1995). Their algorithm uses an iterative projection strategy to seek the solution that is closest to the data in a least-squares sense while satisfying a set of constraints implied by the tree model.

Combinatorial tree-fitting methods search through subsets of the objects to be clustered, computing statistics based on interobject distances in order to infer the optimal topology of the tree to be constructed. These algorithms are usually implemented as sequential agglomerative algorithms (Sneath & Sokal, 1972) that combine one or more pairs of objects at each stage, until only two or three "objects" (actually clusters corresponding to subsets of objects) remain. Estimation of the lengths of arcs in the tree is done either in parallel with the combining of objects at each stage, or after the tree topology is

completely determined. The numerical "mathematical programming" approaches generally recast the discrete combinatoric problem of finding the optimal tree topology into a related problem in a continuous parameter space. An initial solution is then iteratively adjusted towards one compatible with a discrete tree structure, guided by the use of a "penalty function". The penalty function usually incorporates two terms, one measuring departure from a tree structure and one measuring departure of the model distances from the data. The relative weight of the first component is gradually increased to insure that the final model distances are consistent with the tree inequality. This procedure is analogous to gradually imposing (0,1) discreteness constraints on the tree-structure parameters.

Combinatorial algorithms for fitting additive trees to data that may not perfectly satisfy the additive inequality have been introduced by Cunningham (1978), Sattath and Tversky (1977), Fitch (1981), Corter (1982), Guénoche (1987), Roux (1986), and Barthélemy and Luong (1986). Perhaps the most widely used of these is the Sattath and Tversky (1977) algorithm, referred to henceforth as algorithm ST. The variant of the algorithm introduced by Corter (1982) (algorithm STC) results in slightly improved performance compared with the original Sattath and Tversky algorithm (improved fit in about 10% of cases, with only a slight increase in processing time), and is available in at least one mass-market statistical package. For these reasons, and because the present "generalized triples" algorithm is based on the modification to ST introduced by Corter (1982), the ST and STC algorithms will be described below.

The first mathematical programming approach to fitting additive trees to data was proposed by Carroll and Pruzansky (1975). Their method was based on the idea that additive tree distances can be decomposed into the sum of an ultrametric distance matrix plus a matrix generated by single additive constants for each object. Graphically, this corresponds to decomposing the additive tree into an ultrametric tree plus a singular tree like that shown in Figure 1b. Another mathematical programming approach to the problem was developed by De Soete (1983). This method, referred to here as algorithm LSADT, works by iteratively adjusting the matrix of data dissimilarities, **D**, so that the dissimilarities among each quadruple of objects eventually satisfy the additive tree inequality. The goal is to find that matrix which everywhere satisfies the inequality and is as close as possible (in a least-squares sense) to the original matrix of dissimilarities. One of the advantages of this LSADT method is that it is easily extended to such problems as fitting trees to incomplete data (De Soete, 1984). Disadvantages of the mathematical programming methods include relatively high computational cost, the possibility of convergence to a local minimum (admittedly also a problem with combinatorial methods), and

the fact that the output from the basic method is simply a matrix of distances satisfying the tree inequality, so that actual construction of the tree and estimation of arc lengths remains to be performed.

The Sattath & Tversky (1977) Algorithm

Like all the methods considered in this article, the ST algorithm introduced by Sattath & Tversky (1977) accepts as input a matrix **D** of proximities between each pair of "objects", and outputs an additive tree graph displayed in rooted form. Leaves of the tree correspond to the objects being scaled, and internal nodes of the rooted tree may be thought of as representing "clusters" of objects (Corter, 1996; Sattath & Tversky, 1977).

The initial step of the *ST* algorithm converts the input proximities into distance-like numbers; that is, ensures that the data are dissimilarities satisfying the axioms of a metric space: symmetry, positivity, and the triangle inequality. If the data are similarities, they are converted to dissimilarities by subtracting them all from the maximal similarity value. In order to achieve symmetry if the data are asymmetric, corresponding entries in the top and bottom halves of the proximity matrix are averaged. An additive constant is then added to all entries in the matrix. That constant is selected to be the minimum value that results in the dissimilarities *exactly* satisfying the triangle inequality: that is, $d(x,y) + d(y,z) \geq d(x,z)$ for all x, y, z, with equality holding for at least one triple.

The *ST* algorithm proceeds in a sequential agglomerative manner. At each stage, one or more pairs of objects are selected to be combined into a cluster, as described below. For each pair selected (say, objects x and y) and combined into a common node, the lengths of the arcs under this common node are calculated, and the distance of the combined object xy to each other object z is calculated as the (weighted) average of $d(x,z)$ and $d(y,z)$. For each such pair combined at a given stage i, the proximity matrix is thus reduced in size from $n_i \times n_i$ to $(n_i - 1) \times (n_i - 1)$. Up to $n_i/2$ objects pairs may be combined on stage i. These stages continue until n_i is less than or equal to 3, at which point the topology of the tree is completely determined. The final objects are combined and the remaining arcs estimated, then the tree is output.

At a given stage, pairs of objects are selected to be combined as follows. Denote the matrix of "data distances" at stage i by $\mathbf{D}_i$. The algorithm looks through all quadruples of objects x, y, u, v in $\mathbf{D}_i$, and computes three sums: $S_1 = d(x,y) + d(u,v)$, $S_2 = d(x,u) + d(y,v)$, and $S_3 = d(x,v) + d(y,u)$. According to the tree inequality, if the true tree topology among these four objects is $(x\ y)(u\ v)$, then $S_1 < S_2 = S_3$. If the true structure is $(x\ u)(y\ v)$, then $S_2 < S_1 = S_3$, and if it is $(x\ v)(y\ u)$, then $S_3 < S_1 = S_2$. Thus with distances

perfectly satisfying the tree inequality, we can conclude that the two objects pairs corresponding to the smallest sum are neighbors in the true tree structure. It can be proven that if we assign a "neighbor score" of 1 to the two pairs in the smallest sum [say, pairs (x,y) and (u,v)], and a score of 0 to the other four object pairs, and then sum these scores over all quadruples of objects, that the maximal entry in the resulting "neighbor score" matrix **N** corresponds to two objects that are "nearest neighbors" in the true tree structure. Thus the entries $\mathbf{N}(x,y)$ in the neighbor score matrix can be thought of as a measure of closeness of the corresponding pair in the tree topology. Using this 1-0-0 scoring rule and combining only a single pair of objects at each stage results in an algorithm that can be proven to always find the true tree structure for errorless data, if one exists.

Of course, with real data, error of various kinds will in general prevent the two larger sums from being exactly equal, and possibly even reverse the ordering of the smallest and the larger sums for one or more quadruples. Thus the *ST* algorithm uses a scoring rule designed to be more robust in the face of error: pairs of objects corresponding to the smallest sum are assigned a score of 2, those pairs corresponding to the next smallest sum are given a score of 1, and those for the largest sum are given a score of 0. This 2-1-0 scoring rule was advocated by Sattath and Tversky as improving performance of the algorithm with errorful data.

If only a single pair of objects were combined on each stage, the algorithm would be slow, especially for larger data sets. The reason is that with n objects, there are Q quadruples of objects, where $Q = n(n\text{-}1)(n\text{-}2)(n\text{-}3)/24$. Combining only a single pair of objects at each stage results in $(n\text{-}3)$ stages before the topology of the tree is determined. Thus this slow version of the algorithm (henceforth denoted algorithm ST_1) requires processing time roughly proportional to n^5 [actually to $n(n\text{-}1)(n\text{-}2)(n\text{-}3)(n\text{-}3)$], and quickly becomes impractical in most implementations as n grows.

Accordingly, Sattath & Tversky (1977) used the following rule to select up to $n_i/2$ pairs of objects to be combined on stage i. For each object x, the other object y corresponding to the largest entry in that row and column of the neighbor score matrix **N** is defined as x's nearest neighbor. All object pairs for which this nearest neighbor relationship is reciprocal are termed "mutual nearest neighbors", and that pair is marked to be combined on this stage. Using this rule results in best-case processing time approximately proportional to $(n^4)\log_2(n)$, as discussed below.

Corter (1982) introduced a modification (algorithm *STC*) of the basic *ST* algorithm, motivated by the observation that sometimes the nearest neighbor relationship is not reciprocal. For example, x may be y's nearest neighbor, $\mathbf{N}(x,y) = MAX[\mathbf{N}(y,*)]$, but u is x's nearest neighbor: $\mathbf{N}(x,u) = MAX[\mathbf{N}(x,*)]$.

Or, the neighbor scores of y with x and u with x may be maximal but equal: $\mathbf{N}(x,y) = \mathbf{N}(x,u)$. In these cases, it is not clear whether y or u should be combined with x at this stage (simply refusing to combine either pair does not result in effective performance).

The modification introduced in Corter's (1982) algorithm STC is as follows. In the specific cases described above, the problem is to choose between y and u as the object to be combined with x. This can be accomplished in an optimal way simply by checking the sums of the tree inequality for the "generalized quadruple" x,y,u,V, where V is a composite object consisting of the set of all n_i - 3 objects $\neq x,y,u$. That is, the following sums are computed: $S_1 = d(x,y) + d(u,V)$, $S_2 = d(x,u) + d(y,V)$, and $S_3 = d(x,V) + d(y,u)$, where $d(x,V)$ is defined as the average distance of all objects v in V to x:

$$(1) \qquad d(x,V) = \frac{1}{n_i - 3} \sum_{v \in V} d(x, v).$$

If y is nearer to x in the true tree structure than is u, then S_1 should be less than S_2, which should be equal to S_3.

The use of this criterion for disambiguating ties and non-reciprocity in the **N** matrix scores results in improved performance for the *STC* algorithm over the original *ST* version. Specifically, in about 10% of randomly generated data sets with error, algorithm STC gave slightly improved fits with only a small increase in processing time (Corter, 1992, 1996). The improvement in fit was generally on the order of 1% in proportion of variance accounted for (PVAF), defined as the squared correlation of the model distances with the generated data dissimilarities. Algorithm *STC*, implemented as a PASCAL program called "ADDTREE/P", has been disseminated widely (Corter, 1996) and used successfully in a variety of applications.

One drawback to the *STC* and *ST* algorithms is that the computation time they require increases rapidly with n. As already mentioned, the search through all quadruples of objects at each stage of the algorithm requires processing time proportional to $n(n\text{-}1)(n\text{-}2)(n\text{-}3)/24$, the number of quadruples at a given stage. If the program is successful in finding $n/2$ or $(n\text{-}1)/2$ pairs to be combined at each stage (the best case), this would result in performance approximately proportional to $n(n\text{-}1)(n\text{-}2)(n\text{-}3)\log_2(n)$. Actually best-case performance would be somewhat better than this, because n decreases at each stage (by a factor of approximately 1/2 in this best case), thus the number of quadruples may be reduced at each stage by a factor of roughly 16 to 1. Worst case performance of the algorithm results when only a single pair of mutual nearest neighbor objects are found at each stage,

perhaps because of error in the data. In this worst case, performance will be proportional to $n(n\text{-}1)(n\text{-}2)(n\text{-}3)(n\text{-}3)$, because n-3 stages will be required.

In summary, because processing time requirements for the *STC* algorithm increase rapidly as n grows, it is not practical for very large data sets. The "generalized triples" or *GT* algorithm, described in the next section, offers similar performance in terms of fit, with a theoretical order(n) improvement in processing time that makes it more practical for large data sets.

The Generalized Triples (GT) Algorithm

Finding Tree Neighbors Using "Metric" Information

It is natural to ask whether the least-squares criterion used in algorithm *STC* to decide on the true tree structure for certain individual quadruples of objects might be used as the basis for a new algorithm. That is, rather than using the 2-1-0 scoring rule of the *ST* and *STC* algorithms to compile a neighbor-count matrix **N**, and using the least-squares criterion of algorithm *STC* only to resolve ambiguities in the neighbor score relationships, it might be possible to use the metric information in the sums S_1, S_2 and S_3 computed across all quadruples of objects to initially pick the correct neighbor for each object x. "Metric" information is used here to refer to the actual value of the differences among the sums S_1,S_2, and S_3, as opposed to the merely ordinal relationships among them that determine the *ST* algorithm's 2-1-0 scoring rule. Thus, the resulting algorithm is "metric" in the sense that it uses numerical information about the sums (implying that the proximities must be measured at the interval or ratio level), rather than merely their ordinal relationships.

Consider the additive tree defined on four objects shown in Figure 1a. Each arc of the tree in Figure 1a is marked with a greek letter denoting the arc-length parameter. For dissimilarities on four objects perfectly satisfying the additive tree inequality, these parameters can be estimated without error as follows, if the true tree topology is known to be as shown in Figure 1a.

$$\hat{\alpha} = 1/2(d(x,y) + 1/2\{[d(x,u) + d(x,v)] - [d(y,u) + d(y,v)]\})$$

$$\hat{\beta} = 1/2(d(x,y) + 1/2\{[d(y,u) + d(y,v)] - [d(x,u) + d(x,v)]\})$$

$$\hat{\gamma} = 1/2(d(u,v) + 1/2\{[d(x,u) + d(y,u)] - [d(x,v) + d(y,v)]\})$$

$$\hat{\delta} = 1/2(d(u,v) + 1/2\{[d(x,v) + d(y,v)] - [d(x,u) + d(y,u)]\})$$

$$\hat{\epsilon}_1 = 1/2\{[d(x,u) + d(y,v)] - [d(x,y) + d(u,v)]\} = 1/2(S_2 - S_1)$$

$$-\hat{\epsilon}_3 = -1/2\{[d(x,y) + d(u,v)] - [d(x,v) + d(y,u)]\} = -1/2(S_1 - S_3)$$

Note that two equivalent estimates of ϵ, $\hat{\epsilon}_1$ and $-\hat{\epsilon}_3$, are available for errorless data, because in an additive tree the sums $d(x,u)+d(y,v)$ and $d(x,v)+d(y,u)$ must be equal ($\hat{\epsilon}_3$ is used as a negative quantity for consistency with developments below) . For errorful data, a more efficient estimate of the length of the central tree arc is available:

$$\hat{\epsilon}^* = 1/2\{1/2[d(x,u) + d(y,v) + d(x,v) + d(y,u)] - [d(x,y) + d(u,v)]\}$$

$$= 1/2[1/2(S_2 + S_3) - S_1]$$

It is evident that this estimate is simply the average of $\hat{\epsilon}_1$ and $-\hat{\epsilon}_3$. Note also that the remaining such quantity that can be computed among the three sums, $\hat{\epsilon}_2 = 1/2(S_3 - S_2)$, is equal to 0 for errorless data.

Now consider the problem of trying to identify the true tree topology using the six interobject distances. When the true tree structure is unknown, the tree topology of any quadruple of objects x,y,u,v corresponds to one of the three cases shown in Figure 2. In Case 1, y is x's tree neighbor, and u is neighbors with v. In Case 2, u is x's neighbor, and y is neighbors with v. Finally, in Case 3, neither y nor u is x's neighbor, rather v is. It is not difficult to see that the problem of determining the true tree topology is equivalent to the problem of estimating ϵ appropriately. Because the true tree structure is not known, we are not sure if Case 1, 2, or 3 applies: thus we cannot be sure which of the following expressions provides an appropriate estimate of the central arc length parameter ϵ:

$$\hat{\epsilon}_1 = 1/2\{[d(x,u) + d(y,v)] - [d(x,y) + d(u,v)]\} = 1/2(S_2 - S_1)$$

$$\hat{\epsilon}_2 = 1/2\{[d(x,v) + d(y,u)] - [d(x,u) + d(y,v)]\} = 1/2(S_3 - S_2)$$

$$\hat{\epsilon}_3 = 1/2\{[d(x,y) + d(u,v)] - [d(x,v) + d(y,u)]\} = 1/2(S_1 - S_3).$$

For errorless data, perfect estimates of are provided by both $\hat{\epsilon}_1$ and $-\hat{\epsilon}_3$ in Case 1; by both $\hat{\epsilon}_2$ and $-\hat{\epsilon}_1$ in Case 2; and by $\hat{\epsilon}_3$ and $-\hat{\epsilon}_2$ in Case 3. Therefore for errorful data the most efficient estimators of the central arc length are given by:

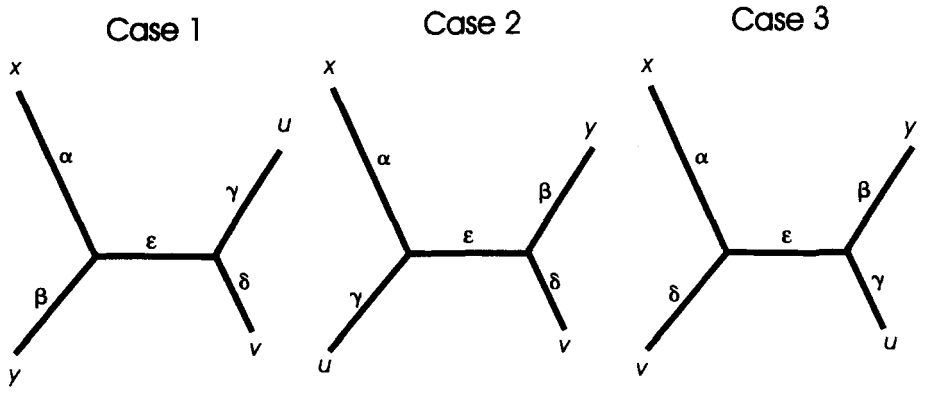

Figure 2
The three possible tree topologies for distinguished objects x,y,u,v.

$$\text{Case 1: } \hat{\epsilon}_1^* = 1/2(\hat{\epsilon}_1 - \hat{\epsilon}_3)$$

$$\text{Case 2: } \hat{\epsilon}_2^* = 1/2(\hat{\epsilon}_2 - \hat{\epsilon}_1)$$

$$\text{Case 3: } \hat{\epsilon}_3^* = 1/2(\hat{\epsilon}_3 - \hat{\epsilon}_2).$$

It is easy to verify that for errorless data these expressions both give the correct estimates for the central arc length and indicate the true tree structure: if the true tree structure is $(x\ y)(u\ v)$ (Case 1), then $\hat{\epsilon}_1 = \hat{\epsilon}_3$ will both be positive (and equal), while $\hat{\epsilon}_2 = 0$. If the tree structure is $(x\ u)(y\ v)$ (Case 2), then $\hat{\epsilon}_2 = -\hat{\epsilon}_1$ will be positive and equal, with $\hat{\epsilon}_3 = 0$, and if the true tree is $(x\ v)(y,u)$ (Case 3), then $\hat{\epsilon}_3 = -\hat{\epsilon}_2$ will be positive and equal, with $\hat{\epsilon}_1 = 0$.

When $n \geq 4$, the above results hold for every quadruple of objects x,y,u,v. In the general case, some of these quadruples will have structure corresponding to Case 1, some to Case 2, and some to Case 3. However, in order to generalize the above observations and at the same time lay the groundwork for the more efficient algorithm proposed here, let us recast the question somewhat and ask, for each *triple* of objects x,y,u: "Is y or u x's nearest neighbor in the tree?" In fact, the answer to this question depends on the placement of any fourth object v in the tree, as depicted schematically in Figure 3. For a given focal object x, and candidate neighbors y and u, let us

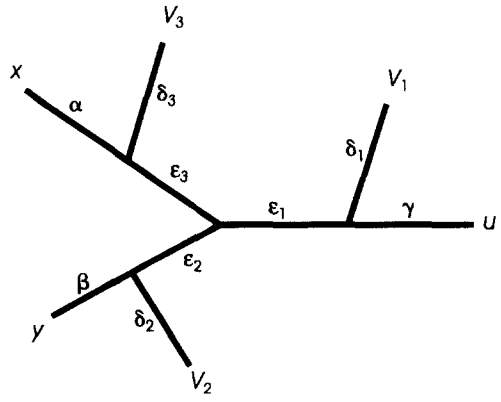

Figure 3

Schematic representation of an additive tree, showing the topologies of quadruples induced for distinguished objects x,y,u by differing placements of a fourth object v.

denote the set of objects v inducing Case 1 as V_1 (with cardinality n_1), the set of objects inducing Case 2 as V_2 (with cardinality n_2), and the set inducing Case 3 as V_3 (with cardinality n_3). The tree of Figure 3 is described as merely schematic because the leaves corresponding to individual objects in set V_3 might join the tree at different points along the arc consisting of the union of segments labelled α and ϵ_3.

When y is indeed x's nearest neighbor in the true tree structure, it can be seen that Case 1 will obtain for all n_1 quadruples x,y,u,v, and $\hat{\epsilon}_1$, $-\hat{\epsilon}_3$, and hence $\hat{\epsilon}^*_1$ will be positive for every quadruple. Furthermore, for any y that is not x's nearest neighbor, at least one quadruple will induce Case 2 or Case 3, and either $\hat{\epsilon}_1 = -\hat{\epsilon}$ and $-\hat{\epsilon}_3 = 0$, or $\hat{\epsilon}_1 = 0$ and $-\hat{\epsilon}_3 = -\hat{\epsilon}$, for any such quadruple. These observations suggest a way of identifying the nearest neighbor of object x: for all $y \neq x$, define:

$$T(x,y) = \sum_{u,v \neq x,y} \hat{\epsilon}^*_1.$$

Then that object y for which $T(x,y)$ is maximal can be identified as the nearest neighbor of x.

Thus one strategy for finding the best tree structure for errorful data would be to compute the estimates $\hat{\epsilon}^*_1$, $\hat{\epsilon}^*_2$, $\hat{\epsilon}^*_3$ for all quadruples of objects, summing $\hat{\epsilon}^*_1$ across all quadruples involving x and y to create a summed T score for the pair (x,y), then joining that pair of objects that has the highest such summed score. But this strategy would not lead to an algorithm more

efficient than algorithm *STC*, because it still requires looking through all quadruples of objects at each stage.

However, a shortcut strategy is available for computing these summed estimates when $n \geq 4$ (Corter, 1992). This shortcut requires precomputation of the summed distances of each object x to all other objects:

$$R(x) = \sum_{y \neq x} d(x,y) ,$$

The goal is to compute $T(x,y)$, summed across all $u,v \neq x,y$. Thus,

$$T(x,y) = \sum_{u,v \neq x,y} \hat{\epsilon}_1^* = \sum_{u,v \neq x,y} \tfrac{1}{2}(\hat{\epsilon}_1 - \hat{\epsilon}_3)$$

$$= \tfrac{1}{2}\left(\sum_{u,v \neq x,y} \hat{\epsilon}_1 - \sum_{u,v \neq x,y} \hat{\epsilon}_3 \right) = \tfrac{1}{2}(T_1 - T_3) .$$

Taking each term T_1 and T_3 separately, we have

$$T_1 = \sum_{u,v \neq x,y} \hat{\epsilon}_1 = \sum_{u,v \neq x,y} \tfrac{1}{2}(S_2 - S_1)$$

$$= \tfrac{1}{2} \sum_{u,v \neq x,y} [(d(x,u) + d(y,v) - (d(x,y) + d(u,v)]$$

$$= \tfrac{1}{2} \sum_{u \neq x,y} [(n-3)d(x,u) + \sum_{v \neq x,y,u} d(y,v) - (n-3)d(x,y) - \sum_{v \neq x,y,u} d(u,v)]$$

However, because

$$R(y) = \sum_{v \neq y} d(y,v) = [\sum_{v \neq x,y,u} d(y,v)] + d(x,y) + d(y,u)$$

$$R(u) = \sum_{v \neq y} d(u,v) = [\sum_{v \neq x,y,u} d(u,v)] + d(x,u) + d(y,u),$$

it follows that

$$T_1 = \tfrac{1}{2} \sum_{u \neq x,y} \{(n-3)d(x,u) + [R(y) - d(x,y) - d(y,u)]\} - (n-3)\, d(x,y)$$

$$- [R(u) - d(x,u) - d(y,u)]\}$$

$$= \tfrac{1}{2} \sum_{u \neq x,y} [(n-2)d(x,u) + R(y) - (n-2)d(x,y) - R(u)]$$

$$= \tfrac{1}{2} \sum_{u \neq x,y} \{R(y) - R(u) + (n-2)[d(x,u) - d(x,y)]\}$$

Similar steps give

$$T_3 = \tfrac{1}{2} \sum_{u \neq x,y} \{R(y) - R(x) + (n-2)[d(x,y) - d(y,u)]\}$$

The $T(x,y)$ scores, computed as $T(x,y) = 1/2(T_1 - T_3)$, can be compiled into a matrix **T**, where a higher $T(x,y)$ score indicates that x and y are relatively close in the tree structure. In fact, it can be shown that the maximal $T(x,y)$ score in the matrix indicates a pair of objects in the tree that are "tree neighbors" (i.e., x is topologically closest to y in the tree graph, and y is closest to x), and should be joined at this stage of the algorithm.

Of course, combining only a single pair of objects per stage results in relatively slow operation of the algorithm. The basic estimation step described above requires looking through all triples of objects x,y,u, summing the appropriately signed estimates $\hat{\epsilon}^*_1$, $\hat{\epsilon}^*_2$, and $\hat{\epsilon}^*_3$ into the entries $T(x,y)$, $T(x,u)$, and $T(y,u)$. Combining only a single pair of objects (corresponding to the maximal **T** score) per stage would require $(n\text{-}3)$ stages, and therefore result in an algorithm with computation time proportional to $(n)(n\text{-}1)(n\text{-}2)(n\text{-}3)$. This slow variant of the present method may be referred to as algorithm GT_1 (for "Generalized Triples — combining only 1 pair").

In general, however, more than one pair will be neighbors in the tree structure at any given stage of the agglomerative algorithm. Thus we need some way of identifying these additional neighbor pairs. A straightforward approach begins by defining a nearest neighbor relation using the scores in the **T** matrix. Y is defined as x's nearest neighbor if $T(x,y) = MAX\ [T(x,z)]$ for all $z \neq x$. If this nearest neighbor relationship is reciprocal, then x and y are said to be mutual nearest neighbors (*MNN*). One approach to identifying additional pairs at each stage to be combined would be to combine all pairs of objects that are *MNN* (this is the approach taken by Sattath & Tversky,

1977). However, this turns out not to work well, because occasionally pairs of objects are *MNN* merely as a result of error in the dissimilarities. A heuristic that works well in preventing the use of such "gratuitous" *MNN* pairs involves finding the maximal **T** score in the matrix that is dominated by another **T** score in that row or column. This quantity, T_{inf}, is used as a cutoff: all pairs of objects x,y that are *MNN* are marked to be combined at this stage, as long as $T(x,y)$ is greater than T_{inf}. This principled heuristic works well, and allows more than one pair of *MNN* objects to be combined at each stage of the algorithm. The resulting algorithm, termed the "generalized triples" or *GT* algorithm, is outlined in Table 1 (next page).

The *GT* algorithm can be expected to work in best-case time proportional to $n(n\text{-}1)(n\text{-}2)\log_2(n)$, because at each stage a search is made through all $n(n\text{-}1)(n\text{-}2)/6$ triples of objects, and if $n/2$ or $(n\text{-}1)/2$ pairs of objects are combined at each stage then approximately $\log_2(n)$ stages will be needed. In the worst case only a single pair of mutual nearest neighbors will be found and combined at each stage, resulting in $(n\text{-}3)$ stages, and performance will be proportional to $n(n\text{-}1)(n\text{-}2)(n\text{-}3)$.

A program, named "GTREE", has been written in Turbo PASCAL to implement the generalized triples (*GT*) algorithm. The next section presents results of simulation studies comparing the performance of this program with that of some previous methods.

Simulation Studies

Two simulation studies were conducted to evaluate the effectiveness of the proposed generalized triples or *GT* algorithm. The first study compared the performance of the *GT* algorithm with the Sattath-Tversky-Corter (*STC*) algorithm. The second compared the performance of *GT* with De Soete's (1983) mathematical programming approach to fitting additive trees (algorithm LSADT).

For the first study, both the *GT* and the *STC* algorithms were implemented in PASCAL and compiled using Turbo PASCAL for DOS, version 6.0. For comparison, "slow" variants of the two algorithms ("ST_1" and "GT_1") that only combine a single object pair at each stage were also programmed and evaluated. All analyses reported in the simulations were conducted on a Pentium-based personal computer running at 90 MegaHerz, with 16 Megabytes of RAM.

Random binary trees were constructed, with arc lengths chosen randomly from a uniform distribution on (0,1), using n's of either 20, 40, 60, or 80 objects. The use of random binary trees (generated separately for each proximity matrix) ensures that the results of the simulation can be generalized

Table 1
Summary of Algorithm "*GT*"

GIVEN: A matrix **D** of distances or dissimilarities among n "objects" denoted $x,y,u...$: $\mathbf{D} = [d(x,y)]$.

Step 0. Adjust dissimilarities to satisfy the metric axioms: symmetry, positivity, and the triangle inequality (see text).

STAGE i: ITERATE STEPS 1-4 UNTIL $N_i \leq 3$:

Step 1. For each object x, compute $R(x) = d(x,y)$, the sum (over y) of dissimilarities of x to all objects $y(x)$. For every pair of objects (x,y), initialize a "tree score" variable $T(x,y)$ to 0.

Step 2. For every triple of objects (x,y,u), compute:

$$T_1 = \{R(y) - R(u) + (n_i-2)[(d(x,u)-d(x,y)]\}$$
$$T_2 = \{R(x) - R(y) + (n_i-2)[(d(y,u)-d(x,u)]\}$$
$$T_3 = \{R(u) - R(x) + (n_i-2)[(d(x,y)-d(y,u)]\}$$

Accumulate these estimates in the matrix of T scores:

$$T(x,y) = T(x,y) + T_1 - T_3$$
$$T(x,u) = T(x,u) + T_2 - T_1$$
$$T(y,u) = T(y,u) + T_3 - T_2$$

Step 3. For each x, define y as x's nearest neighbor if

$$T(x,y) = MAX\,[T(x,z)] \text{ for all } z \neq x.$$

If the nearest neighbor relationship is reciprocal, that is,

$$T(x,y) = MAX\,[T(x,z)] \text{ for all } z \neq x = MAX\,[T(y,z)] \text{ for all } z \neq y,$$

then define pair (x,y) as mutual nearest neighbors (*MNN*).

Define $T_{inf} = \{MAX\ [T(x,y)] \mid \exists v$ such that $T(x,y) < T(x,v)$ or $T(x,y) < T(v,y)\}$.

Step 4. For each of the k *MNN* pairs (x,y) for which $T(x,y) > T_{inf}$, combine the pair (x,y) into a cluster (xy). Redefine distances from each new "object" (xy): $d[(xy),z] = AVE[d(x,z), d(y,z)]$ for all $z \neq x,y$, where the average is weighted by the number of leaves in branches x and y. Estimate lengths of the two arcs below each new node (xy). Reduce n for the next stage: $n_{i+1} = n_i - k$.

to a variety of tree structures, though the results might differ for other populations of tree-generated data sets. Both errorless and errorful data were used. In the errorless case, no error was added to the constructed tree distances. In the low-error condition, random normal error with variance equal to 1/4 the variance of the errorless distances was added. In the high-error condition, this error had variance equal to 1/2 the variance of the errorless distances. One-hundred data sets were generated in each condition.

Results of the simulation are shown in Table 2. The basic criteria for evaluation include the proportion of variance explained in the data by the model distances (*RSQ*), and the computation time in seconds (*SEC*) used by the program. Most noteworthy is the fact that all four algorithms perform indistinguishably well in terms of fit of the solution. In the case of errorless data, all four algorithms give mean *RSQ* values of 1.000. For the low error condition, *RSQ* declines from approximately .84 for the data sets with n=20 to .81 for the n=80 data sets. For high error, the fits decline from about .74 to about .68. Note that there is no evidence of suboptimal fits anywhere, because for the low-error conditions the true proportion of model variance in the data should be .80, while for the high-error conditions it should be .67. The finding that observed fits generally exceed these values shows that some overfitting is occurring, especially for the lower n conditions.

Regarding the computation time required for solution, the first thing to note is that when n=20 the algorithms are not distinguishably different in speed. When n=40 or higher, it can be seen that for errorless data, algorithms *GT* and *STC* are roughly comparable in performance, with perhaps a slight advantage for *STC* with fewer objects and a slight advantage for *GT* with n=80. For errorful data, however, *GT* shows a consistent advantage (beginning with n=40) that increases with the amount of added error and with n. Algorithms GT_1 and ST_1 are markedly inferior to the others, as expected, since they exemplify worst-case performance (because these algorithms combine only a single pair at each stage). The advantage of GT_1 over ST_1 increases with n, as expected. Note that amount of error does not affect the performance of GT_1 and ST_1. This is not surprising, because the main effect of error on computation time is to reduce the number of mutual nearest-neighbor pairs found at each stage, thus increasing the number of stages required for algorithms *STC* and *GT*, but not affecting ST_1 and GT_1.

In order to check if the processing time advantage of *GT* over *STC* gibes with the theoretical predictions made above, further simulated data sets were produced and analyzed. Additional cells were created, corresponding to n=10, 30, 50, or 70 objects and either no added error or error variance equal to 50% of the generated tree distances. Thus for the no-error and high-error conditions of Table 2 (next page), data sets of size n=10 to n=80 were

Table 2

Simulation Results for Four Algorithms, Showing Mean *R*-squared of Final Solution and Mean Computation Time per Solution (in Seconds). Results Based on 100 Runs for Each Cell

		ALGORITHM "*STC*"			ALGORITHM "*GT*"		
	σ^2_e =	.00	.25	.50	.00	.25	.50
n= 20	*RSQ*:	1.000	0.839	0.741	1.000	0.840	0.738
	SEC:	0.14	0.12	0.14	0.11	0.17	0.15
n= 40	*RSQ*:	1.000	0.819	0.702	1.000	0.819	0.701
	SEC:	1.06	1.60	1.78	1.22	1.45	1.49
n= 60	*RSQ*:	1.000	0.813	0.690	1.000	0.812	0.689
	SEC:	4.50	8.60	10.61	4.67	5.80	5.97
n= 80	*RSQ*:	1.000	0.810	0.682	1.000	0.810	0.681
	SEC:	14.38	34.31	43.27	11.88	15.60	15.40
		ALGORITHM "ST_1"			ALGORITHM "GT_1"		
	σ^2_e =	.00	.25	.50	.00	.25	.50
n= 20	*RSQ*:	1.000	0.840	0.736	1.000	0.842	0.737
	SEC:	0.20	0.22	0.29	0.24	0.22	0.30
n= 40	*RSQ*:	1.000	0.821	0.700	1.000	0.820	0.697
	SEC:	4.34	4.29	4.12	2.87	2.97	3.02
n= 60	*RSQ*:	1.000	0.814	0.690	1.000	0.815	0.690
	SEC:	31.70	30.38	30.67	13.48	13.87	14.00
n= 80	*RSQ*:	1.000	0.810	0.681	1.000	0.811	0.682
	SEC:	136.92	134.42	133.63	40.93	42.59	43.20

available. A total of 100 data sets for each cell were generated, analyzed by either *GT* or *STC*, and the mean computation time (in seconds) was recorded. For each algorithm (*GT* or *STC*) and each error condition (no error or high error), the mean computation times (*T*) for different levels of *n* (=10 to 80) were related to various polynomial functions of *n*, using a linear model with no intercept parameter (because computation time can be expected to be equal to 0 for *n*=0 objects). Thus the models used were of the form: $T = ß\ F(n)$. The specific functions investigated were $F_3(n) = n(n\text{-}1)(n\text{-}2)$, $F_{3L}(n) = n(n\text{-}1)(n\text{-}2)\log(n)$, $F_4(n) = n(n\text{-}1)(n\text{-}2)(n\text{-}3)$, $F_{4L}(n) = n(n\text{-}1)(n\text{-}2)(n\text{-}3)\log(n)$, and $F_5(n) = n(n\text{-}1)(n\text{-}2)(n\text{-}3)(n\text{-}3)$. Note that according to the

analyses presented in previous sections, best-case performance of the *GT* algorithm ought to be proportional to $F_{3L}(n)$, while worst-case performance ought to be proportional to $F_4(n)$. For the *STC* algorithm, best-case performance ought to be proportional to $F_{4L}(n)$ and worst case to $F_5(n)$.

These five polynomial models were fit to the array of computation times for n=10 to n=80, for the following four conditions: algorithm *GT* with no error, algorithm *GT* with high error, algorithm *STC* with no error, algorithm *STC* with high error. Fits were obtained by ordinary least-squares, specifically by using a regression package to request a simple regression equation with no intercept term. For each condition, the best-fitting polynomial function was defined as that function resulting in the lowest *SSE*. Results were as follows. For algorithm *GT*, the best-fitting polynomial function was $F_{3L}(n) = n(n\text{-}1)(n\text{-}2)\log(n)$, for both the errorless and the high-error data sets (*SSE*=.014 and =.041, respectively). This means that average-case performance (for this population of simulated data sets) approximates the predicted best-case performance, even with noisy data. For algorithm *STC*, the best-fitting function for errorless data was $F_4(n) = n(n\text{-}1)(n\text{-}2)(n\text{-}3)$, with *SSE*=.287). This is somewhat faster than the expected best-case performance proportional to $F_{4L}(n)$, which had a slightly larger error (*SSE*=.533). For errorful data, however, the best-fitting function was $F_5(n) = n(n\text{-}1)(n\text{-}2)(n\text{-}3)(n\text{-}3)$, in line with worst-case theoretical predictions. Note that error was relatively large for this model at *SSE*=1.70, indicating that none of the functions described *STC* high-error performance very well. To summarize, the approximately order(n) advantage in computation time expected for algorithm *GT* is confirmed for errorful data, but the advantage seems to be somewhat less than expected for errorless data. It may be that with errorless data, algorithm *STC*'s method of choosing mutual nearest-neighbor pairs is relatively more effective (meaning that it combines closer to $n/2$ pairs at each stage). When close to $n/2$ pairs are combined at the first stage, the number of quadruples to be checked at the second stage is reduced to approximately 1/16 of the number of quadruples at the first stage. Thus for errorless data it may be that the computational requirements of the first stage, with number of quadruples proportional to $n(n\text{-}1)(n\text{-}2)(n\text{-}3)$, dominate the requirements of subsequent stages.

A graphical summary of these results is shown in Figure 4 (next page), which plots the mean computation time for data sets of size n=10 to 80 versus the best-fitting polynomial function for the *GT* algorithm, namely $F_{3L}(n) = n(n\text{-}1)(n\text{-}2)\log(n)$. Each separate line plots the results for a given algorithm and level of error. The main observation is that the lines for algorithm *GT* are remarkably linear with $F_{3L}(n)$, both for errorless and high-error data. In contrast, the lines for algorithm *STC* are positively accelerated

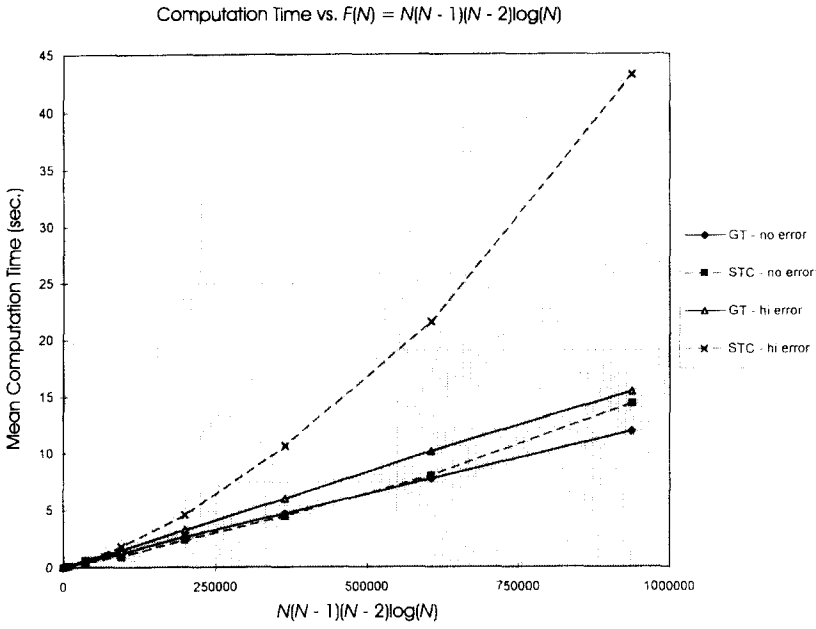

Figure 4

Simulation study 1: plot of computation time (in seconds) for data sets of size n=10 to n=80, with the x-axis plotted in coordinates proportional to $n(n\text{-}1)(n\text{-}2)\log(n)$. The solid lines show the performance of the *GT* algorithm with errorless and high-error data sets; dotted lines show the performance of the *STC* algorithm.

functions of this polynomial, meaning that computation time for this algorithm rises faster than this function of n. It can be seen from the plot that with errorless data algorithm *STC* is actually slightly faster than *GT* for data sets smaller than about n=65, at which point *GT* becomes more efficient. For high-error data, however, algorithm *GT* is consistently faster than *STC*, and this advantage increases rapidly with n.

A second simulation study was conducted to compare performance of the *GT* algorithm with that of the LSADT program, that implements the mathematical programming algorithm described by De Soete (1983). This simulation study also varied the level of error added to the generated data and the number of objects. The level of error used was again either 0, 1/4, or 1/2 of the variance of the errorless data. The number of objects in each generated data set was either 12, 24, or 36, corresponding to the levels used in the simulations reported by De Soete (1983) and Pruzansky, Tversky, & Carroll (1982). Fifty data sets were generated and fit for each combination of study parameters.

Results of this simulation study are reported in Table 3. De Soete's FORTRAN-based LSADT program offers several options for the starting configuration used by the iterative algorithm. In Table 3, the runs labeled "LSADT(data)" used the unperturbed data as the initial configuration, while those labeled "LSADT(rand)" used uniform random numbers as the starting configuration. The results reported are the mean *R*-squared of the final model (tree) distances with the data, and the approximate computation time in seconds for each single run. The table shows that differences in fit between the three algorithms are minimal. In the errorless case the mean *R*-squared is equal to 1.000 for all three algorithms in all conditions (i.e. fit is perfect). For data with added error, the mean *R*-squared values are very close, and apparently within the range of sampling error. More specifically, LSADT(data) shows higher fits than *GT* in three out of six error conditions, while LSADT(rand) outperforms *GT* in four out of six cells. The advantage for LSADT(rand) in the six error cells averages 0.2%, however, so this may be a real though small effect.

However, in terms of computation time the algorithms differ markedly, especially for the higher n's. The *GT* algorithm's advantage in speed ranges from roughly one order of magnitude (when n=12) to roughly two orders of magnitude (when n=36). For example, with data on n=36 objects and high error, *GT* takes approximately one second, while both versions of LSADT require more than 2.5 minutes. Note that the *GT* algorithm's advantage in speed grows as n increases, so with larger data sets the *GT* algorithm may be the only practical approach.

Table 3
Simulation Results for Algorithms *GT* and Two Versions of *LSADT* (De Soete, 1983), Showing Mean *R*-squared of Final Solution and Approximate Computation Time per Solution (in Seconds).

		GT			*LSADT*(data)			*LSADT*(rand)		
	σ^2_e =	.00	.25	.50	.00	.25	.50	.00	.25	.50
n=12	*RSQ*:	1.000	.862	.784	1.000	.865	.781	1.000	.862	.778
	SEC:	0.0	0.0	0.0	0.2	0.7	1.0	0.4	0.7	0.9
n=24	*RSQ*:	1.000	.834	.724	1.000	.834	.727	1.000	.836	.733
	SEC:	0.2	0.3	0.3	3.6	18.3	21.5	9.1	19.4	22.2
n=36	*RSQ*:	1.000	.822	.705	1.000	.821	.710	1.000	.824	.709
	SEC:	0.9	1.1	1.1	26.3	115.1	155.9	62.8	136.2	162.3

Summary

A new combinatorial algorithm for fitting additive trees has been described, the "generalized triples" or *GT* algorithm. The method involves computations performed on all triples of objects in the set to be modeled, rather than requiring search through all quadruples of objects as do previous combinatorial algorithms such as the Sattath & Tversky (1977) *ST* algorithm, and the modified version introduced by Corter (1982) (algorithm *STC*). The computation time required by the *GT* algorithm is approximately $(n^3)\log(n)$ for both errorless and errorful data, making it substantially faster than previous algorithms for moderate- to large-sized data sets. Simulation studies also indicated that the *GT* algorithm performs effectively in terms of fit of the obtained solution. Thus the algorithm may prove useful for the fitting of additive trees to proximity data, especially with large data sets.

References

Abdi, H. (1990). Additive-tree representations. *Lecture Notes in Biomathematics, 84*, 41-59.

Abdi, H., Barthélemy, J.-P., & Luong, X. (1984). Tree representations of associative structures in semantic and episodic memory research. In E. DeGreef & J. van Buggenhaut (Eds.), *Trends in mathematical psychology*. Amsterdam: North-Holland.

Arabie, P. & Hubert, L. J. (1992). Combinatorial data analysis. *Annual Review of Psychology, 43*, 169-203.

Barthélemy, J.-P. & Guénoche, A. (1991). *Trees and proximity representations*. New York: Wiley.

Barthélemy, J.-P. & Luong, X. (1986). Représentations arborée des mesures de dissimilarité. *Statistiques et Analyse de Données, 11*, 20-41.

Beller, M. (1990). Tree vs. geometric representation of tests and items. *Applied Psychological Measurement, 14(1)*, 13-28.

Buneman, P. (1971). The recovery of trees from measures of dissimilarity. In F.R. Hodson, D.G. Kendall, & P. Tautu (Eds.), *Mathematics in the archaeological and historical sciences*. Edinburgh: Edinburgh University Press.

Carroll, J. D. & Pruzansky, S. (1975, July). *Fitting of hierarchical tree structure (HTS) models, mixtures of HTS models, and hybrid models, via mathematical programming and alternating least squares*. Paper presented at U.S.-Japan Seminar in Theory, Methods, and Applications of Multidimensional Scaling and Related Techniques. Uuniversity of California San Diego.

Clinchy, R. M. (1990). *Mental representations of organizational structure: An investigation of the psychological underpinnings of the division of labor*. Unpublished doctoral dissertation, City University of New York, New York.

Corter, J. E. (1982). ADDTREE/P: A PASCAL program for fitting additive trees based on Sattath and Tversky's ADDTREE algorithm. *Behavior Research Methods & Instrumentation, 14*, 353-354.

Corter, J.E. (1992, June). *An order-N^3 algorithm for fitting additive trees*. Paper presented at the annual meeting of the Classification Society of North America, E. Lansing, MI.

Corter, J.E. (1996). *Tree models of similarity and association.* Newbury Park, CA: Sage.

Critchley, F. (1986). Some observations on distance matrices. In J. DeLeeuw, W. Heiser, J. Meulman, and F. Critchley (Eds.), *Multidimensional data analysis.* Leiden, The Netherlands: DSWO Press.

Cunningham, J. P. (1978). Free trees and bidirectional trees as representations of psychological distance. *Journal of Mathematical Psychology, 17*, 165-188.

DeSarbo, W. S., Manrai, A. K., & Manrai, L. A. (1993). Non-spatial tree models for the assessment of competitive market structure: An integrated review of the marketing and psychometric literatures. In J. Eliashberg and G. L. Lilien (Eds.), *Handbook in OR & MS, 5*, 193-257.

De Soete, G. (1983). A least squares algorithm for fitting additive trees to proximity data. *Psychometrika, 48*, 621-626.

De Soete, G. (1984). Additive-tree representations of incomplete dissimilarity data. *Quality and Quantity, 18*, 387-393.

De Soete, G. & Carroll, J. D. (1996). Trees and other network models for representing proximity data. In P. Arabie, L. Hubert, & G. De Soete (Eds.), *Classification and clustering.* River Edge, NJ: World Scientific.

Dobson, A. G. (1974). Unrooted trees for numerical taxonomy. *Journal of Applied Probability, 11*, 32-42.

Fitch, W. M. (1981). A non-sequential methods for constructing trees and hierarchical classifications. *Journal of Molecular Evolution, 18*, 30-37.

Furnas, G. W. (1988). Metric family portraits. *Journal of Classification, 6*, 7-52.

Guénoche, A. (1987). Etude comparative de cinq algorithms d'approximation des dissimilarites par des arbres a distance additives. *Mathematique et Science Humaines, 25*, 21-40.

Hubert, L. & Arabie, P. (1995). Iterative projection strategies for the least-squares fitting of tree structures to proximity data. *British Journal of Mathematical and Statistical Psychology, 48*, 281-317.

Johnson, E. J. & Tversky, A. (1984). Representations of perceptions of risks. *Journal of Experimental Psychology: General, 113*, 55-70.

Patrinos, A. N. & Hakimi, S. L. (1972). The distance matrix of a graph and its tree realization. *Quarterly of Applied Mathematics, 30*, 255-269.

Pruzansky, S., Tversky, A., & Carroll, J. D. (1982). Spatial vs. tree representations of proximity data. *Psychometrika, 47*, 3-24.

Roux, M. (1986). Representation d'une dissimilarite par un arbre aux aretes additives. In D. Diday et al. (Eds.), *Data analysis and informatics IV.* Amsterdam: North Holland.

Sattath, S. & Tversky, A. (1977). Additive similarity trees. *Psychometrika, 42*, 319-345.

Sneath, P. & Sokal, R. (1972). *Numerical taxonomy.* San Francisco: Freeman.

Taylor, H. & Tversky, B. (1992). Descriptions and depictions of environments. *Memory & Cognition, 20*, 483-496.

Accepted November, 1997.