A Class of Parametric Tree-Based Clustering Methods

Fred Glover and Yang Wang

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/intechopen.76406

Abstract

We introduce a class of tree-based clustering methods based on a single parameter W and show how to generate the full collection of cluster sets C(W), without duplication, by varying W according to conditions identified during the algorithm's execution. The number of clusters within C(W) for a given W is determined automatically, using a graph representation in which cluster elements are represented by nodes and their pairwise connections are represented by edges. We identify features of the clusters produced which lead to special procedures to accelerate the computation. Finally, we introduce a related node-based variant of the algorithm based on a parameter Y which can be used to generate clusters with complementary features, and a method that combines both variants based on a parameter Z and a weight that determines the contribution of each variant.

Keywords: clustering, minimum spanning trees, spanning forests, machine learning, big data analytics

1. Introduction

Clustering methods have long been a mainstay of statistics and machine learning [1–3], and have experienced a surge in importance with the advent of Big Data Analytics [4, 5]. A highly successful use of clustering in practical applications has been to seek out particular kinds of clustering methods that are effective in particular settings, based on the finding that different classes of problems respond best to specific classes of clustering methods. This finding motivates the work of this paper, which introduces a new class of tree-based clustering methods with an ability to modify the kinds of clusters produced by changing the value of a particular parameter. Moreover, we show all members of class can be generated without duplication by a process that adaptively determines each new parameter value from the information produced by executing the class member that precedes it.

We are motivated to use a tree-based algorithm due to their applications in genome analysis [6–8], image segmentation [9, 10], statistics [11] and microaggregation [12]. The most common forms of the tree-based clustering methods in the literature [8, 13–15] begin with a minimum spanning tree and then successively delete edges according to various criteria. However, our approach has a greater level of flexibility than these commonly applied methods due to the fact that the clusters produced include those that cannot be obtained by removing edges of a minimum spanning tree.

We introduce special techniques for accelerating the execution of our basic approach by exploiting its underlying properties and then introduce a closely related clustering algorithm that replaces an "edge-based" focus with a complementary "node-based" focus. We unify these two classes of approaches by identifying a third class that marries their complementary features, and which provides additional variation by means of a weight that permits the contribution of these complementary approaches to be varied along a continuum. We conclude by demonstrating how the procedures for accelerating the first method can be expressed in a more general form to accelerate the execution of the combined procedure as well.

The ability to generate a family of clustering methods from each of the three basic clustering designs by varying a single parameter (and the weight employed by the third method) invites empirical research to determine parameter ranges that are effective for specific types of clustering applications, opening the possibility to produce clusters exhibiting features different from those customarily obtained.

2. Cluster problem formulation

The clustering problem in our treatment is formulated by reference to a graph $G = (N, E)$ where $N = \{1, \ldots, n\}$ is a set of nodes (cluster elements) and E is a set of edges (pairwise connections between elements) given by $E \subset N \times N = \{(p,q): p,q \in N\}$. The notation (p,q) is understood to represent an unordered pair (hence $(p,q) = (q,p)$, and is equivalently represented by the set notation $\{p,q\}$). Each edge $e = (p,q) \in E$ has an associated cost (or length) denoted by $c(e)$ $(= c(p,q))$. It is not necessary to assume that G is complete or connected. We also do not require that the costs $c(e)$ be nonnegative.

The goal is to partition N into sets (clusters) N_k, $k \in K = \{1, \ldots, ko\}$, where the value ko is automatically determined by the clustering process. We also identify an associated set of edges $E_k \subset \{(p,q), p,q \in N_k\}$, where the subgraph (N_k, E_k) of G constitutes a min cost spanning tree over the nodes of N_k. In contrast to those tree-based clustering approaches that begin with a min cost spanning tree over all of G and selectively delete particular edges, our algorithm produces subgraphs (N_k, E_k), $k \in K$, that may not be possible to obtain by deleting edges from such a tree.

The class of clustering methods we describe is based on specifying the value of a parameter W, whose value uniquely determines the outcome of each clustering method within the class. W is expressed as an additive threshold for selecting edges and hence nodes to be added to a current construction (collection of subgraphs), and observe that W can equally be expressed as a multiplicative threshold in the case where the costs are nonnegative and the two approaches are equivalent in this instance.

We start with any selected value W = Wo≥ 0 and after obtaining a collection of clusters C(W) for a given W we systematically modify W so that over successive iterations all possible cluster collections C(W) for W ≥ Wo will be generated without duplication. The complete range of cluster collections results by choosing Wo = 0 (or Wo = 1 in the multiplicative version).

3. Algorithm to generate the cluster collections C(W)

In overview, we index the edges of E in ascending cost order so that c(e(1)) ≤ c(e(2)) ≤ … ≤ c (e(|E|)), and identify the nodes of edge e(s) by writing e(s) = (p(s), q(s)). We start with each cluster N_k consisting of just the node k, that is, each cluster is a degenerate single node tree given by.

$$N_k = \{k\}, k \in K \text{ for } K = N = \{1, \dots, n\}$$

The associated set E_k of edges in the tree corresponding to N_k is empty ($E_k = \varnothing$). As the algorithm progresses, the composition of the clusters will change and the index set K of clusters will change accordingly.

In addition, we keep a cost value denoted by MinCost(k) for each k ∈ K which identifies the cost of the minimum cost edge e ∈ E_k. To begin, since no cluster yet contains an edge, we define MinCost(k) = Large, a large positive number, for all k ∈ K. (We will not have to examine the set E_k to identify MinCost(k) = Min(c(e): e ∈ E_k) because the structure of the algorithm will insure that MinCost(k) will equal the cost of the first edge added to E_k. In general, while we describe the composition of E_k and the manner in which it changes, the organization of the algorithm assures that it is unnecessary to keep track of E_k since the sets N_k, for k∈ K, will identify the elements in the clusters produced.)

We also maintain a list L(i) for each i∈ N that names the cluster that node i belongs to. Hence, initially, L(i) = (i) since i∈ Ni = {i} for all i∈ N. The redundancy provided by this list enables updates to be performed efficiently. Subsequently, L(i) is modified as node i becomes the member of a new cluster N_k. As this is done, the list K will come to have "holes" in it, i.e., will not consist of consecutive indexes. (At the end of the algorithm we can rename the clusters indexes, if desired, so that K = {1, 2, …, ko} where ko = |K|.)

Finally, during the process of generating the cluster collection C(W) for the current W value, we will identify a value Wnext so that the process may then be repeated for W: = Wnext to generate a new collection of clusters. As previously noted, by starting with W = Wo = 0 (or W = Wo = 1 in the multiplicative version), and then successively identifying Wnext each time a cluster collection C(W) is generated, we can ultimately generate all possible collections C(W), without duplication. The process terminates when W becomes large enough that C(W) consists of a min cost spanning tree over each connected component of G. (A simple condition for identifying this termination point is identified below.)

Building on these observations, we now state the full form of our algorithm.

C(W) Algorithm (Multiplicative Version)

Inputs: The graph G(N, E), cost vector c(e), e ∈ E, initial Wo value for W.

```
Edges are ordered so that the costs satisfy $c(e(1)) \le c(e(2)) \le \dots \le c(e(|E|))$.
Set $W = Wo$ and $sLast = |E|$
**Begin Outer Loop**
While $W < Large$
    *Initialization(A)*. Set $Wnext = Large$, $K = \{1, \dots, n\}$, and for each $k \in K$ let $L(k) = k$,
    $N_k = \{k\}$, $E_k = \emptyset$, and $MinCost(k) = Large$.
    *Initialization(B)*. Let $i' = p(1)$ and $i'' = q(1)$ and select $e(1)$ $(= (i', i''))$, to create the first non-
    degenerate cluster (containing more than one node and hence more than 0 edges) by
    identifying $k' = L(i')$ and $k'' = L(i'')$ and absorbing $N_{k''}$ into $N_{k'}$ to create the cluster $N_{k'} := $
    $N_{k'} \cup N_{k''} = \{i', i''\}$ with edge set $E_{k'} = e(1)$. Set $MinCost(k') = c(e(1))$ and conclude by elimi-
    nating the superfluous cluster $N_{k''}$ (now contained within $N_{k'}$) by setting $K := K \setminus \{k''\}$.
    Finally, initialize the edge index s by setting $s = 1$.
    **Begin Inner Loop**
    While $s < sLast$
        Set $s := s + 1$ and identify edge $e(s) = (p(s), q(s))$. Let $i' = p(s)$, $i'' = q(s)$ and let $k' = L(i')$
        and $k'' = L(i'')$. There are three cases:
        Case (1): If $k' = k''$ ($i'$ and $i''$ belong to the same cluster), then continue to the next
              iteration of the Inner Loop.
        Case (2): If $c(e(s)) > W + MinCost0$, for $MinCost0 = Min(MinCost(k'), MinCost(k''))$,
              then edge $e(s)$ is forbidden to be added to join the clusters $N_{k'}$ and $N_{k''}$ into
              a single cluster. In this case, compute $Wnext = Min(Wnext, c(e(s)) - MinCost0)$
              and continue to the next iteration of the Inner Loop.
        Case (3) (If (1) and (2) do not apply)[1]: Absorb $N_{k''}$ into $N_{k'}$ to create the larger
              cluster $N_{k'} := N_{k'} \cup N_{k''}$ with its associated edge set $E_{k''} := E_{k'} \cup E_{k''} \cup \{e(s)\}$.
              Correspondingly, update $L(i)$ by setting $L(i) = k'$ for all $i \in Nk''$, and set
              $MinCost(k') := Min(MinCost(k'), MinCost(k''), c(e(s))$. Finally, eliminate the
              superfluous cluster $Nk''$ (whose elements are now contained within $N_{k'}$) by
              setting $K := K \setminus \{k''\}$.
    Endwhile.
    // The node and edge sets for the collection of clusters C(W) for the current W are given.
    // by $N_k$ and $E_k$ for $k \in K$. The node sets can alternatively be recovered by reference to.
    // the values $L(i)$, $i = 1, \dots, n$.
    $W = Wnext$
Endwhile
**End of C(W) Algorithm**
```

[1]Case (3) generalizes Initialization(B).

We employ the customary convention that a loop of the form "While x < Constant" will be bypassed if the beginning value of x does not satisfy "x < Constant" and that the execution of the loop will not be interrupted if x is changed so that x ≥ Constant within the loop (though the execution will then terminate at the loop's conclusion). Hence, for example, in the Inner Loop when s: = s + 1 results in s = sLast, the loop will continue its execution until the current iteration ends.

We now make several observations about the algorithm.

Remark 1: The multiplicative version of the C(W) Algorithm results by modifying Case (2) to replace W + MinCost0 by W·MinCost0 and to replace Wnext = Min(Wnext, c(e(s)) – MinCost0) by Wnext = Min(Wnext, c(e(s))/MinCost0). (Hence, addition is replaced by multiplication and subtraction is replaced by division.) These approaches will generate the same collection of clusters under the assumption that all c(e) > 0 for the following reason: a positive value W′ can always be found for the multiplicative case that will cause Wnext to screen out the same set of elements as any positive value W for the additive case, and vice versa. This relationship can also be extended to cover the situation where all c(e)are nonnegative.

Remark 2: The assignment W = Wnext at the end of the outer loop can be replaced by setting W:= Wnext + Δ for a chosen increment Δ to generate only a subset of the possible C(W) collections. Experimentation with a given class of cluster applications may additionally lead to identifying upper and lower bounds on W (or specific intervals for W) that prove most effective for that class.

Remark 3: To reduce the updating effort of Case (3), the indexes i′ = p(s) and i″ = q(s) can be interchanged (hence also interchanging k′ and k″) to assure that |Nq(s)| ≤ |Np(s)|. (More comprehensive ways of reducing computation are identified in Sections 4 and 7.)

Remark 4: The justification of terminating the outer loop of the algorithm when W = Large (after setting W = Wnextat the conclusion of the inner loop) derives from the observation that Wnext = Large implies the condition c(e(s)) > W + MinCost0 is never satisfied in Case (2). (When this terminating condition occurs in a connected graph, the method will have generated a min cost spanning tree.) Moreover, if the algorithm is repeated for W = Large, the same outcome will result.

Remark 5: When Wo = 0 (or Wo = 1 for the multiplicative case), each resulting node-disjoint subgraph (N_k, E_k) in the collection C(W) consists of a tree in which the cost c(e) for all edges $e \in E_k$ is the same.

Remark 6: In a complete graph, the algorithm will leave at most one node isolated (with $N_k = \{k\}$ and $E_k = \emptyset$) at the conclusion of the Inner Loop for any W. In a graph that is not complete or not connected, no node that is not isolated in G will be left isolated in the collection C(W) for W sufficiently large. (To permit additional isolated nodes, a limit c_{lim} may be imposed that prevents C(W) from including any edges e such that $c(e) > c_{lim}$.)

Remark 7: When there are tied (duplicate) cost values c(e), all orderings of e(1) to e(|E|) satisfying c(e(1)) ≤ c(e(2)) ≤ … ≤ c(e(|E|)) will produce the same collection of clusters C(W) in the following sense: For a given value of W, all orderings will produce the same node sets N_k defining C(W), and the sum of costs over the edge sets E_k will also be the same, though the edges within these sets may differ.

4. Fundamental relationships for accelerating the algorithm

A number of key relationships hold for the C(W) Algorithm that make it possible to accelerate its execution. We discuss the relationships here in broad outline and then incorporate them in Section 7 within a template for a computer code that applies not only to C(W) but to additional related types of cluster collections C(Y) and C(Z) whose algorithms are described in Sections 5 and 6.

4.1. Early termination of the inner loop

The Inner Loop can typically terminate far in advance of satisfying the condition s = sLast for sLast = |E|, hence making it unnecessary to examine all edges of the graph.

First note that the process of examining the edges in ascending cost order implies that once c(e(s)) > W + MinCost(k) for a given k = ko∈ K, then the inequality c(e(v)) > W + MinCost(ko) will also hold for all subsequent edges e(v) for v > s. Hence, by Case (2) of the algorithm, no nodes or edges will be adjoined to the cluster sets N_{ko} and E_{ko} for v > s. In addition, it will be unnecessary to update Wnext by reference to ko in the future.

It may further be observed that the MinCost(k) values are generated in a sequence that makes it possible to readily identify (without sorting) the values k(1), k(2), …, k(m), so that Min Cost(k(1)) ≤ MinCost(k(2)) ≤ … ≤ MinCost(k(m)). It is convenient to define m so that these values refer just to those k ∈ K such that MinCost(k) < Large. (Recall that MinCost(k) = Large implies that N_k consists of a single node k, and E_k = ∅.)

Thus if c(e(s)) > W + MinCost(k(m)), we know that none of the clusters indexed from k(1) to k(m) can take part in the creation of new clusters. Alternatively, if we start by checking whether c(e(s)) > W + MinCost(k(h)) holds for h = 1 and work forward until finding the first index k(h*) for which the inequality does not hold, then on future encounters with Case (2) it is possible to start from k(h*) rather than k(1) to begin checking whether c(e(s)) > W + MinCost(k(h)).

In consideration of these relationships, it should be kept in mind that when two clusters k′ and k″ are joined, then MinCost(k″) will no longer be referenced (since the cluster k″ will no longer exist). To see the consequences of this, suppose that k′ and k″ are interchanged, if necessary, so that MinCost(k′) ≤ MinCost(k″). Then when $N_{k''}$ is absorbed into $N_{k'}$, the following two possibilities arise:

i. MinCost(k′) < Large (hence MinCost(k′) identifies the cost of an edge previously added) and MinCost(k′) will be unchanged;

ii. MinCost(k′) = Large, and the new MinCost(k′) will be the value c(e(s)) of the edge e(s) currently added.

This implies that in the sequence MinCost(k(1)) ≤ MinCost(k(2)) ≤ … ≤ MinCost(k(m)), the value MinCost(k″) will drop out, and the value MinCost(k′) will either be unchanged and retain its position, or else it will change from a Large value to become the new value MinCost(k(m)) at the end of the ordered list.

However, applying this knowledge to shortcut the checks performed in Case (2) does not make it possible to save appreciable computation, since the amount of effort to perform the checks of Case (2) is not great in any case. Instead, we can make use of the foregoing relationships in a simpler manner without having to keep track of the values k(1), k(2), …, k(m).

To accomplish this, we record the number of elements n_k in each node set N_k by initializing all $n_k = 1$, and then setting $n_{k'}: = n_{k'} + n_{k''}$ when $N_{k''}$ is absorbed into $N_{k'}$ in Case (3). We also record the number of times t(i) each node i is encountered as a node i′ = p(s) or i″ = q(s) by initializing t(i) = 0 for all i, and then setting t(i′): = t(i′) + 1 and t(i″): = t(i″) + 1 when the edge e(s) is examined in the prelude to Case (1)of the algorithm (and also for i′ and i″ in the Initialization). Note that t(i) is bounded by tMax(i) which is the number of nodes adjacent to i in the graph G (where tMax(i) = n – 1 if G is complete).

We are interested in determining when t(i) = tMax(i) for an isolated node. We can conveniently identify the condition of being isolated by i = L(i). In conjunction with the preceding records, this makes it possible to keep track of the number nTrack of nodes that cannot take part in any further steps of adding an edge to C(W), and hence permitting the inner loop to terminate when nTrack = n.

Specifically, by initializing nTrack = 0, the first time c(e(s)) > W + MinCost(k)occurs for a given k = k′ or k″ in Case (2), we set nTrack: = nTrack + n_k. (To identify this first occurrence, initialize FirstTime(k) = True, and then set FirstTime(k) = False at the point of setting nTrack: = nTrack + nk.) We also set nTrack: = nTrack + 1 whenever t(i) is incremented for i = i′ and i″ in the prelude to Cases (1) to (3) to yield t(i) = tMax(i) under the condition that i = L(i). By checking for nTrack = n at each point where nTrack changes its value, we can then terminate the inner loop when this condition occurs.

Having performed the foregoing operations to terminate early for W = Wo, we may take advantage of another useful relationship to terminate early for all W > Wo. In particular, let sEnd(W) equal the value of s for the final edge e(s) added to C(W) for a given W. Then for values W′ and W″ such that W″ > W′, we are assured that sEnd(W″) ≤ sEnd(W′). Consequently, we can exploit this fact by introducing a variable sEnd which is set to sEnd = s at the conclusion of Case (3), which will cause sEnd to be the index s of the final edge added in constructing the current C(W). Then it is only necessary to set sLast = sEnd after the termination of the Inner Loop, thus overriding the initialization sLast = |E| to permit the next execution of the Inner Loop to terminate earlier. We can also allow the final execution of the Inner Loop to terminate earlier by the fact that the spanning tree generated on this execution will have n − 1 edges, while all other constructions must have fewer than this number of edges.

4.2. Advanced starting for successive W values

We can also accelerate the computation of the algorithm by saving information to produce an advanced start on successive executions of the Inner Loop. The underlying relationships are as follows.

Let s2(1), s2(2), …, s2(v_2) denote the s indexes starting with s2(1) = 1 (when Wnext = Large) where the values s2(v) for v > 1 identify successive edges e(s) for which a new (smaller) value of Wnext is identified in Case (2). Also, starting with W(1) = Large, let W(1), W(2), …, W(v″) denote the corresponding values for Wnext identified at these points (hence, for $v_2 > 1$, W(1) > W(2) > … > W(v_2)). Similarly, let s3(1), s3(2), …, s3(v_3) denote the s indexes starting with s3(1) = 1 where the values s3(v) for v > 1 identify successive edges e(s) that are added in Case (3) of the Inner Loop(to generate the current cluster collection C(W)).

After completing the Inner Loop for W = Wo (while saving this information), upon assigning W the value Wnext = W(v_2), the fact that Wnext < W(v) for v < v_2 implies that the algorithm will perform exactly the same sequence of steps until reaching s = s2(v_2), at which point the edge e(s) for s = s2(v_2), will be added to the construction (although this edge was not added on the previous execution of the inner loop).

Consequently, all edges e(s3(v)) for s3(v) < s2(v_2) will again be added to the current construction, and the values s2(v) for v < v_2 will also be unchanged. Hence, letting v* = Max(v: s3(v) < s2(v_2)) we can start the current construction by simply adding the edges e(s) for s = s3(1) to s3(v*), followed by adding the edge e(s) for s = s2(v_2) (whose index s2(v_2) therefore becomes recorded as the new index s3(v* + 1)). Then the customary Inner Loop for W > Wo can be executed starting with s initialized by setting s = s2(v_2) instead of s = 1. Subsequent executions of the Inner Loop continue to save the same information, which is used again to create an advanced start in the manner described.

By this means, we avoid examining all edges e(s) for s < s2(v_2) that were not added to the previous construction. We also avoid having to re-do the checks to determine that the remaining edges qualify to be added. Together this can amount to a considerable savings in computation.

A possibility arises to save additional computation by using more memory. Each time a new candidate for Wnext is identified, in the process of identifying the indexes s2(1), s2(2), …, s2(v_2), we can save a current copy of the arrays N_k, E_k, MinCost(k) and K used by the algorithm, avoiding the burden of excessive memory by overwriting the previous copy each time a new one is made. Then the latest copy will be available at the point where the edge e(s) for s = s2(v_2) is added on the current execution of the loop, making it possible to recover the arrays without having to regenerate them to resume the current loop.

However, it may not be possible to take advantage of a current copy of the saved arrays on every iteration of the Inner Loop (unless previous copies are not overwritten when new ones are made). After re-starting by recovering the arrays for s2(v_2), if now a new Wnext value is determined for s > s2(v_2) (referring to the v_2 of the previous execution) then we can proceed by again making a copy of the arrays for the next execution of the loop. But if it no new value of Wnext is found for s > s2(v_2), then the previous value W(v_2–1) (for s = s2(v_2–1) will be the new final Wnext value, and no copies of the arrays remain in memory for this value.

Consequently, in this latter case we resort to the construction that does not rely on the copied arrays, generating the arrays instead in the process of adding edges. Thus, on the next execution of the inner loop we will again have the copies available. Hence in this fashion we will be able to take advantage of the copied arrays at least on every second execution of the loop, if not more frequently.

As previously noted, the foregoing relationships and their implications are embodied in a format suitable for creating a computer code in Section 7, after we first describe two additional algorithmic variants that can be exploited by analogous relationships.

5. Algorithm C(Y): a node-based algorithmic variant

It is possible to formulate a node-based variant of the C(W) algorithm which follows a closely related format and is supported by a similar rationale.

In the node-based approach, we replace the parameter W by a parameter Y which is linked to costs associated with nodes in N_k rather than to costs associated with edges in E_k. (More precisely, the costs associated with nodes are also derived from edges—i.e., the edges that meet these nodes—though these edges are different from those referenced in the C(W) algorithm.)

Accompanying this parameter change, we replace the value MinCost(k) associated with the sets indexed by k ∈ K with a value MinCostB(i) associated with the nodes i∈ N, and more particularly, we replace MinCost(k′) for k′ = L(i′) by MinCostB(i′), and replace MinCost(k″) for k″ = L(i″) by MinCostB(i″)).

This replacement changes the updating rule when $N_{k''}$ is absorbed into $N_{k'}$ in Case (3). Specifically, the values MinCostB(i′) and MinCostB(i″)) are updated by setting MinCostB(i): = Min(MinCostB(i),c(e(s)) for i = i′ and i″, in contrast to the update involving MinCost(k′) and MinCost(k″) (which setsMinCost(k′): = Min(MinCost(k′),MinCost(k″), c(e(s))).

The reason for these changes is as follows. In the node-based version, to permit the edge e(s) = (i′, i″) (= (p(s),q(s))) to be added and hence to join the subgraphs ($N_{k'}$, $E_{k'}$) and ($N_{k''}$, $E_{k''}$), we require that c(e(s)) ≤ Y + MinCostB(i) for both i = i′ and i″. Hence we require c(e(s)) ≤ Y + MinCostB0, for MinCostB0 = Min(MinCostB(i′), MinCostB(i″)). On the other hand, if c(e(s)) > Y + MinCostB0, we are prevented from adding edge e(s), and by the preceding relationships this causes the first part of Case (2) to retain exactly the same form as in the C(W) algorithm.

To update MinCostB(i′) and MinCostB(i″) in Case (3), we must account for the fact that each of these two values is affected only by the cost of the edge e(s), and hence will either retain its present value or become equal to c(e(s)), according to which is smaller. (It may be noted that once node i for i = i′ or i″ has been assigned an edge cost c(e(s)), then MinCostB(i) will not change thereafter, since any edge e(s) that is added later to meet node I will have a cost no less than that of the earlier edge.)

Based on these observations, we can state the form of the C(Y) algorithm as follows.

C(Y) Algorithm (Node-Based Version)

Inputs: The graph G(N, E), cost vector c(e), e ∈ E, initial Yo value for Y.

Edges are ordered so that the costs satisfy c(e(1)) ≤ c(e(2)) ≤ … ≤ c(e(|E|)).

Set Y = Yoand sLast = |E|.

Begin Outer Loop

While Y < Large

```
    *Initialization(A)*. Set Ynext = Large, $K = \{1, \ldots, n\}$, and for each $k \in K$ let $L(k) = k$,
    $Nk = \{k\}$, $E_k = \varnothing$, and MinCostB(k) = Large.
    *Initialization(B)*. Let $i' = p(1)$ and $i'' = q(1)$ and select $e(1)$ $(= (i', i''))$ by identifying $k' = L(i')$ and $k'' = L(i'')$ and absorbing $N_{k''}$ into $N_{k'}$ to create the cluster $N_{k'}: = N_{k'} \cup N_k{}'' = \{i', i''\}$ with edge set $E_{k'} = e(1)$. Set MinCostB(k′) = c(e(1)) and set $K: = K \setminus \{k''\}$. Finally, initialize the edge index s by setting $s = 1$.
    **Begin Inner Loop**
    While $s < sLast$|
        Set $s: = s + 1$ and identify edge $e(s) = (p(s), q(s))$. Let $i' = p(s)$, $i'' = q(s)$ and let $k' = L(i')$ and $k'' = L(i'')$.
        Case (1): If $k' = k''$ ($i'$ and $i''$ belong to the same cluster), then continue to the next iteration of the Inner Loop.
        Case (2): If $c(e(s)) > Y +$ MinCostB0, for MinCostB0 = Min(MinCostB(i′), MinCostB(i″)), then compute Ynext = Min(Ynext, c(e(s)) – MinCostB0) and continue to the next iteration of the Inner Loop.
        Case (3) (If (1) and (2) do not apply): Absorb $N_{k''}$ into $N_{k'}$ to create $N_{k'}: = N_{k'} \cup N_{k''}$ with its associated edge set $E_{k''}: = E_{k'} \cup E_{k''} \cup \{e(s)\}$. Set $L(i) = k'$ for all $i \in N_k{}''$, and setMinCostB(i): = Min(MinCostB(i), c(e(s)) for $i = i'$ and $i''$. Finally, set $K: = K \setminus \{k''\}$.
    Endwhile.
    Y = Ynext.
Endwhile.
**End of C(Y) Algorithm**
```

The Remarks concerning the C(W) algorithm in Section 3 can be applied as well to the C(Y) algorithm.

Now we show how to join the C(W) and C(Y) algorithms.

6. Algorithm C(Z): a combination of C(W) and C(Y)

Each of the C(W) and C(Y) algorithms has features lacking in the other. However, the C(Y) algorithm has a potential deficiency, which resides in the fact that it is subject to “drift”—a phenomenon where the costs of edges in an edge set E_k can grow along a chain, where each new edge added to E_k has a higher cost than the previous one. Such an eventuality can arise because the cost of an edge in a chain is limited only by the cost of the previous edge.

It is possible to combat drift and also take advantage of the different features of the C(W) and C(Y) algorithms by joining these algorithms to create an algorithm C(Z) that incorporates the MinCost evaluation criteria of both C(W) and C(Y) simultaneously.

Let α be a nonnegative weight applied to the edge selection criterion of C(W) and let $\beta = 1 - \alpha$ be a nonnegative weight applied to the edge selection criterion of C(Y). We construct Algorithm C(Z) so that it will be the same as C(W) if $\alpha = 1$ and will be the same as C(Y) if $\alpha = 0$ ($\beta = 1$).

For notational convenience, we refer to the value MinCost(k) of the C(W) algorithm as MinCostA(k). Then the MinCost evaluation criterion of C(Z) is given by.

$$\text{MinCostC(i)} = \alpha\cdot\text{MinCostA(k)} + \beta\cdot\text{MinCostB(i)}, \text{ for } k = L(i).$$

To apply this criterion, we create values MinCostC(i′) and MinCostC(i″) for nodes i′ = p(s) and i″ = q(s), and for k′ = L(i′) and k″ = L(i″), given by

$$\text{MinCostC}(i') = \alpha\cdot\text{MinCostA}(k') + \beta\cdot\text{MinCostB}(i')$$

$$\text{MinCostC}(i'') = \alpha\cdot\text{MinCostA}(k'') + \beta\cdot\text{MinCostB}(i'')$$

Associated with the foregoing values, we define

$$\text{MinCostC0} = \text{Min}(\text{MinCostC}(i'), \text{MinCostC}(i''))$$

We state the C(Z) algorithm by reference to these definitions.

C(Z) Algorithm.

Inputs: The graph G(N, E), cost vector c(e), e ∈ E, initial Zo value for Z.

Edges are ordered so that the costs satisfy c(e(1)) ≤ c(e(2)) ≤ … ≤ c(e(|E|)).

Set Z = Zo and sLast = |E|

Begin Outer Loop

While Z < Large

Initialization(A). Set Znext = Large, K = N (= {1, …, n}), and for each k ∈ K let.

L(k) = k, N_k = {k}, E_k = ∅, and MinCostA(k) = MinCostB(k) = MinCostC(k) = Large.

Initialization(B). Let i′ = p(1) and i″ = q(1) and select e(1) (= (i′, i″)).Set k′ = L(i′) and k″ = L(i″) and absorb $N_{k''}$ into $N_{k'}$ to create the cluster $N_{k'}$: = $N_{k'}$∪ N_k″= {i′, i″} with edge set $E_{k'}$ = e(1). Set MinCostA(k′) = c(e(1)) and MinCostB(i) = c(e(1)) for i = i′ and i″. Set.

K: = K \ {k″} and s = 1.

```
    Begin Inner Loop

    While s < sLast

        Set s: = s + 1 and identify edge e(s) = (p(s), q(s)). Let i′ = p(s), i″ = q(s) and let k′ =
        L(i′) and k″ = L(i″).

        Case (1): If k′ = k″, then continue to the next iteration of the Inner Loop.

        Case (2): If c(e(s)) > Z + MinCostC0 (for MinCostC0 determined by the preceding
                definitions), then compute Znext = Min(Znext, c(e(s)) – MinCostC0) and
                continue to the next iteration of the Inner Loop.

        Case (3) (If (1) and (2) do not apply): Absorb $N_{k''}$ into $N_{k'}$ to create the larger clus
                ter $N_{k'}: = N_{k'} \cup N_{k''}$ with its associated edge set.

                $E_{k''}: = E_{k'} \cup E_{k''} \cup \{e(s)\}$.

        Correspondingly, update L(i) by setting L(i) = k′ for all i∈ Nk″, and set MinCostA(k′):
        = Min(MinCostA(k′),MinCostA(k″), c(e(s)); MinCostB(i): = Min(MinCostB(i),c(e(s))
        for i = i′ and i″ (thus yielding MinCostC(i) by the definitions above).Finally, set K:
        = K \ {k″}.

    Endwhile

    Z = Znext

Endwhile

End of C(Z) Algorithm
```

The next section shows how to carry out accelerated updates in the context of the preceding algorithm.

7. Implementing accelerated updates for the C(Z) algorithm

We build on the relationships identified in Section 4 to apply them to the more general C(Z) algorithm. By the implications described earlier, our general updates also apply directly to the C(W) and C(Y) algorithms by respectively replacing Z with W and Y (hence Znext with Wnext and Ynext) and replacing MinCostC(·) by MinCost(·) and MinCostB(·).

7.1. Early termination for the C(Z) inner loop

Early termination for the Inner Loop of C(Z) is effected by creating an Initialization(C) immediately following Initialization(B) (hence inheriting the assignments of Initialization(B)) and modifying the Inner Loop as follows. We apply these changes for Z = Zo and thereafter take advantage of setting sLast = sEnd as indicated below, in order to allow for the advanced updating of C(Z) for successive Z values.

Initialization(C) for Z = Zo:

Set $n_k = 1$, $k \in K \setminus \{k'\}$, and $n_{k'} = 2$ (hence $n_k = |N_k|$, $k \in K$); set t(i) = 0 for $i \in N \setminus \{i', i''\}$and set t(i′) = t(i″) = 1 (hence t(i) identifies the number of edges e(s) = (i,j) for all s currently examined in the Inner Loop (and its initialization). Finally, set nTrack = 0 and set FirstTime(k) = True, k $\in$ K (to identify the sets that have not been prevented from having an edge added to them).

Modification of Inner Loop for Z = Zo:

In the prelude to Case (1): Execute the following for i = i′ and i″: If i = L(i) then set t(i): = t(i) + 1 and if (now) t(i) = tMax(i) then set nTrack: = nTrack +1 and if (now) nTrack = n terminate the Inner Loop. (tMax(i) denotes the number of nodes adjacent to i in the graph G.)

Re-organize Case (2) to become as follows: Execute the following for k = k′ and k = k″: If FirstTime(k) = True, then if c(e(s)) > Z + MinCostC(k): set FirstTime(k) = False, set nTrack: = nTrack + $n_{k'}$, compute Znext = Min(Znext, c(e(s)) – MinCostC(k) and if (now) nTrack = n, terminate the Inner Loop.After performing the preceding for both k′ and k″: If FirstTime(k) = False for k = k′ or k = k″, then continue to the next iteration of the Inner Loop. (Note: Case (2) could also be moved to precede Case (1).)

In Case (3) (when $N_{k''}$ is absorbed into $N_{k'}$): Set $n_{k'}$: = $n_{k'}$ + n_k″.

Modifications for all Z values:

Set sEnd = s at the end of Case (3) and set sLast = sEnd immediately after the conclusion of the Inner Loop (following Z = Znext).

For early termination of the last execution of the Inner Loop:

Set $n_e = 1$ in Initialization(B). Then at the end of Case (3) set $n_e = n_e + 1$ and if $n_e = n - 1$ terminate the Inner Loop. (The Outer Loop will automatically terminate as well, by the condition Z = Large.)

Because of the special modifications for Z = Zo that do not apply for Z > Zo, it is convenient to insert the portion of the algorithm for Z = Zo at the very beginning of the C(Z) Algorithm, before the Outer Loop.

7.2. Advanced updating for successive Z values

We draw on the relationships of Section 4.2 to create the instructions for updating C(Z) to reduce the amount of computation required on successive iterations of the Inner Loop.

We adopt the following notation of Section 4.2, re-expressed in terms of the C(Z) algorithm: s2(1), s2(2), …, s2(v_2)identifies the successive s indexes that occur each time a new (smaller) value of Znext is identified in Case (2) of the Inner Loop, beginning with the initialized value s2(1) = 1. Similarly, s3(1), s3(2), …, s3(v_3)identifies the s indexes of successive edges e(s) that are added in Case (3), beginning with the initialized value s3(1) = 1. (In the re-organized Case (2) for Z = Zo in Section 7.1, the value Znext may be reduced twice for a given edge e(s) and we only consider the last (smaller) Znext value produced for e(s).) The sequence Z(1), Z(2), …, Z(v_2) with the initialization Z(1) = Large, denotes the corresponding candidate values for Znext generated in Case (2). We therefore have Z(1) > Z(2) > … > Z(v_2), where the final Znext is given by Znext = Z(v_2).

Assume the algorithm for Z = Zo has been inserted before the start of the Outer Loop, as indicated in Section 7.1.

Modification of Initialization(B) for Z = Zo only:

Add $v_2 = v_3 = 1$; s2(1) = s3(1) = 1 and Z(1) = Large.

Modification of the Inner Loop for all Z:

In Case (2) when Znext is updated (reduced): set $v_2 = v_2 + 1$; s2(v_2) = s, and Z(v_2) = Znext.

In Case (3) upon adding e(s): set $v_3 = v_3 + 1$; s3(v_3) = s.

Modification After the Inner Loop for all Z:

Following the instructions Z = Znext, and sLast = sEnd:If $v_2 = 1$, terminate.

Modification Before the Inner Loop for Z > Zo:

Insert a Preliminary Loop before the Inner Loop for Z > Zo as follows:

(After Initialization(A) and Initialization(B))

Preliminary Loop

```
Set v* = Max(v: s3(v) < s2($v_2$)); $v_3$ = v* + 1; s3($v_3$) = s2($v_2$); $v_2$: = $v_2$–1.
v = 1
While v < $v_3$
    v: = v + 1
    s = s3(v)
    Insert the prelude to Case (1) followed by Case (3) (excluding the modification above
      that adds $v_3 = v_3 + 1$; s3($v_3$) = s)
EndWhile[2]
Set s = s3($v_3$)
```

7.3. Modifications involving additional memory

Added Modification to Initialization for Z = Zo:

Copy = False (where Copy indicates whether a copy is made of the arrays indicated in the modification below).

Added Modification of the Inner Loop for all Z:

Each time Case (2) yields a new value of Znext (after checking for both k′ and k″ for Z = Zo) record a copy of ne, the set K and arrays N_k,E_k, MinCostA(k), k ∈K and the arrays L(i),

[2]As before, we adopt the convention whereby the loop is bypassed if $v_3 = 1$ but will be executed on the iteration where v: = v + 1 results in $v = v_3$.

MinCostB(i), MinCostC(i), i∈ N (writing over any previous copy). Let Copy = True if such a copy is made. (As remarked earlier, it is not necessary to keep track of the E_k array.)

Added Modification for the Preliminary Loop

If Copy = False, execute the Preliminary Loop. Otherwise, if Copy = True, instead of executing the Preliminary Loop read the copies of the saved arrays into the active form of these arrays, followed by setting s = s2(v_2) and Copy = False. (The Inner Loop immediately follows this modification.)

The complete C(Z) Algorithm that incorporates all of these changes is shown in the Appendix for convenience.

8. Conclusions

The new classes of tree-based clustering algorithms represented by C(W), C(Y) and in the most general case by C(Z), afford the possibility to generate clusters with a range of different characteristics as the parameters W, Y and Z are varied. The fact that the key parameter can be varied adaptively to generate all cluster collections in its class without duplication invites empirical studies to identify parameter ranges that are effective for particular types of clustering applications.

The C(Z) Algorithm can be made still more general by changing the implicit definition of MinCostC(i) for i = i′ and i″ by defining MinCostA(k), for k = k′ and k″, to be any convex combination of the costs of edges in the set $E(N_k) = \{e = (i,j) \in N_k\} \subset E$ (hence $(N_k, E(N_k))$ is the subgraph of G spanned by the nodes of N_k) and defining MinCostB(i) to be any convex combination of the edges of $E(N_k)$ that are adjacent to node i in G. However, this requires updating these MinCost values in a more complex way than in the current form of Case (3).

Future work to exploit the properties of these algorithms can include an investigation of the choice of the parameter α (and hence $\beta = 1 - \alpha$) in Algorithm C(Z) to similarly determine ranges that are effective for particular types of clustering applications.

Acknowledgements

This research was supported in part by the Key Laboratory of International Education Cooperation of Guangdong University of Technology and by the Fundamental Research Funds for the Central Universities (No3102017zy059).

A. Appendix

A.1. Algorithm C(Z) with accelerated updates

We identify the form of the C(Z) algorithm that results by incorporating all of the accelerated updates of Section 7. As before, the sets E_k do not need to be maintained unless they are of interest for experimental purposes. The case where added memory is used is identified in Case (2) so that this memory need not be used if desired. (In that case, the variable Copy will always remain False, as in Initialization(B).)

C(Z) Algorithm with Accelerated Updates

```
Inputs: The graph $G(N, E)$, cost vector $c(e)$, $e \in E$, initial Zo value for Z.
Edges are ordered so that the costs satisfy $c(e(1)) \le c(e(2)) \le \ldots \le c(e(|E|))$.
Set $Z = Zo$ and $sLast = |E|$. Copy = False.
Execute Routine for Z = Zo (Given Below)
Begin Outer Loop for Z > Zo
While $Z <$ Large
    Initialization(A). Set Znext = Large, $K = N$ (= $\{1, \ldots, n\}$), and for each $k \in K$ let.
    $L(k) = k$, $N_k = \{k\}$, $E_k = \varnothing$, and MinCostA(k) = MinCostB(k) = MinCostC(k) = Large.
    Initialization(B). Let $i' = p(1)$ and $i'' = q(1)$ and select $e(1)$ (= $(i', i'')$).Set $k' = L(i')$ and $k'' = L(i'')$ and absorb $N_{k''}$ into $N_{k'}$ to create the cluster $N_{k'}: = N_{k'} \cup N_k'' = \{i', i''\}$ with edge set $E_{k'} = e(1)$. Set MinCostA(k′) = c(e(1)) and MinCostB(i) = c(e(1)) for i = i′ and i″ and set $K: = K \setminus \{k''\}$. Then set $n_e = 1$ and $s = 1$.
    If (Copy = False) then.
        Execute Preliminary Loop
        Set $v^* = Max(v: s3(v) < s2(v_2))$; $v_3 = v^* + 1$; $s3(v_3) = s2(v_2)$; $v_2: = v_2 - 1$.
        $v = 1$
        While $v < v_3$
            $v: = v + 1$
            $s = s3(v)$
            Identify edge $e(s) = (p(s), q(s))$. Let $i' = p(s)$, $i'' = q(s)$ and let $k' = L(i')$ and $k'' = L(i'')$.
            Execute Case (3): Absorb $N_{k''}$ into $N_{k'}$ by setting $N_{k'}: = N_{k'} \cup N_{k''}$ with its associated edge set $E_{k''}: = E_{k'} \cup E_{k''} \cup \{e(s)\}$. Set $L(i) = k'$ for all $i \in N_k''$, and set MinCostA(k′): = Min(MinCostA(k′),MinCostA(k″), c(e(s)); MinCostB(i): = Min(MinCostB(i),c(e(s)) for i = i′ and i″ (thus yielding MinCostC(i) by the definitions of Section 7). Set $K: = K \setminus \{k''\}$ and sEnd = s. Set ne: = $n_e + 1$ and if $n_e = n - 1$ terminate the C(Z) Algorithm.
        EndWhile
    Set $s = s3(v_3)$
  Else
    Read the latest copy of $n_e$ and the arrays saved in Case (2) of the Inner Loop into the active form of these arrays. Set $s = s2(v_2)$ and Copy = False.
    Endif
    **Begin Inner Loop**
    While s < sLast
        Set s: = s + 1 and identify edge e(s) = (p(s), q(s)). Let i′ = p(s), i″ = q(s) and let k′ = L(i′) and
        k″ = L(i″).
        Case (1): If k′ = k″, then continue to the next iteration of the Inner Loop.
        Case (2): If c(e(s)) > Z + MinCostC0, then compute Znext = Min(Znext, c(e(s)) –
            MinCostC0), set $v_2 = v_2 + 1$, $s2(v_2) = s$, $Z(v_2) = Znext$.
            *For added memory case*:
            Set Copy = True and record a copy of ne, the set K and arrays $N_{k}$, $E_{k}$, MinCostA(k),
            $k \in K$ and the arrays L(i), MinCostB(i), MinCostC(i), $i \in N$ (writing over any previous
            copy).
            Continue to the next iteration of the Inner Loop.
        Case (3): Absorb $N_{k''}$ into $N_{k'}$ by setting $N_{k'}: = N_{k'} \cup N_{k''}$ with its associated edge set $E_{k''}$:
                =$E_{k'} \cup E_{k''} \cup \{e(s)\}$. Set L(i)=k′ for all $i \in N_{k}''$, and set MinCostA(k′):=Min(MinCostA(k′),
                MinCostA(k″), c(e(s)); MinCostB(i): = Min(MinCostB(i), c(e(s)) for i = i′ and
                i″ (thus yielding MinCostC(i) by the definitions of Section 7). Set K: = K \ {k″}
                and sEnd = s.Set $v_3 := v_3 + 1$, $s3(v_3) = s$, ne:= $n_e + 1$ and if $n_e = n - 1$ terminate the
                Inner Loop.
        Endwhile
        Z = Znext
        sLast = sEnd
        If $v_2 = 1$ terminate the C(Z) Algorithm
Endwhile
**End of C(Z) Algorithm**
**Routine for Z = Zo**
            *Initialization(A)*. Set Znext = Large, K = N (= {1, …, n}), and for each $k \in K$ let.
            L(k) = k, $N_k = \{k\}$, $E_k = \varnothing$, and MinCostA(k) = MinCostB(k) = MinCostC(k) = Large.
            *Initialization(B)*. Let i′ = p(1) and i″ = q(1) and select e(1) (= (i′, i″)).Set k′ = L(i′) and.
            k″ = L(i″) and absorb $N_{k''}$ into $N_{k'}$ to create the cluster $N_{k'}: = N_{k'} \cup N_{k}'' = \{i', i''\}$ with
            edge set $E_{k'}$ = e(1). Set MinCostA(k′) = c(e(1)) and MinCostB(i) = c(e(1)) for i = i′ and
            i″ and set K: = K \ {k″}. Set $v_2 = v_3 = 1$; s2(1) = s3(1) = 1 and Z(1) = Large. Set $n_e = 1$
            and s = 1.
            Initialization(C): Set nk = 1, $k \in K \setminus \{k'\}$, and $n_{k'} = 2$; set t(i) = 0 for $i \in N \setminus \{i', i''\}$ and
            set t(i′) = t(i″) = 1. Set nTrack = 0 and set FirstTime(k) = True, $k \in K$.
```

```
Begin Inner Loop

    While s < sLast

        Set s: = s + 1 and identify edge e(s) = (p(s), q(s)). Let i′ = p(s), i″ = q(s) and let k′ = L(i′) and k″ = L(i″). For i = i′ and i″: If i = L(i) then set t(i): = t(i) + 1 and if (now) t(i) = tMax(i) then set nTrack: = nTrack +1 and if (now) nTrack = n terminate the Inner Loop.

        Case (1): If k′ = k″ (i′ and i″ belong to the same cluster), then continue to the next iteration of the Inner Loop.

        Case (2): Execute the following for k = k′ and k = k″: If FirstTime(k) = True, then if c(e(s)) > Z + MinCostC(k): set FirstTime(k) = False, set nTrack: = nTrack + $n_k$, compute Znext = Min(Znext, c(e(s)) – MinCostC(k) and if (now) nTrack = n, terminate the Inner Loop. After performing the preceding for both k′ and k″: If FirstTime(k) = False for k = k′ or k = k″ execute the following (and otherwise proceed to Case (3)): set $v_2 = v_2 + 1$, $s2(v_2) = s$, $Z(v_2)$ = Znext.

        For added memory case:

        Set Copy = True and record a copy of ne, the set K and arrays $N_{k'}$, $E_{k'}$, MinCostA(k), $k \in K$ and the arrays L(i), MinCostB(i), MinCostC(i), $i \in N$ (writing over any previous copy).

        Continue to the next iteration of the Inner Loop.

        Case (3): Absorb $N_{k''}$ into $N_{k'}$ to create the larger cluster $N_{k'}: = N_{k'} \cup N_{k''}$ with its associated edge set $E_{k''}: = E_{k'} \cup E_{k''} \cup \{e(s)\}$. Correspondingly, update L(i) by setting L(i) = k′ for all $i \in N_k''$, and set MinCostA(k′): = Min(MinCostA(k′),MinCostA(k″), c(e(s)); MinCostB(i): = Min(MinCostB(i),c(e(s)) for i = i′ and i″ (thus yielding MinCostC(i) by the definitions of Section 7). Set K: = K \ {k″}, $n_{k'}: = n_{k'} + n_k''$, and sEnd = s. Set $v_3 := v_3 + 1$, $s3(v_3) = s$, $n_e := n_e + 1$ and if $n_e = n – 1$ terminate the Inner Loop.

    Endwhile

    Z = Znext

    sLast = sEnd

    If $v_2 = 1$ then

        Terminate the C(Z) Algorithm

    Else.

        $v^* = Max(v: s3(v) < s2(v_2))$; $v_3 = v^* + 1$; $s3(v_3) = s2(v_2)$; $v_2 : = v_2 – 1$.

    Endif

End of Routine for Z = Zo
```

Author details

Fred Glover[1]* and Yang Wang[2]

*Address all correspondence to: fredwglover@yahoo.com

1 School of Engineering and Science, University of Colorado, Boulder, CO, USA

2 School of Management, Northwestern Polytechnical University, Xian, China

References

[1] Anderberg MR. Cluster analysis for applications. In: Monographs and Textbooks on Probability and Mathematical Statistics. New York: Academic Press, Inc.; 1973

[2] Stefik MJ. Machine learning: An artificial intelligence approach. R.S. Michalski, J.G. Carbonell and T.M. Mitchell, (Tioga, Palo Alto, CA); 572 pages, $39.50. Artificial Intelligence. 1985;**25**(2):236-238

[3] Michalski RS, Carbonell JG, Learning TMMM. An artificial intelligence approach. Understanding the Nature of Learning. 1983;**2**:3-26

[4] Rinzivillo S, Pedreschi D, Nanni M, Giannotti F, Andrienko N, Andrienko G. Visually driven analysis of movement data by progressive clustering. Information Visualization. 2008;**7**(3):225-239

[5] Chen H, Chiang RHL, Storey VC. Business intelligence and analytics: From big data to big impact. MIS Quarterly. 2012;**36**(4):1165-1188

[6] Xu Y, Olman V, Xu D. Minimum spanning trees for gene expression data clustering. Genome Informatics. 2001;**12**:24-33

[7] Xu Y, Olman V, Xu D. Clustering gene expression data using a graph-theoretic approach: An application of minimum spanning trees. Bioinformatics. 2002;**18**(4):536-545

[8] Jana PK, Naik A, editors. An efficient minimum spanning tree based clustering algorithm. In: Proceedings of International Conference on Methods and MODELS in Computer Science; 2009

[9] Grygorash O, Zhou Y, Jorgensen Z, editors. Minimum spanning tree based clustering algorithms. In: IEEE International Conference on TOOLS with Artificial Intelligence; 2008

[10] Wang ZM, Soh YC, Song Q, Kang S. Adaptive spatial information-theoretic clustering for image segmentation. Pattern Recognition. 2009;**42**(9):2029-2044

[11] Gower JC, Ross GJS. Minimum spanning trees and single linkage cluster analysis. Journal of the Royal Statistical Society. 1969;**18**(1):54-64

[12] Laszlo M, Mukherjee S. Minimum spanning tree partitioning algorithm for microaggregation. IEEE Transactions on Knowledge and Data Engineering. 2005;**17**(7):902-911

[13] Asano T, Bhattacharya B, Keil M, Yao F, editors. Clustering algorithms based on minimum and maximum spanning trees. In: Symposium on Computational Geometry; 1988

[14] Päivinen N. Clustering with a minimum spanning tree of scale-free-like structure. Pattern Recognition Letters. 2005;**26**(7):921-930

[15] Wang X, Wang X, Wilkes DM. A divide-and-conquer approach for minimum spanning tree-based clustering. IEEE Transactions on Knowledge and Data Engineering. 2009;**21**(7):945-958