Efficient Hierarchical Clustering Algorithms using Partially Overlapping Partitions

Manoranjan Dash[1] and Huan Liu[2]

[1] School of Computing, National University of Singapore
[2] Dept. of Computer Sci. and Engg., Arizona State University

Abstract. Clustering is an important data exploration task. A prominent clustering algorithm is agglomerative hierarchical clustering. Roughly, in each iteration, it merges the closest pair of clusters. It was first proposed way back in 1951, and since then there have been numerous modifications. Some of its good features are: a natural, simple, and non-parametric grouping of similar objects which is capable of finding clusters of different shape such as spherical and arbitrary. But large CPU time and high memory requirement limit its use for large data. In this paper we show that geometric metric (centroid, median, and minimum variance) algorithms obey a 90-10 relationship where roughly the first 90iterations are spent on merging clusters with distance less than 10the maximum merging distance. This characteristic is exploited by partially overlapping partitioning. It is shown with experiments and analyses that different types of existing algorithms benefit excellently by drastically reducing CPU time and memory. Other contributions of this paper include comparison study of multi-dimensional vis-a-vis single-dimensional partitioning, and analytical and experimental discussions on setting of parameters such as number of partitions and dimensions for partitioning.

1 Introduction

Clustering is an important data exploration task. It is applied in different areas including data mining. Surveys on clustering algorithms can be found in [1, 9]. Among the more prominent clustering algorithms hierarchical clustering is one. It was first proposed in 1951 [6], and since then there have been numerous modifications including the recent ones [8, 10]. The fact that it has been a prominent algorithm for last half a century shows that it has stood the test of time. Among its various good features it is a non-parametric (assumes very little in the way of data characteristics), natural and simple way of grouping objects, and capable of finding clusters of different shapes such as spherical, arbitrary. But its large CPU time and high memory requirement make it unsuitable for large data. Efficient techniques to handle large data can be roughly classified as sampling, summarizing, and partitioning. Sampling has been used in several algorithms and recently in CURE [8]. Basically, clustering is done over a small sample and results are extended to the whole data set. Although it is criticized for missing out small clusters, sampling usually retains the underlying cluster structure. Summarizing algorithms are based on the fact that data points that are very close to each other can be merged and their summary information can be used for efficient clustering [5, 18]. Partitioning is used in [8] and in parallel algorithms in [12] where data is partitioned and each partition is then clustered and later the results are consolidated to determine a global set of clusters. These techniques help in reducing the size of the data while trying to retain the original cluster structure but they still require to run the traditional algorithms on the reduced data – a sample, a summary, or a partition. Improving the traditional algorithms will have complementary effect on these algorithms. This paper focuses on reducing the time and memory requirement of hierarchical algorithms.

2 Background

It is assumed that data contains N data points with each point x described by an M-tuple $x = (x_1, ..., x_M)$ of real numbers where M is the number of attributes or dimensions [1]. Distance between two objects is usually calculated using *Minkowski* metrics where for fixed d and for any two objects x and y, $L_d(x, y) = \{\sum_{j=1}^{M} |x_j - y_j|^d\}^{\frac{1}{d}}$. This family includes the *Manhattan* metric L_1, *Euclidean* metric L_2, and *Chebychev* metric L_∞. For ease of explanation it is assumed that distance is measured in *Euclidean* metric.

Hierarchical algorithms can be broadly classified as divisive and agglomerative. In divisive type, starting with one cluster that contains all data points, a cluster is divided in each iteration until each data point belongs to a distinct cluster. In agglomerative type, starting with each point in a distinct cluster, the closest pair of clusters are merged in each iteration until there remains only one cluster containing all data points. The output of hierarchical clustering is a dendrogram which is a hierarchy of divisions or agglomerations with the top level or root representing all points in one cluster, and the bottom level or leaves representing each point in a distinct cluster. The computation needed in an agglomerative algorithm to go from one level to another is usually simpler than a divisive algorithm. In this paper we are concerned with agglomerative algorithms. They can be classified into two categories depending on the type of similarity measure used, namely graph and geometric metrics. Algorithms that use graph metrics are single link, complete link, and average link, and those using geometric metrics are centroid, median and minimum variance. A basic difference between the two types is that in graph metrics each point is a representative while in geometric metrics each cluster has only one representative, eg. centroid. This difference is important from the point of view of performance and shape of clusters. For example, graph metric algorithm (eg. single link) can discover arbitrarily shaped clusters while geometric metric algorithm (eg. centroid method) is more suitable for clusters of spherical shape.

It is observed that for geometric metric algorithms, except for the last portion of the iterations, the size (i.e. number of points) and the merging distance of the closest pair of clusters is very small compared to their maximums respectively. It is further observed that initial iterations are typically much costlier

[1] Hierarchical clustering over binary and nominal data types are discussed in [7].

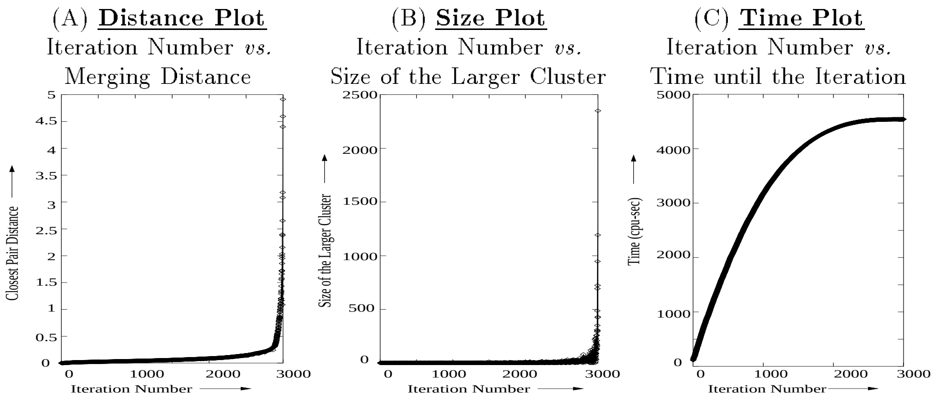


Fig. 1. Important observations for geometric metric algorithms

than those towards the end. See Figure 1 for plots showing these observations pictorially. The plots are based on the results obtained for a 2-D data set containing 3000 points in 100 clusters, each a Gaussian ball, distributed randomly in the feature space with some noise. Each dimension has a range [0.0 – 10.0]. This data is a simulated version of the data set DS2 reported in BIRCH [18]. All experiments here and later in the paper are run on Digital 8400 5/350 with 350 MHz processor and 1 GByte of ECC main memory. The results are for centroid type algorithm. In Figure 1(A) the merging distance is plotted against iteration number. Although the merging distance is not monotonically increasing, i.e. in a later iteration it can be larger than that in an earlier iteration, but still, roughly only in the last 10% or so iterations it is more than 10% or so of the maximum merging distance. This behavior is seen for other geometric metric algorithms and for different data including high-dimensional, multi-resolution, and skewed distribution. But it may not be observed for graph metrics. The reason is in geometric metric (eg. centroid type), a cluster is represented by single point (i.e. centroid); after merging the closest pair of clusters, the distance of the new cluster from most of the other clusters (except for those clusters very close to the bisecting hyper-plane between the two merging centroids) is larger than the smaller of the two distances before merging, whereas in graph metric (eg. single link) distance after merging will be equal to the smaller of the two distances before merging [2]. We will call these observations *90-10 relationship* which means *roughly 90% or so iterations from the beginning merge clusters that are separated by less than 10% or so of the maximum merging distance.*

Centroid and median types of geometric metric algorithms merge clusters whose centroids are the closest. Centroid type is known as unweighted as it treats each point in a cluster equally whereas median type is known as weighted as it weights all clusters the same, so points in small clusters are weighted more heavily than points in large clusters. The minimum variance Ward's method merges clusters that results in minimum change in square error [16]. These algo-

[2] Because of this characteristic single link algorithm follows reducibility property [11].

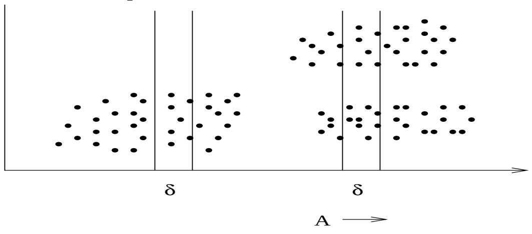
rithms observe the 90-10 relationship. In the rest of the paper centroid method will be discussed in detail except when otherwise specified. The first centroid algorithm proposed uses a similarity matrix. In each iteration the similarity matrix is searched to find the closest pair of clusters. Let us call this '*step 1*'. For n ($n = N...2$) clusters in an iteration, it requires $O(n^2)$ time to search through the similarity matrix, and for $N-1$ iterations this step takes $O(N^3)$ time. After merging the closest pair the similarity matrix is updated by deleting the column entries for the pair that merged and by creating a new row for the merged cluster by determining distance from other clusters. Let us call this '*step 2*'. For n clusters in an iteration this step takes $O(n)$ time to update, and so for $N-1$ iterations it takes $O(N^2)$ time. Memory required is $O(N^2)$ because of the similarity matrix. This simple algorithm can be improved by maintaining a nearest neighbor array that stores nearest neighbor for each cluster. This way *step 1* requires only to find the minimum of the nearest neighbor distances from the nearest neighbor array in $O(n)$ time for n clusters in any iteration. But *step 2* requires $O(n^2)$ time in each iteration if naive nearest neighbor algorithm of $O(n)$ time complexity is used. In 1984, Day and Edelsbrunner [2] suggested two ways to improve the time complexity of this algorithm: Type (1) – Obtain an improved bound on the required number of nearest neighbor updates – i.e. improve the *step 1*; Type (2) – Obtain an improved bound on the time required for each update – i.e. improve the *step 2*. For type (1) they suggested to use a heap-based priority queue that requires $O(\log n)$ time to find nearest neighbor giving an over-all time complexity of $O(N^2 \log N)$. For type (2) algorithm if there are α number of clusters to be updated after each iteration then the over-all complexity becomes $O(\alpha N^2)$. Using geometric preliminaries they proposed an upper bound for α as $2(3^M - 2)$ where M is the number of dimensions.

Anderberg classified hierarchical algorithms into 'stored matrix' and 'stored data' [1]. Stored matrix algorithms maintain a similarity matrix whereas stored data algorithms do not but instead calculate the similarities as required. A major distinction is that stored matrix methods are preferred when memory is sufficient to store the similarity matrix of size $O(N^2)$, otherwise stored data is the way out. Type (1) algorithm of previous paragraph that uses priority queues is a stored matrix method while type (2) algorithm is more suitable as stored data if M is not large and if memory is not enough to store $O(N^2)$ similarity matrix. Recent algorithms on hierarchical clustering [8, 10, 12] uses priority queues. The algorithms presented here improves both stored matrix and stored data algorithms by reducing their CPU time significantly, and furthermore it reduces the memory requirement substantially for stored matrix algorithm.

3 Proposed Algorithms

In the previous section we discussed the existence of a 90-10 relationship between the stage of algorithm and the merging distance. We also observed that initial iterations are very costly. In this section we propose algorithms using *partially overlapping partitioning* that exploits these properties. The following figure pictorially shows a single-dimensional partially overlapping partitioning approach.

Data is divided into p number of partitions or cells by dividing a dimension range (in this case A) into p number of smaller ranges. Each cell is sandwiched between two δ-regions which are typically much smaller than the cell. The first and the last cells have only one adjacent δ-region. Henceforth, by definition, a *cell* is inclusive of its adjacent δ-regions. Note that every δ-region is included in two cells. The value of δ can be mentioned as a percentage of the range of the attribute or as an absolute value. Using this data structure we propose two algorithms: *2-phase* and *nested*. In the rest of the section we use priority queue algorithm to explain although complexity analysis is done for other algorithms.



2-Phase Algorithm The basic idea is instead of creating priority queues for all data, now there will be priority queues for each cell separately. In each iteration closest pair is found for each cell, and from those the over-all closest pair is found. If the over-all closest pair has distance less than δ then they are merged and the priority queues of only the corresponding cell are updated. If any of the merging cluster centers or merged cluster center is in any δ-region then the priority queue of the affected cell is also updated. *This procedure is repeated until the merging distance is larger than δ.* This is the *phase 1* of the algorithm. In *phase 2* traditional priority queue algorithm is employed over the remaining clusters to complete the dendrogram. Note that the same δ is used to partition the data as well as to stop the algorithm. This way large number of small sized clusters are merged in the *phase 1* that uses partitioning and only small number of larger clusters are merged in *phase 2* that uses traditional algorithm. Partially overlapping partitions are employed earlier in [13, 17] in order to speed-up nearest neighbor search. In nearest neighbor algorithm δ is ideally set to a value slightly larger than the minimum distance between all points so that actual nearest neighbor is located in the same cell as the candidate point and optimum performance is obtained. But an ideal δ for clustering purposes is the distance corresponding to the point at which distance plot takes a sharp up-turn as shown in Figure 1. More on this is discussed later. Efficient nearest neighbor algorithm cannot benefit the priority queue clustering algorithm where nearest neighbors of each cluster is already in the top of each queue. Instead it requires to be maintained efficiently as the algorithm progresses. Nearest neighbor algorithms gain in time but not in memory, whereas, as will be evident soon, clustering algorithms gain significantly in both.

Table 1 exhibits the 2-phase partitioning algorithm using priority queues with time and memory complexities. Steps 1–10 is *phase 1* and 11-12 is *phase 2*. *Notations*: $|\delta|$ is number of points in a δ-region, p is number of cells, k' is the remaining number of clusters after *phase 1*. Complexity analysis is done by assuming all cells to be of equal size.

Complexity Analysis: The 2-phase algorithm in Table 1 uses priority queues

Algorithm	Time Complexity	Memory Complexity
Input: Data (N,M), p, δ		
Output: Dendrogram		
1. choose an attribute to partition the data	$O(1)$	
2. divide the data into p number of cells where each cell has two adjacent regions of width δ (except first and last cells)	$O(N)$	$O(N + p * \lvert\delta\rvert)$
3. Create priority queues P for each cell	$p * O((\frac{N}{p} + \lvert\delta\rvert)^2)$	$p * O((\frac{N}{p} + \lvert\delta\rvert)^2)$
4. while merging distance $< \delta$		
5. for each cell find the closest pair of clusters, C_1 and C_2	$p * O(\frac{n}{p} + \lvert\delta\rvert)$	
6. find the over-all closest pair	$O(p)$	
7. merge the over-all closest pair	$O(1)$	
8. update corresponding P	$O((\frac{n}{p} + \lvert\delta\rvert) \log(\frac{n}{p} + \lvert\delta\rvert))$	
9. if any of the pair or merged cluster in δ-region then update P of affected cell	$O((\frac{n}{p} + \lvert\delta\rvert) \log(\frac{n}{p} + \lvert\delta\rvert))$	
10. remove the duplicate copies of clusters in δ-regions, if any	$p * O(k')$	
11. cluster the remaining k' clusters	$O(k'^2 \log k')$	$O(k'^2)$
12. return dendrogram		
Over-All Complexity	$(N - k') * O((\frac{N}{p} + \lvert\delta\rvert) \log(\frac{N}{p} + \lvert\delta\rvert)) + O(k'^2 \log k')$	$p * O((\frac{N}{p} + \lvert\delta\rvert)^2)$ or $O(k'^2)$

Table 1. An example of 2-phase partitioning approach over the priority queue algorithm

first proposed in [2]. The original algorithm has an over-all time complexity $O(N^2 \log N)$ and memory complexity $O(N^2)$. On the other hand, the 2-phase algorithm has an over-all time complexity $(N - k') * O((\frac{N}{p} + |\delta|) \log(\frac{N}{p} + |\delta|)) + O(k'^2 \log k')$ assuming $\log(\frac{N}{p} + |\delta|)$ to be greater than p. Memory complexity is $p * O((\frac{N}{p} + |\delta|)^2)$ or $O(k'^2)$ whichever is larger. If $|\delta|$ and k' are small and if they are assumed negligible then the complexities simplify significantly giving time complexity $N * O(\frac{N}{p} \log \frac{N}{p})$, i.e. $O(\frac{N^2}{p} \log \frac{N}{p})$, and memory complexity $p * O(\frac{N^2}{p^2})$, i.e. $O(\frac{N^2}{p})$.

In Table 2 we give time and memory complexities of 'stored matrix' and 'stored data' types of algorithms for before and after simplification. Under 'stored matrix' category, in addition to 'priority queue' type there is 'similarity matrix' algorithm that uses a similarity matrix in place of priority queues.

Correctness: The overlapping partitioning data structure guarantees a correct dendrogram. Note that number of correct dendrogram can be more than one due to the existence, if any, of ties between merging distances.

Type of Algorithm	Traditional Algorithm	2-Phase Algorithm: Before* Simplification	2-Phase Algorithm: After* Simplification
	Time Complexity		
Stored Matrix			
– Similarity Matrix	$O(N^3)$	$(N-k') * p * O((\frac{N}{p}+\lvert\delta\rvert)^2)$ $+O(k'^2 \log k')$	$O(\frac{N^3}{p})$
– Priority Queues	$O(N^2 \log N)$	$(N-k') * O((\frac{N}{p}+\lvert\delta\rvert)*$ $\log(\frac{N}{p}+\lvert\delta\rvert)) + O(k'^2 \log k')$	$O(\frac{N^2}{p} \log \frac{N}{p})$
Stored Data	$O(\alpha * N^2)$	$(N-k') * O(\alpha * (\frac{N}{p}+\lvert\delta\rvert))$ $+O(\alpha * k'^2)$	$O(\alpha * \frac{N^2}{p})$ if $\alpha > p$ or, $O(N^2)$ if $p > \alpha$
	Memory Complexity		
Stored Matrix			
– Similarity Matrix	$O(N^2)$	$p * O((\frac{N}{p}+\lvert\delta\rvert)^2)$	$O(\frac{N^2}{p})$
– Priority Queues	$O(N^2)$	$p * O((\frac{N}{p}+\lvert\delta\rvert)^2)$	$O(\frac{N^2}{p})$
Stored Data	$O(N)$	$p * O(\frac{N}{p}+\lvert\delta\rvert)$	$O(N)$

Table 2. Time and Memory Complexity Comparison: * – Simplification is done by assuming $|\delta|$ and k' negligible.

Nested Algorithm Time taken by a 2-phase algorithm depends on the total time taken by *phase 1* and *2*. As *phase 2* is the traditional algorithm whose time complexity directly depends on the remaining number of clusters (k') after *phase 1*, for good performance *phase 1* should take small time and k' should be small as well. Experiments show that it may not be easy to detect the point at which the total time is the least. Experiments are conducted over DS2 data set (used earlier for distance plots in Figure 1) with N=3k in 2-D. Number of cells is fixed to 10 while δ varies from 0.25 to 9.0. Figure 2(A) shows the time taken by *phases 1, 2* and their total time. We redraw the distance plot of Figure 1(A) with some

(A) δ ***vs.*** ***Phase 1, Phase 2,*** **& Total time** ($p=10$) (B) **Iteration Number** ***vs.*** **The Closest Pair Distance** (C) **A sample follows the whole data**

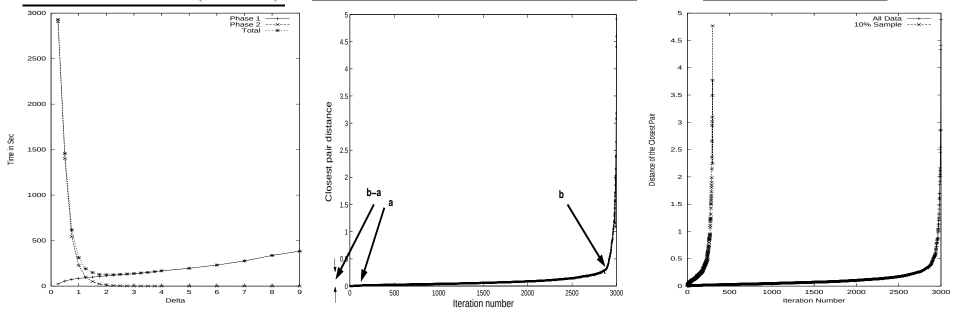


Fig. 2. Results and Analysis of Two-Phase Algorithm

labels in Figure 2(B). Label b points to a proper value of δ, label a points to the minimum merging distance. The reason why b is considered as an ideal δ value is that while on one hand it is very small, on the other hand it is able to capture most of the iterations leading to good performance. For δ smaller than b *phase 2* takes long time as remaining number of clusters is still large, whereas for δ larger than b *phase 1* takes more time than the optimal as larger δ means larger overlapping between cells. So we see that b is an ideal value for δ for efficient clustering. Contrast this with the ideal δ for nearest neighbor search methods discussed in [13, 14] that employs partially overlapping partitioning. Hence, we see that a is an ideal δ for nearest neighbor search while b is ideal for clustering.

It may not be easy to guess b value properly though. The following approach may be helpful. Take a random sample of points and obtain its distance plot. Figure 2(C) shows that a sample follows the whole data where the b values for the sample and the whole data almost match. The main reason is a sample will typically increase the smallest distance among *points* in a cluster but will retain the larger distances among the *clusters* themselves. This is also the reason why the sampling approach has been efficient for clustering. Manual observation can find a proper δ value. A window function to detect the point at which distance increases significantly can lead to automatic detection.

We propose a nested algorithm that does not require a manual setting of δ. It tries to make parameter setting even easier while gaining in performance in comparison to 2-phase algorithm. It is similar to the 2-phase algorithm except that it does not switch to *phase 2* immediately. Instead it performs *phase 1* with reduced number of clusters until a negligible number of clusters remain (eg. 2). The algorithm starts typically with a large value for p and a small value for δ. With each iteration of *phase 1*, p is gradually reduced while δ is increased. Arguably this is easier than to set δ to b from the start itself as in 2-phase algorithm. Nested partitioning attempts to divide the difference (b-a) (shown in Figure 2(B)) to gain in performance over 2-phase algorithm. More discussion on how to break the range (b-a) is given in the next section. *Complexity Analysis*: Let us assume that the nested sequence is specified by $<p_j, \delta_j>$ for j=1...s where p_j and δ_j are p and δ values in j^{th} nested iteration respectively, and s is the number of iterations in the nested sequence. Let n_j be the number of clusters remaining after nested iteration $j-1$. Initially n_1 is N and finally n_{s+1} is k'. Time complexity of nested algorithm is given as $\sum_{j=1}^{s}((n_j - n_{j+1})O((\frac{n_j}{p_j} + |\delta_j|)\log(\frac{n_j}{p_j} + |\delta_j|))) + O(k'^2 \log k')$. In this case k' is made negligible by specifying sufficient number of iterations in the nested sequence so that the last term $O(k'^2 \log k')$ can be discarded from the complexity. Memory complexity of nested algorithm is $\max_{j=1...s}(p_j * O((\frac{n_j}{p_j} + |\delta_j|)^2))$ because after each iteration of the nested sequence the priority queues are freed.

Overlapping Partitioning for Minimum Variance Ward's Method Partially overlapping partitions can be applied with some modification to Ward's minimum variance method [16]. Like centroid method, it represents a cluster by the centroid. But unlike centroid method, in each iteration the pair of clusters with the least increase in the sum of square error are merged. Square-error for cluster k is the sum of squared distances to the centroid for all points in cluster k. Mathematically, $e_k^2 = \sum_{i=1}^{n_k} \sum_{j=1}^{M} \left[x_{ij}^{(k)} - \mu_j^{(k)}\right]^2$ where $x_{ij}^{(k)}$ is the j^{th} dimension value of point x_i of centroid k, $\mu_j^{(k)}$ is the j^{th} dimension value of cluster k and $j = 1...M$. If cluster r and s merge in an iteration to create cluster t, then the change in square errors is given as: $\Delta E_{rs}^2 = e_t^2 - e_r^2 - e_s^2$. By replacing the expressions for square error and after simplification, we obtain that: $\Delta E_{rs}^2 = \frac{n_r n_s}{n_r + n_s} \sum_{j=1}^{M} [\mu_j^{(r)} - \mu_j^{(s)}]^2$. Note that the second term of the right hand side is the squared distance of the centroids of clusters r and s. The first term $\frac{n_r n_s}{n_r + n_s}$ can take values less than, equal to, or greater than 1. It is less than 1 when at least one of the clusters has size 1. This term makes direct application of the partially overlapping partitioning unsuitable although it shows 90-10 relationship. The reason is there is no guarantee that the closest pair is merged in each iteration as they may be in different cells. But note that the minimum value of this first term is $\frac{1}{2}$ for the case where size of both the merging clusters r and s is 1. The minimum possible ΔE_{rs}^2 with distance δ is $\frac{1}{2}\delta^2$. Hence to guarantee that the two clusters merged belong to at least one common partition the following condition must hold: *the ΔE_{rs}^2 of the closest pair r and s must be less than $\frac{1}{2}\delta^2$ so that distance between them will be less than δ.*

4 Performance

In this section we discuss different performance issues, and then perform experiments using the suggested parameter settings. *Data Sets* used for experiments include synthetic and benchmark data. Synthetic data sets are: DS2 used in BIRCH [18]; t4.8k, t5.8k, t8.8k, t7.10k used in CHAMELEON [10] and CURE [8]; SEQUOIA benchmark data [15] used in DBSCAN [4]. We generated DS2 data in different dimensions with uniform (DS2_U) and Gaussian (DS2_G) distribution within clusters . The rationale behind experimenting over these data is to compare the partitioning algorithm with traditional algorithm for different distribution of data and different shape of clusters in varying dimension and noise.

Performance Issues From the complexity analyses in the previous section, it is apparent that performance of the partitioning algorithm depends largely on number of cells p, and δ. Other important factors are (a) number (m) and choice of dimensions to partition, (b) how to change δ and p values in nested partitioning.

How Many Dimensions to Partition (m): It is affected by these factors: (1) the larger the m (possibly) the more uniform the distribution of points across cells leading to CPU time reduction, (2) given p and δ, the larger the m the smaller the number of δ regions leading to less CPU time and memory. These two factors indicate that larger m is preferable. But this may not be true as the larger the m the larger the number of cross-sections of δ-regions. As the number of cells required to update if a point is in the cross-section of m dimensions is 2^m, larger m will require more updating when a point is in cross-section. We conducted experiments by running phase 1 only for different m over DS2_U and DS2_G data sets, and for different p and δ. Our findings can be summarized as follows: multi-dimensional outperforms single-dimensional partitioning; the change in memory requirement is insignificant; for $m > 1$ the CPU time depends

on different factors but a noticeable thing is there is no consistent trend and it is very close in general.
Choice of Dimensions can affect performance. Some of the factors affecting it are variance, range and outliers. If the range is not affected by outliers, then attributes with larger range and variance are usually preferred. Other possible approaches include principal components analysis where data is projected to the first m principal components.
Setting δ & p for Nested Partitioning In nested partitioning a sequence of δ, p is given as the input. Experiments suggest to initialize δ to a very small value and then increment it gradually by small values until there remains negligible number of clusters. Value of p can be set by first setting a probable number (n') of clusters in each cell and then p is set to $\frac{N}{n'}$. Experiment with different data sets show that optimum performance is obtained for n' between 5 and 20.

Experimental Results Experiments are conducted on data sets described before. Nested algorithm is used where $n' = 7$ (i.e. $p = \frac{N}{7}$), number of dimensions for partitioning $m = 2$ each with $\lfloor\sqrt{p}\rfloor$ divisions, δ is initialized to 0.25% and is incremented by 0.25% for each nested iteration. Experiments are conducted to find speed-up and memory scale-down for varying N, M, and number of clusters.
Varying N: For this experiment N varies from 0.5k to 3k. As the traditional algorithm exceeds the memory allocated in our server we could not test for N larger than 3k. Figure 3(A) shows the speed-up of the nested multi-dimensional parti-

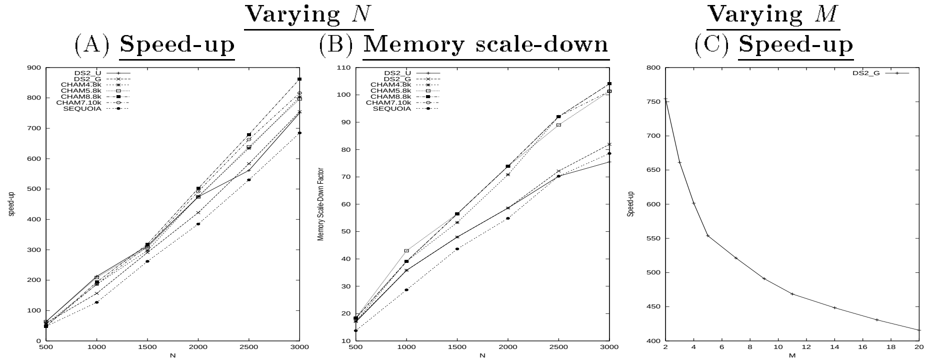


Fig. 3. Speed-up and Memory Scale-Down for 2-D data sets (A and B) and speed-up for varying M (C)

tioning algorithm over the traditional priority queue algorithm. The maximum speed-up is 862 for t8.8k with N=3k where traditional algorithm took 4717.7 CPU-sec while nested algorithm took only 5.47 CPU-sec. Note that speed-up factor increases with N. The reason is the complexity of j^{th} iteration of the nested sequence is given as $(n_j - n_{j+1})O((\frac{n_j}{p_j} + |\delta_j|)\log(\frac{n_j}{p_j} + |\delta_j|))$. Note that $\frac{n_j}{p_j}$ is set to 7 in our experiments. So the second term of the expression is mostly indifferent to increase in N except for the increase in $|\delta|$. In fact the time difference for partitioning algorithm for data set DS2_G for N=0.5k and 3k is less than 6 CPU-sec. But, as increase in N significantly affects CPU time of the traditional algorithm, hence the gain factor increases with N. Variation in speed-up for different data sets can be attributed to different distributions. Figure 3(B) shows the memory scale-down of the nested method over the traditional. For both algorithms the reported memory is the maximum during the complete duration of program where memory requirement of the programs themselves is negligible. The maximum scale-down is 105 for t8.8k data set where for N=3k, the traditional algorithm requires 385M and the nested algorithm requires only 5.1M. The increase in scale-down factor with N is due to the same reason as explained for CPU time speed-up. The variation in the scale-down factor among different data sets is more than that for the CPU time speed-up because the distribution has a direct effect over the memory requirement.

Varying M: Experiments are conducted over DS2_G data set with N=3k, M=2 to 20. Number of dimensions to partition m is set to 2 for all experiments. As seen in Figure 3(C) the gain factor for CPU time reduces with increasing M for the following reason. Increasing M does not affect the traditional algorithm much because the dominant time factor of maintaining the heap-based priority queues due to the insertions and deletions is not affected by increasing M. Hence, the small increase in time due to the larger M for nested algorithm reduces the gain factor, while on the other hand, memory complexity is affected very little as priority queues are the dominant factor which do not change with dimensions.

Varying Number of Clusters: DS2_G data with N=3k in 2-D is used. Number of clusters varies from 5 to 100 with approximately equal sized clusters. Experiments are conducted by fixing and varying N. For varying N, reduction in the number of clusters decreases N which reduces the speed-up factor. This result is similar to the results in Figure 3(A). When N is fixed the speed-up factor reduces as well with decrease in number of clusters because each cluster contains more points leading to more points in each cell. For this reason the memory requirement increases with decrease in number of clusters.

Minimum Variance Ward's Method CPU time speed-up factor is measured for partitioning algorithm over traditional algorithm for minimum variance measure. For DS2_G data set with $N = 3k$ and $M = 2$ the gain factor is 835; for $M = 10$ the gain factor is 532; for $M = 20$ it is 460.

5 Conclusion and Future Work

In this paper we studied an important behavior of geometric metric hierarchical clustering algorithms that show a 90-10 relationship where roughly 90% or so iterations from the beginning merge clusters that are separated by less than 10% or so of the maximum merging distance. We proposed two algorithms using partially overlapping partitions that exploit this behavior. These algorithms are suitable for both stored data and stored matrix type of algorithms. Complexity analysis and experiments show significant reduction in time and memory. It is found that gain factor increases with increase in number of points in the data. Increase in dimensions reduces the gain factor but still the gain is significant for high-dimensional data. We introduced a modified method to apply this partitioning technique to minimum variance Ward's method.

A future work is to parallelize this technique. In [12] parallel algorithms employ naive partitioning where all N priority queues are distributed among various processors. But partially overlapping partitioning is foreseen to be very suitable for parallelization as a processor will be in charge of priority queues for clusters in only one or more number of cells which should reduce memory and CPU time. Other future work includes testing the suitability of other more sophisticated partitioning [3].

References

1. M. R. Anderberg. *Cluster Analysis for Applications.* Academic Press, NY, 1973.
2. W. H. E. Day and H. Edlesbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, 1(1):7–24, 1984.
3. W. DuMouchel, C. Volinsky, T. Johnson, C. Cortes, and D. Pregibon. Squashing flat files flatter. In *Proceedings of KDD'99*, pages 6–15, 1999.
4. M. Ester, H. P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of KDD'96*, pages 226–231, 1996.
5. U. Fayyad, C. Reina, and P. S. Bradley. Initialization of iterative refinement clustering algorithms. In *Proceedings of KDD'98*, pages 194–198, 1998.
6. K. Florek, J. Lukaszewicz, J. Perkal, H. Steinhaus, and S. Zubrzycki. Sur la liason et la division des points d'un ensemble fini. *Colloq. Math.*, 2:282–285, 1951.
7. S. Guha, R. Rastogi, and S. Kyuseok. ROCK: A robust clustering algorithm for categorical attributes. In *Proceedings of ICDE'99*, pages 512–521, 1999.
8. S. Guha, R. Rastogi, and K. Shim. CURE: An efficient clustering algorithm for large databases. In *Proceedings of ACM SIGMOD'98*, pages 73–84, 1998.
9. A. K. Jain and R. C. Dubes. *Algorithm for Clustering Data*, chapter Clustering Methods and Algorithms. Prentice-Hall Advanced Reference Series, 1988.
10. G. Karypis, E-H. Han, and V. Kumar. CHAMELEON: A hierarchical clustering algorithm using dynamic modeling. *IEEE Computer*, 32:68–75, 1999.
11. F. Murtagh. A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26:354–359, 1983.
12. C. F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21:1313–1325, 1995.
13. M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity*, pages 21–39. Academic Press, New York, 1976.
14. F. J. Rohlf. Computation efficiency of agglomerative clustering algorithms. Technical Report Report RC 6831, IBM T. J. Watson Research Center, NY, 1977.
15. M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The SEQUOIA 2000 storage benchmark. In *Proceedings of ACM SIGMOD*, pages 2–11, 1993.
16. Jr. J. H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58:236–244, 1963.
17. G. Yuval. Finding nearest neighbors. *Information Processing Letters*, 5:63–65, 1976.
18. T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *Proceedings of ACM SIGMOD*, pages 103–114, 1996.