C-1

Efficient Algorithms for k-Separation Clustering

Teofilo F. Gonzalez
Department of Computer Science
University of California
Santa Barbara, CA, 93106
teo@cs.ucsb.edu

Abstract

The k-separation problem is to partition a set of n objects into k sets or clusters so that the dissimilarity (distance or weight) between the two most similar objects belonging to different clusters is as large as possible. Solutions to this clustering problem have been used to construct dynamic Internet distance maps and for the efficient dissemination of mobile agents. The current best algorithm for this problem has time complexity $O(n^2 \log n)$. In this paper we present several algorithms with running times $O(e+n^2)$ and $O(e \log n)$ to solve the k-separation problem, where e is the total number of measured dissimilarities between the objects. In the applications mention above e is about n^2, so the best of our algorithms takes linear time. The fastest of our algorithms has a small constant associated with its time complexity bound.

Keywords: Clustering, Algorithms, k-Separation, Polynomial Time.

1 Introduction

Theilmann and Rothermel [6, 7] introduced a new clustering criteria, called *max k-separation*, which they use to speed-up communication through the Internet. More specifically, they use solutions to the clustering problem to construct dynamic Internet distance maps [6] and for the disseminate efficiently mobile agents [7]. The max k-separation problem consists of

partitioning a set of n objects into k sets so that the dissimilarity (distance or weight) between the two most similar objects belonging to different clusters is as large as possible. Formally, we have a graph $G = (V, E, W)$ where V is the set of n vertices, E is the set of e edges, and the weight function $W : \{i, j\} \rightarrow R^+$ assigns a positive weight to each edge in the graph. The weight or distance associated with each edge represents the dissimilarity between the two objects the edge joins. When the graph is complete and the weights satisfy the triangle inequality, i.e., for every three vertices i, j, and k, $W(\{i, j\}) \leq W(\{i, k\}) + W(\{k, j\})$, the graph is said to be in *metric space*. The problem instances that arise in the applications mentioned above are in metric space.

There are three closely related problems: the k-maxcut problem (maximize the sum of inter-cluster weights), the k-min cluster problem (minimize the sum of the inner-cluster weights) and the k-gMM clustering problem (minimize the maximum inner-cluster weight). Under this notation the max k-separation problem is to maximize the minimum inter-cluster distance. Therefore, the k-separation problem is to the k-maxcut problem as the k-gMM clustering problem is to the k-min cluster problem. The k-maxcut problem and the k-min cluster problem are computationally identical problems in the sense that an optimal solution for one of these problems is an optimal solution to the other one. However, the k-gMM and the k-separation problems do not share this type of relationship. The k-separation clustering problem can be solved in polynomial time, but the other three clustering problems have been shown to be NP-hard. Theilmann and Rothermel [6, 7] developed an algorithm that takes $O(n^2 \log n)$ time, where n is he number of objects, for the max k-separation problem.

Sahni and Gonzalez [8] developed a linear time approximation algorithm for the k-maxcut problem that generates a clustering with objective function value at least equal to $(k-1)/k$ times the objective function value of an optimal solution. Clearly, for large k the clustering generated by the algorithm are almost optimal. When the set of points is in metric space and clusters are required to have the same number of objects, Gonzalez and Murayama [2], and subsequently Guttmann-Beck and Hassin [4], showed that any (balanced) partition has objective function value at least equal to $(k-1)/(k+1)$ times that of a (balanced) optimal partition. Sahni and Gonzalez [8] showed that for $k \geq 3$ the k-min cluster problem cannot be approximated in polynomial time for any constant unless P = NP, i.e., if there is a

polynomial time algorithm that generates a clusterings with objective function value within c times the optimal solution value, then P = NP. Guttmann-Beck and Hassin [3] showed that in metric space the problem can be approximated in $O(n^{k+1})$ to within a factor of two, i.e., the objective function value of the solution generated by the algorithm is within twice the optimal one. The k-gMM for arbitrarily graphs cannot be approximated within any constant unless P=NP [1], but in metric space the problem is called the k-tMM problem, and it can be approximated within two times the optimal solution value in $O(nk)$ time, but approximating it within $2 - \epsilon$ is an NP-hard problem, for every $\epsilon > 0$.

Theilmann and Rothermel [6, 7] evaluated the performance of their clustering algorithm and Gonzalez' algorithm [1] for their applications. Both algorithms have advantages and disadvantages, but the one for the k-separation problem outperforms the other one in their experimentation. As pointed out in [6, 7] the k-separation problem may lead to clusterings that are highly unbalanced with respect to the number of points in the clusters, but optimal clusters for the k-tMM problem may assign to different clusters objects that are close to each other. Algorithms for other clustering problems are discussed in [1, 2, 3, 4, 6, 7].

1.1 Max k-Separation

Theilmann and Rothermel's algorithm [6, 7] generates a clustering by a greedy procedure similar to Kruskal's algorithm for constructing a minimum cost spanning tree [5]. The algorithm begins by sorting and then considering the edges in increasing (actually non-decreasing) order of the their weights. When selecting edge $\{u, v\}$ vertices u and v are merged into one vertex. The weight of the edge between the new vertex and each vertex x is defined as $\min\{W\{u, x\}, W\{v, x\}\}$. The process terminates when there are k vertices, each of which represents the set of objects in each cluster. For completeness the procedure is given below using a notation that is slightly different for the one in [6, 7].

```
Procedure TR(G = (V, E, W), k);
Vertex v_i is said to represent vertex i;
Sort the edges in G in non-decreasing order with respect to their weight;
while | V |> k do
  select edge (v_1, v_2) ∈ G with minimum weight;
  // merge nodes v_1 and v_2 into v_1 and adjust the edge weights. //
  for every v ∈ V − {v_1, v_2} do
    w({v_1, v}) = min {W({v, v_1}), W({v, v_2})}
  endfor
  Hereafter vertex v_1 will also represent the vertices that vertex v_2 represents;
  Remove edges incident to v_2 in E;
  V ← V − {v_2};
endwhile
// The resulting k vertices represent the k clusters.//
end of TR;
```

As pointed out in [6, 7], it is not difficult to establish that procedure TR generates an optimal solution to the Max k-Separation problem. With respect to the running time, the sorting of the edges requires $O(e \log n)$ time and the while-loop is executed $n-k$ times, each time requiring $O(n)$ time provided the data is represented by an appropriate structure. For brevity we will not elaborate on the appropriate data structures for this algorithm, but claim that the overall time complexity is $O(e \log n + n^2)$. For problems in metric space the time complexity is $O(n^2 \log n)$ since the graph is complete. These results are summarized in the following lemma whose proof appears in [6, 7].

Lemma 1.1 ([6, 7]) *Procedure* TR *generates an optimal clustering for the Max k-Separation problem and can be easily implemented to take $O(e \log n + n^2)$ time.*

Another way to implement the above strategy is to take all the edges and add them to a min-heap instead of sorting them first. Then at each iteration one deletes the min-weight edge in the heap and joins the two vertices such edge joins (say v_1 and v_2). Then one

needs to update the weights of some of the edges emanating from v_1 and delete the ones emanating from v_2. The overall time complexity remains the same as with the previous implementation. The running time reduces to $O(e \log n)$ if one delays the deletion and update operation inside the while-loop. Lets call this new algorithm TR-K. The idea is to use the classic implementation of Kruskal's algorithm, for the minimum cost spanning tree problem, in which at each iteration we test whether the minimum weight edge deleted from the min-heap joins vertices in the same component (cluster). If so, the edge is tossed away, otherwise both clusters (components) are merged into a single one. The while-loop can be implemented to take $O(\log n)$ by using an efficient implementation for the disjoint union-find operations used to test whether two objects belong to the same connected component (same spanning tree in the spanning forest). We do not discuss more details about the implementation because they are similar to the ones for Kruskal's algorithm [5]. This result is formalized in the following lemma.

Lemma 1.2 *Procedure* TR-K *generates an optimal clustering for the Max k-Separation problem and it can be easily implemented to take $O(e \log n)$ time.*

Proof: By the above discussion.

□

Let us offer a simple characterization of the objective function value of an optimal solution to the Max k-Separation clustering problem.

Fact 1.1 *The objective function value of an optimal solution to the Max k-Separation problem is the weight of an edge, call it WT, such that the number of connected components in the graph $G_{<WT}(V, \{\text{set of edges with weight} < WT \text{ in } G\})$ is more than k, but the number of connected components in the graph $G_{\leq WT}(V, \{\text{set of edges with weight} \leq WT \text{ in } G\})$ is at most k.*

An approach that is asymptotically faster based on Fact 1.1 can be easily developed. Let us assume that the weights of all the edges are different. Later on we describe how the procedure can be easily modified to solve the more general problem. First we find the median weight of the edges. Let us refer to the median weight by MW. Now consider the

graph with only those edges whose weight is less than or equal to MW (call it G_{MW}) and find the connected components of G_{MW}. Let $k_{G_{MW}}$ the the number of such components. If $k = k_{G_{MW}}$ the algorithm terminates at this point and each cluster is simply one of the connected components. If $k > k_{G_{MW}}$ then invoke the procedure with the graph G_{MW}. Otherwise, if $k < k_{G_{MW}}$, then we combine each component into a single vertex and invoke the procedure using the resulting graph. Each vertex representing a component is used to represent all the vertices in the component. The specifics of the algorithm are given below.

`Algorithm Div-and-Conq` $(G = (V, E, W), k)$
Vertex v_i is said to represent vertex i;
$MW \leftarrow$ median weight of an edge in E;
$G_{MW} \leftarrow G -$ { edges with weight greater than MW};
Let $k_{G_{MW}}$ be the number of connected components in G_{MW};
case
:$k > k_{G_{MW}}$: `Div-and-Conq` (G_{MW}, k);
:$k = k_{G_{MW}}$: All the vertices represented by the vertices in each connected component
in G_{MW} form a cluster;
return;
:$k < k_{G_{MW}}$: Combine each vertex in the same component into a single vertex in G'.
Hereafter the new vertices represent all the vertices represented by the vertices
in the component;
Call the resulting graph G';
`Div-and-Conq` (G', k);
endcase
`End of Div-and-Conq`;

Using Fact 1.1 it is simple to establish that procedure `Div-and-Conq` generates an optimal solution to the Max k-Separation problem. With respect to the running time, finding the connected components of a graph takes $O(n+e)$ time, where n is the number of vertices in the graph and e is the number of edges by using depth-first search. Shrinking each connected component into a single vertex can be easily done in $O(n + e)$ time. If there is another

invocation of procedure `Div-and-Conq` it will involve a graph with no more than half of the edges in the current graph. Therefore at the ith recursive there are no more than $c_1 e/2^{i-1}$ edges, i.e., the first invocation there are $c_1 e$ edges, the second at most $c_1 e/2$ edges, the third $c_1 e/4$ and so on. Using these observations and a couple of other facts, one can establish that the time complexity bound for the proposed procedure is just $O(e)$. These results are summarized in the following lemma.

Lemma 1.3 *Procedure* `Div-and-Conq` *generates an optimal clustering for the Max k-Separation problem and can be easily implemented to take $O(e)$ time.*

Proof: By the above discussion.

□

For the case when some edges have identical weights, the graph G_{MW} consists of exactly half of the edges in E and contains all the edges with weight at most MW and some of the edges with weight equal to MW. A procedure based on this idea can be shown to generate an optimal solution to the k-Separation problem within the same time complexity bound as procedure `Div-and-Conq`.

How efficient is this new algorithm in practice for large values of n? We are not so sure, but we feel it might not be as fast in practice as the one developed by Theilmann and Rothermel [6, 7]. There are many reasons for this, one is that we there are too many invocations to the linear time procedure that finds the median of a set of numbers, which for huge data sets makes a large number of I/O operations.

Let us examine an algorithm that is faster in practice when the graph is in metric space. The idea behind the algorithm is to keep for each vertex i an array $adj_i[j]$ of distances to the other vertices and have the smallest of all these distances stored in $min[i]$. All the $min[i]$ elements are stored in a min-heap. So initially every vertex is a cluster by itself and at each iteration two clusters are combined into a single one. The ones combined are the two "closest" ones, i.e., the weight of the least-weight edge between the clusters combined is least possible. Each time one needs to update the $adj_i[.]$ array, an entry for $min[i]$ and the heap. When cluster j is merged into cluster i we set $adj_i[j] = \infty$ so that it will never be selected again, since these two clusters have been merged into one. The array $Tree[i]$ indicates for each object i the connected component (or cluster or tree) where it belongs.

Procedure `Fast-Separation` $(G \leftarrow (V, E, W), k)$

for $i = 1$ **to** n **do**;

$tree[i] \leftarrow i$;

for $j = 1$ **to** n **do**;

if $i = j$ **then** $adj_i[j] \leftarrow W(i, j)$;

else $adj_i[j] \leftarrow \infty$;

endfor;

$min[i] \leftarrow$ smallest value in $adj_i[.]$ and

$index[i]$ is the value of j such that $adj_i[j]$ is the smallest value in $adj_i[.]$

endfor;

Create a min-heap of tuples $(i, index[i], min[i])$ ordered with respect to 3rd component;

for $l = 1$ **to** $n - k$ **do**;

$(i, j, val) \leftarrow$ **DeleteMin** (H);

$(i', j', val') \leftarrow$ **DeleteMin** (H);

//Note that $tree[i] = tree[j']$, $tree[j] = tree[i']$ and $val = val'$//

//because all the weights of the edges are distinct.//

for $q = 1$ **to** n **do**;

if $tree[q] = tree[j]$ **then** $tree[q] \leftarrow tree[i]$;

endfor;

for $q = 1$ **to** n **do**;

if $tree[i] = tree[j]$ **then** $adj_i[q] \leftarrow \infty$;

$adj_i[q] \leftarrow$ **min**$\{adj_i[q], adj_j[q]\}$

endfor;

$min[i] \leftarrow$ smallest value in $adj_i[.]$

$index[i]$ is the value of j such that $adj_i[j]$ is the smallest value in $adj_i[.]$

add to the heap the tuple $(i, index[i], adj_i[index[i]])$

endfor;

//All the vertices with the same $tree[]$ value form a cluster.//

`End of Fast-Separation`;

The time complexity for the above procedure is $O(n^2)$. The initialization part (first i-loop) clearly takes $O(n^2)$ because each line inside it takes $O(n)$ time and each line inside the two for-loops takes $O(c)$ for some constant c. The main loop in the remaining part is executed $O(n)$ times. One can easily establish that, as in the previous case, each line inside the outter for-loop takes $O(n)$ time and each line inside the inner for-loop takes $O(c)$ for some constant c. Therefore the overall time complexity is $O(n^2)$ time. When the graph is complete, the algorithm takes linear time with respect to the number of edges. But if there are $O(n)$ edges, then the time complexity will be quadratic with respect to the input length. When the number of edges is $\Omega(n^2)$ procedure `Fast-Separation` is faster than the `Div-and-Conq` algorithm simply because one does not need to invoke the median-finding algorithm and all the structures used by the algorithm are quite simple. Procedure `Fast-Separation` outperforms the `TR` algorithm because it performs fewer of the same type of operations. Our result is summarized in the following theorem.

Lemma 1.4 *Procedure* `Fast-Separation` *generates an optimal clustering for the Max k-Separation problem and can be easily implemented to take $O(n^2)$ time.*

Proof: By the above discussion.

□

The algorithm can be easily modified to handle the case when some edges have identical weights. The idea is at each iteration to delete edges until an edge which has not yet been deleted is found (remember that edges are added twice). This new procedure has the same running time and generates an optimal solution.

1.2 Discussion

We have presented algorithms with running times $O(e + n^2)$ and $O(e \log n)$ to solve the k-separation problem. This problem arises in the construction of dynamic Internet distance maps and in the efficient dissemination of mobile agents in which the objects are in metric space. For these applications our algorithm has a linear time complexity bound, rather than $O(n^2 \log n)$ as in the previous best algorithm. Procedure Fast-Separation outperforms the `TR` algorithm because it performs fewer of the same type of operations.

References

[1] Gonzalez, T. F., "Clustering to Minimize the Maximum Inter–Cluster Distance," *Journal of Theoretical Computer Science,* No. 38, October 1985, pp. 293 – 306.

[2] Gonzalez, T. F. and T. Murayama, "Algorithms for a Class of Min–Cut and Max–Cut Problems," Proceedings of the Third Annual International Symposium on Algorithms and Computation, Lecture Notes in Computer Science #650, Springer–Verlang, December 1992, pp. 97 – 105.

[3] Guttmann-Beck, N, and R. Hassin, "Approximation Algorithms for Min-Sum P-Clustering" *Discrete Applied Mathematics,* Vol. 89, No 1–3, pp125 – 142, 1998.

[4] Guttmann-Beck, N, and R. Hassin, "Approximation Algorithms for Minimum K-Cut" *Algorithmica,* Vol. 27, pp 198 – 207, 2000.

[5] Horowitz, E., S. Sahni, and S. Rajasekaran, "Computer Algorithms in C++," Computer Science Press, Inc., 1997.

[6] Theilmann, W. and K. Rothermel, "Efficient Dissemination of Mobil Agents," Proc. 19th IEEE Int. Conf. on Distributed Computing Systems Workshop on Web Based Applications, IEEE Press, pp. 9 - 14, 1999.

[7] Theilmann, W. and K. Rothermel, "Dynamic Distance Maps of the Internet," Proceedings of the 2000 IEEE INFOCOM Conference,

[8] Sahni, S. and T. Gonzalez, "P-Complete Approximation Problems," *Journal of the ACM*, 23, 555-565, 1976.