An Efficient Implementation of the Robust k-Center Clustering Problem

Rachel Schwartz
University of Maryland
College Park, Maryland 20742, United States

April 29, 2010

Abstract

The standard k-center clustering problem is very sensitive to outliers. Charikar et al. proposed an alternative algorithm to cluster p points out of n total, thereby avoiding the distortion caused by outliers. The algorithm has an approximation bound of three times the true solution, but is very slow if implemented naively. We propose a modified implementation of the algorithm that runs significantly faster than the standard version. It does this while keeping the memory bound within the same asymptotic bound as that of the naive implementation. We show that, as the size of the problem increases, our algorithm maintains a relatively low running time, while the standard implementation time increases in proportion to it.

1 Introduction

Clustering problems arise almost everywhere in computer science, from systems applications to artificial intelligence research to statistical analysis; the question of how to best categorize points into distinct groups based on distance is therefore a very well studied one. One standard variant of the problem is k-center clustering: the NP-complete challenge of finding k centers from a set of n points, minimizing the maximum distance for each point from its chosen center. The problem has a widely used approximation algorithm, the k-means algorithm, which performs well under most circumstances.

While the standard k-center clustering optimization often provides a good framework for analyzing a set of points, it does not adequately describe the goal in every clustering problem. One drawback of k-center is its sensitivity to every one of the n points in the set. Since the goal of k-center clustering is to keep the maximum distance of a point from its nearest center low, outlying points need to be carefully considered so that they do not inflate the solution radius. This may mean choosing a center especially for these outlying points, which

may be close to very few of all n points. If the goal of the clustering is for every point to be within a given radius of a center, this over-emphasis of the outliers cannot be helped. However, for many problems, the requirement that every point be within some distance of a center is overly stringent. In such a case, the skewing of the centers towards the outliers produces an unnecessarily sub-optimal solution. To obtain the optimal answer, the problem statement must be modified: find k centers from n points such that for p points of the n in the set, the maximum distance for each point from its nearest center is minimized. $(n - p)$ points are then considered outliers, and may be arbitrarily far from a center without affecting the solution.

Charikar, et al. introduce an approximation algorithm for solving this version of k-center clustering, known as the robust k-center clustering problem [2]. Briefly, the algorithm relies on a binary search for the solution from among all possible radii (consisting of distances between every pair of points). For each possible radius tried, a series of discs and expanded discs is constructed. Each point's disc is composed of points within that radius and each point's expanded disc is composed of points within three times that radius. For k iterations, the largest disc is chosen as a center and all points in its expanded disc are marked as covered and removed from the discs. After k iterations, if no more than $(n - p)$ points remain uncovered, a solution has been found for this radius. If not, no solution exists for this radius. The binary search continues until the smallest radius with a solution is found. At that point, the solution is guaranteed to be within three times the optimal radius.

The Charikar et al. algorithm, while it provides a good approximation of an NP complete problem, can be untenably slow if implemented naively. Disc generation can require checking the distance between every pair of points, resulting in $O(n^2)$ time for each solution search. Updating the discs can be even slower, with each point in every disc checked to see if it is a newly covered point; this must be repeated k times for each solution search. There is also a large memory cost associated with storing the discs explicitly. For the algorithm to be a really useful one, it must be capable of computing solutions for large numbers of points. It is therefore necessary to devise a more clever approach of implementing the algorithm. In this paper, we present an implementation that considerably reduces the necessary computing time. We also implement a memory-efficient version, though without any optimizations to reduce computing time.

The rest of the paper is organized as follows. Section 2 contains related work. Section 3 discusses the algorithm and our implementation in detail. Section 4 contains results for the naive and improved implementations while Section 5 concludes the paper.

2 Related work

The general area of clustering is a very well studied one; the specific method of clustering using approximations algorithms has been much explored as well. Shmoys et al. were the first to present a constant time approximation algorithm for the uncapacitated facilities location problem, the most general version of the clustering problem; Guha and Khuller improved on the algorithm with a better approximation bound [9] [4]. A series of methods reduced this constant further, concluding with the Mahdian et al. approximation constant of 1.52 [8].

There is also much research on clustering methods that are robust to outliers, with many different approaches suggested. Baduoi et al. presented a $(1 + \epsilon)$ approximation algorithm for clustering via coreset, and extended their method to allow for outliers [1]. Guha et al. developed a hierarchical clustering method, aimed at large datasets, which is more robust to outliers than standard methods [5]. For streaming algorithms, Charikar et al. developed a constant guarantee approximation for k-center clustering using random sampling [3]. A few methods took a two-phased approach, separating outlier detection and clustering. Jiang et al. used a modified version of the k-means algorithm and then built a minimum spanning tree to detect likely outliers [7]. Hautamaki et al. used a defined *outlyingness factor* to iteratively select clusters that are likely to be outliers [6]. The Charikar method differs from all of the above in being a general purpose single step k-center clustering algorithm using partitional clustering.

3 The Algorithm

3.1 Basic Algorithm

The Charikar et al. algorithm consists of a binary search from a series of possible solutions. Since the maximum radius must be the distance between two points, the solution must be one of those $O(n^2)$ distances. Given a radius r, the algorithm can determine whether there is a solution to the k-center problem with that radius. First, the points within a distance r of each point are assigned as that point's disc and those within $3r$ assigned to its expanded disc. The point with the largest disc is then chosen as the first point of the k in the solution and the points in the expanded disc are marked as covered and removed from all discs. This is repeated k times, with the largest disc chosen as the new point added to the solution. After k repetitions, there are k points in the solution and computation stops. If at least p points are covered, a solution has been found. A binary search from among all possible distances will find the smallest r for which a solution can be found. This solution will be within three times the minimum radius possible.

If naively implemented, the algorithm can be a very expensive one. To generate the discs, each point must be compared with every other, an $O(n^2)$ process. Each of the k iterations requires searching every disc for newly covered points,

an $O(n^2)$ process as well, though practically, it will often be far less expensive. Since a binary search is done to find the minimum solution radius, the total cost is $O(n^2 \log(n^2)) = O(n^2 \log n)$. The memory cost of the solution is heavy as well. At a minimum, each disc/expanded disc must be stored, which is an $O(n^2)$ proposition.

3.2 Fast Disc Generation

The naive method of disc generation necessitates a comparison between every pair of points in the problem. Since in the majority of cases, most of the points will not fall within each other's disc radii, most of these comparisons are superfluous. If distances are known in advance, however, disc generation becomes a trivial task. Obviously, it is not possible to know which points are neighbors initially; however, in order to obtain a list of distances for a binary search, every set of points must be compared at least once. This comparison can be utilized to find neighbors for disc generation purposes. If every pair of points is stored in order of distance, disc generation can be accomplished simply by traversing the list until distances beyond the current radius are reached. To obtain this list, the extra computation cost is an $O(n^2 \log n^2) = O(n^2 \log n)$ sorting cost for this list of pairs, which may not be too painful. However, the extra memory cost is prohibitive: storing every pair of points requires $O(n^2)$ space. Although this is actually the same bound required by the discs themselves, practically, the discs will require much less, while the pairs cost is a high one, independent of the points' layout.

Since for solution searches, the only interesting pairs are those whose distance falls below a given radius, the pair storage can be greatly reduced. In our implementation, a configurable constant is chosen as the bound of pairs stored, where the number of pairs is capped at the constant times the number of points. Alternatively, some radius could be used as the cutoff point, but that raises the question of how a radius is chosen. Additionally, that method would not guarantee an $O(n)$ storage cost, while our method does this naturally. As the pairs are generated, it becomes necessary to determine whether each pair should be stored. Our method of doing so relies on a max heap capped in size at the storage limit. Each new pair is checked against the top of the heap to determine whether its distance is smaller than the current largest distance stored. Checking whether a pair needs to be stored is then a constant operation, although removal and replacement of a pair requires an additional $\log(n)$ cost for each point. This does introduce an extra cost into the distance list/pairs generation, with an $O(n^2 \log n)$ cost instead of the $O(n^2)$ cost in the naive implementation. There is an additional $O(n \log n)$ cost for sorting the pairs into order of distance once the list is generated.

Although limiting the size of the pairs list greatly reduces the memory cost of the implementation, some solution searches will require a radius larger than the maximum distance stored in the list. Without some additional method for gen-

erating discs, it would be necessary to fall back on the naive method, so that no savings at all could be acquired from the pairs list. Our implementation compensates for this deficit with the use of *special points*. A number of points are randomly chosen as special points, and their distances from every other point are stored in order. These points' distances can then be used as approximations when looking for points under a given distance from a point near a special one.

For the disc of a point p, we are actually interested in points p_2 that are at a distance $\leq r$ from p. Since p is a distance of r_{sp} from some special point, points p_2 are at most a distance of $r_{sp} + r$ from the special point. The special point's distance list can therefore be searched for points in p's disc, stopping when the distance reaches $r + r_{sp}$. In addition, the points very near in distance to p need not be searched either, as they will be on the pairs list. Points that are a distance of more than the maximum pairs list radius r_m from p will be at least $r_m - r_{sp}$ from the special point. If a point was a distance less than $r_m - r_{sp}$ from the special point, it would be at most $r_m - r_{sp} + r_{sp} = r_m$ from p. In that case it would be on the pairs list and would be added to the disc using that method. The section of the special points distance list that needs to be traversed in generating a point p's disc is for points of distances between $(r_m - r_{sp})$ and $(r + r_{sp})$. Of course, because the expanded disc must be built as well, the actual upper limit becomes $(3r + r_{sp})$.

The significance of the savings depends on the value of r_{sp}: the closer this value is to 0, the smaller the range of points that needs to be searched for disc generation. Of the special points, the one closest to the center point of the current disc is used as an approximation. A larger number of special points will decrease the average distance r_{sp}. However, each special point requires a full list of point distances stored, resulting in an extra $O(n)$ storage cost for each point. Of course, as long as the number of special points is held constant, the storage cost will stay at $O(n)$, but the number of points chosen must be balanced against the significant cost for each extra point. This number is therefore configurable in the implementation.

3.3 Fast Disc Updating

The disc generation phase of the algorithm is expensive, but, for the naive implementation, the solution search cost exceeds it. Once a disc is chosen to be added to the solution, all its points are marked as covered and removed from all discs. In the basic implementation, this requires traversing each disc and removing any covered points. Of course, many discs will have no covered points at all, as long as they do not overlap with the newly chosen solution disc.

Our implementation utilizes a very simple property of metrics to avoid this extra disc search. Since distances are symmetrical, any point in the disc of another point will contain that other point in its disc. The only discs that may contain newly covered points are those discs with centers that are contained in the discs

of the newly covered points. Furthermore, many of those discs will have centers that are covered themselves; these discs do not need to be searched for newly covered points at all, since they will shortly be removed from the solution. By searching the discs of all the newly covered points, i.e. those that are covered by the point newly added to the solution, we can obtain a list of all the discs that need to be "cleaned up" - have covered points removed. These discs will, in general, be only a subset of the entire disc set that must be searched in the naive implementation. The computational cost of updating the discs is significantly lessened, while no extra memory is required.

The final step for each of the k iterations of the disc update is finding the new biggest disc. This is an $O(n)$ computation, upon which we did not seek to approve in our implementation.

3.4 Memory Considerations

Although the efficient algorithm offers a speed improvement, it actually uses even more memory than the naive implementation. This is not practical for very large numbers of points. We also implement a memory efficient solution method, which does not seek to optimize the computational time. This algorithm does not store discs explicitly at all. Instead, for each of the k iterations of every radius checked for a solution, the number of points in each point's radius is recalculated. This means the computation time basically consists of disc generation time, but that generation happens more frequently.

4 Results

All three implementations, naive, efficient and memory efficient, were written in Java. All tests were run on a 32-bit Windows machine with a 32 bit JVM. The machine used had 4G of RAM and a 3.06 Ghz duel core processor. The JVM was run with a maximum heap size of 1500 MB, which was the limit for this machine.

4.1 Disk Generation Constants Study

Before running the main comparison of the algorithms, we analyzed the effect of the two configurable parameters on the solution time. Both parameters, the pairs list constant and the number of special points, effect only the disc generation time, so only that time is noted here. We used a small number of points so that we could observe the advantage gained from the changing constants without considering the effects of memory shortages. Points were two dimensional Euclidean points distributed in a 500x500 area. We examined two different point layouts, one a completely random generation and one where points were generated uniformly around k clusters, with random outliers. For each distribution, we ran the algorithm for 645 points, with $k = 15$. Up to 30

points were allowed as outliers. The two constants were both tested with values of $\{10, 25, 50, 60, 75\}$. For each combination of values, we ran the algorithm three times each for both the random and centered distributions. The random distribution was slower than the centered one, but they seem to exhibit the same trends. In general, we observed that, unsurprisingly, as the number of special points and number of pairs stored went up, the overall disc generation time went down. Also unsurprisingly, the more dramatic reductions in time came with the earlier increases in the constants. These numbers confirm the general efficacy of the improved implementation in reducing computational time for the solution.

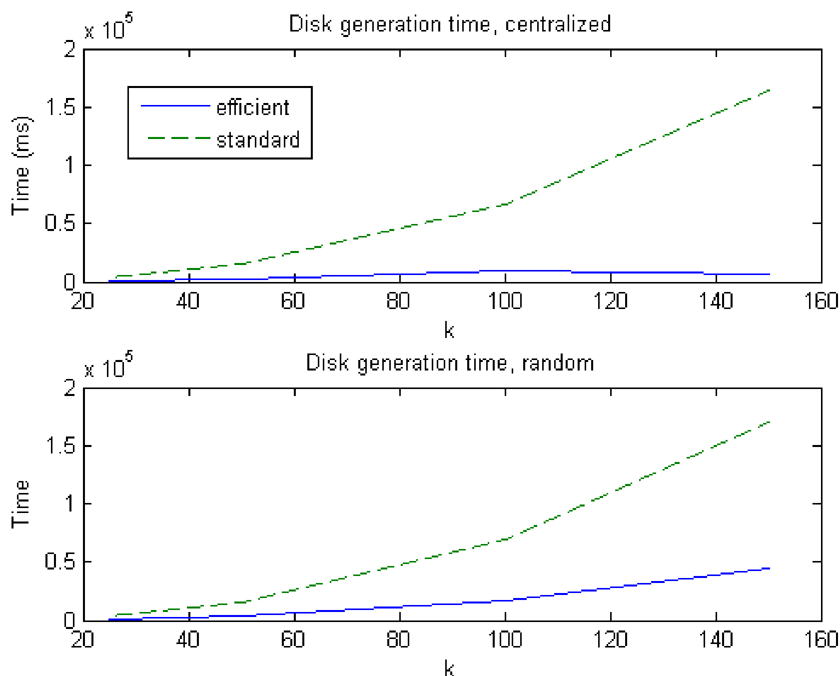
pairs	special	random(ms)	centered(ms)
10	10	798	569
10	25	688	453
10	50	676	426
10	60	620	427
10	75	610	416
25	10	811	349
25	25	660	342
25	60	610	297
25	75	626	313
50	10	427	286
50	25	433	323
50	50	406	318
50	60	448	402
50	75	458	296
60	10	437	307
60	25	453	313
60	50	400	318
60	60	417	312
60	75	402	312
75	10	391	255
75	25	391	292
75	50	390	286
75	60	396	276
75	75	375	298

Table 1: Disk Generation Times for Pairs and Special Points Constants

4.2 Implementation Comparisons

Based on the disk generation times, it would seem that fairly high constant values remain very helpful, where values of about 10% of the total number of points gave good results. However, for testing using a larger number of points, such a high constant is impractical, as 1500 MB of memory cannot handle the

Figure 1: Average Disk Generation Time for Each of the Constants



amount of storage necessary. Based on trial and error, we determined that to avoid a heap space error for the largest number of points tested, 6450, the both the pairs constant and the number of special points should be 20. We therefore used these values for all tests of the efficient implementation against the standard one and against the memory efficient one.

We tested all implementations for numbers of points = $\{1075, 2150, 4300, 6450\}$. k and the number of outliers allowed were both fixed percentages, with $k = \{25, 50, 100, 150\}$ and number of outliers = $\{50, 100, 200, 300\}$ respectively. As stated, the pairs constant was 20, as was the number of special points. The points were all generated from central distributions and were two-dimensional Euclidean on a 2000x2000 grid. As a distance measure, the square of the actual distance was used instead of distance (this is a faster computation, but does not affect results otherwise). The experiment was repeated with a random distribution and all other factors identical.

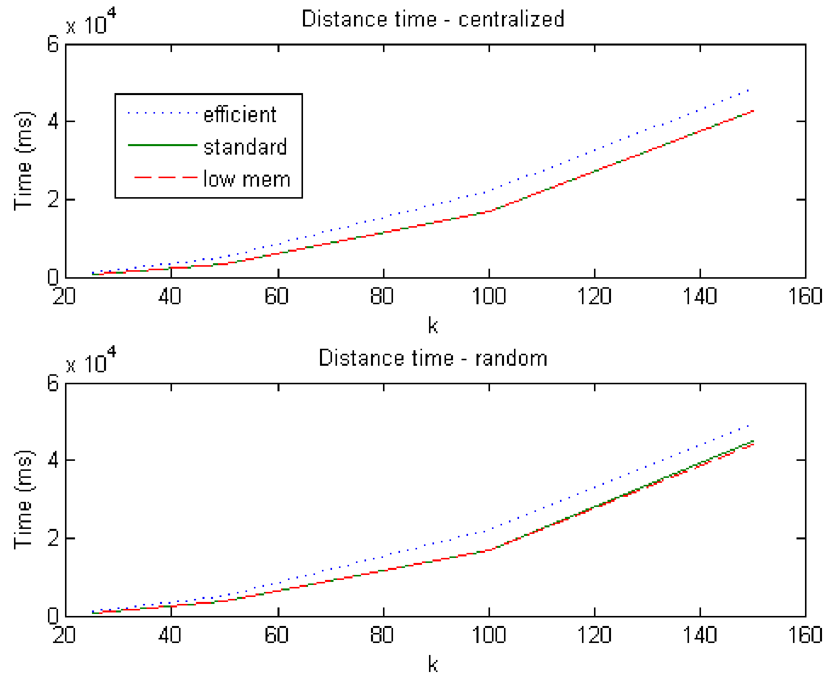
There was not much difference between the random and centralized distributions for any of the implementations. The distance time was similar for both distributions and all implementations. It was slightly larger for the efficient implementation, because of the extra time required to compute pairs lists. How-

ever, none of the differences was great.

The most prominent result was the difference in the disc update time required for the implementations. The standard disc updating time was magnitudes greater than the efficient one. The disc generation time for the efficient solution was also faster than for the naive, though not to the same degree. The disc generation time was slowest for the memory efficient implementation; this was as expected, because the low memory implementation does not do any updating but generates new discs with each of k iterations per a radius search.

For disc generation and updating times added together, the efficient implementation had by far the fastest time. Significantly, the difference in time increased as the number of points increased, indicating that the efficient solution becomes more important for larger problem sizes. In addition, fairly small pair and special point constants were chosen because of memory limits. Larger constants would result in even greater time savings. Overall, the efficient implementation proved its name.

Figure 2: Distance Time for Each of the Implementations, Centralized and Random



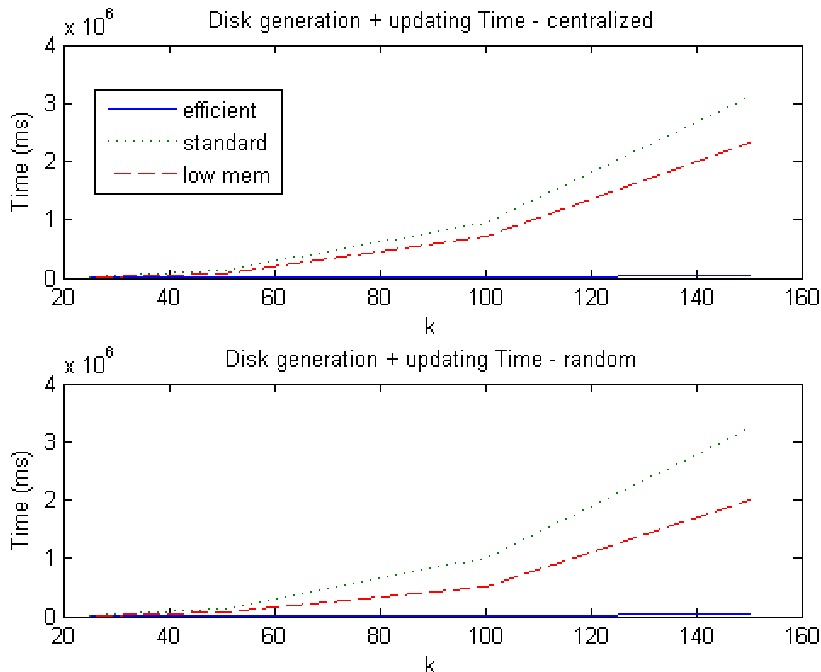
k	distance	dis gen(ms)	update(ms)
Centralized, Efficient			
25	938	515	203
50	5032	2171	594
100	22187	9797	2597
150	48312	6422	25703
Random, Efficient			
25	985	906	297
50	5078	4062	688
100	22297	16485	2827
150	49484	44157	6937
Centralized, Standard			
25	719	3486	16029
50	3484	15082	111590
100	16687	66313	886046
150	42828	163609	2935907
Random, Standard			
25	784	3844	17859
50	3516	15878	125686
100	16906	69360	945890
150	44904	170408	3063186
Centralized, Memory Efficient			
25	783	11766	0
50	3485	86843	0
100	16734	709453	0
150	42906	2328735	0
Random, Memory Efficient			
25	703	9407	0
50	3531	62109	0
100	16890	524563	0
150	43938	1987437	0

Table 2: Distance, Disk Generation, Disk Updating Times for All Implementations

4.3 Memory Considerations

The approximately 6000 point limit we encountered makes the efficient implementation unusable for large problems. We tested the memory efficient implementation, but it could only handle approximately 8000 points without running out of memory. This use of memory was primarily due to the storage of each unique distance between two points, which is essentially $O(n^2)$ storage. To reduce this need for storage, we rounded off distances to a factor of 10. In this way, we greatly reduced the number of unique distances between points. This

Figure 3: Disk Generation + Updating Time for Each Implementation, Centralized and Random



allowed us to run the implementations on many more points, although with some loss in the accuracy of the solution. The efficient implementation ran successfully for sizes up to $k = 400$, with 80 points per cluster, for a total of $33,200$ points. The memory implementation could handle this size, and $k = 500$ with 90 points per cluster, or $46,500$ points, as well. However, when we ran the implementation with this number of points, it had not finished after several hours. To test larger sizes, we simply generated points without actually finding a solution. Using this method, we found that the memory implementation could handle as many as $304,500$ points in terms of storage cost, but was far too slow to run with such a large number of points. As expected, therefore, the memory efficient implementation has by far the smallest storage demand, but it is so slow it cannot be used for a large number of points.

5 Conclusions

The algorithm proposed by Charikar et al. finds an approximate solution of within three times the optimum for the robust k-center clustering problem. The algorithm, while it is polynomial time, is impractically slow when implemented

naively. Our efficient implementation greatly decreases the computational time necessary to compute the solution. It does so while adding only an $O(n)$ memory cost. Though it uses more memory than a memory-focused implementation, it is far faster than that method as well.

There is some work that can be done to improve the algorithm. Firstly, the chief memory block in running the solution for more points is storage of every possible distance, an $O(n^2)$ cost. However, this distance list is used only for binary search purposes. For a lower degree of accuracy, far fewer exact distances need be stored; our preliminary work indicates this savings is considerable. In future work, we will investigate this approximation more thoroughly. In general, the algorithm uses a large amount of memory, and we will examine methods to bring this cost down.

Some sections of the algorithm were not improved in our efficient implementation, but were left identical to their naive counterparts. In particular, disc generation remained an $O(n^2)$ task. Also, we did not develop a fast method for finding the current largest disc after an update. We would like to improve both of these running times in future work.

6 Acknowledgements

We would like to thank Samir Khuller for his help throughout every stage of this work.

References

[1] Mihai Bādoiu, Sariel Har-Peled, and Piotr Indyk. Approximate clustering via core-sets. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 250–257, New York, NY, USA, 2002. ACM.

[2] Moses Charikar, Samir Khuller, David M. Mount, and Giri Narasimhan. Algorithms for facility location problems with outliers. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 642–651, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[3] Moses Charikar, Liadan O'Callaghan, and Rina Panigrahy. Better streaming algorithms for clustering problems. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 30–39, New York, NY, USA, 2003. ACM.

[4] Sudipto Guha and Samir Khuller. Greedy strikes back: improved facility location algorithms. In *SODA '98: Proceedings of the ninth annual ACM-*

SIAM symposium on Discrete algorithms, pages 649–657, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[5] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: an efficient clustering algorithm for large databases. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 73–84, New York, NY, USA, 1998. ACM.

[6] Ville Hautammki, Svetlana Cherednichenko, Ismo Karkkainen, Tomi Kinnunen, and Pasi Franti. Improving k-means by outlier removal. In *SCIA 2005: Scandinavian conference on image analysis*, pages 978–987, Berlin, Allemagne, 2005. Springer.

[7] M. F. Jaing, S. S. Tseng, and C. M. Su. Two-phase clustering process for outliers detection. *Pattern Recogn. Lett.*, 22(6-7):691–700, 2001.

[8] Mohammad Mahdian, Yinyu Ye, and Jiawei Zhang. Approximation algorithms for metric facility location problems. *SIAM J. Comput.*, 36(2):411–432, 2006.

[9] David B. Shmoys, Éva Tardos, and Karen Aardal. Approximation algorithms for facility location problems (extended abstract). In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 265–274, New York, NY, USA, 1997. ACM.