Even Faster Exact k-Means Clustering

Christian Borgelt

Paris-Lodron-University of Salzburg
Depts. Mathematics/Computer Sciences
Hellbrunner Straße 34
A-5020 Salzburg, Austria
christian.borgelt@sbg.ac.at

University of Konstanz
Dept. Computer & Information Science
Universitätsstraße 10
D-78457 Konstanz, Germany
christian@borgelt.net

Abstract. A naïve implementation of k-means clustering requires computing for each of the n data points the distance to each of the k cluster centers, which can result in fairly slow execution. However, by storing distance information obtained by earlier computations as well as information about distances between cluster centers, the triangle inequality can be exploited in different ways to reduce the number of needed distance computations, e.g. [5, 7, 3, 4, 11]. In this paper I present an improvement of the Exponion method [11] that generally accelerates the computations. Furthermore, by evaluating several methods on a fairly wide range of artificial data sets, I derive a kind of map, for which data set parameters which method (often) yields the lowest execution times.

Keywords: exact k-means · triangle inequality · exponion

1 Introduction

The k-means algorithm [9] is, without doubt, the best known and (among) the most popular clustering algorithm(s), mainly because of its simplicity. However, a naïve implementation of the k-means algorithm requires $O(nk)$ distance computations in each update step, where n is the number of data points and k is the number of clusters. This can be a severe obstacle if clustering is to be carried out on truly large data sets with hundreds of thousands or even millions of data points and hundreds to thousands of clusters, especially in high dimensions.

Hence, in our "big data" age, considerable effort was spent on trying to accelerate the computations, mainly by reducing the number of needed distance computations. This led to several very clever approaches, including [5, 7, 3, 4, 11]. These methods exploit that for assigning data points to cluster centers knowing actual distances is not essential (in contrast to e.g. fuzzy c-means clustering [2]). All one really needs to know is which center is closest. This, however, can sometimes be determined without actually computing (all) distances.

A core idea is to maintain, for each data point, bounds on its distance to different centers, especially to the closest center. These bounds are updated by exploiting the triangle inequality, and can enable us to ascertain that the center that was closest before the most recent update step is still closest. Furthermore, by maintaining additional information, tightening these bounds can sometimes be done by looking at only a subset of the cluster centers.

In this paper I present an improvement of one of the most sophisticated of such schemes: the Exponion method [11]. In addition, by comparing my new approach to other methods on several (artificial) data sets with a wide range of number of dimensions and number of clusters, I derive a kind of map, for which data set parameters which method (often) yields the lowest execution times.

2 *k*-Means Clustering

The k-means algorithm is a very simple, yet effective clustering scheme that finds a user-specified number k of clusters in a given data set. This data set is commonly required to consist of points in a metric space. The algorithm starts by choosing an initial set of k cluster centers, which may naïvely be obtained by sampling uniformly at random from the given data points. In the subsequent cluster center optimization phase, two steps are executed alternatingly: (1) each data point is assigned to the cluster center that is closest to it (that is, closer than any other cluster center) and (2) the cluster centers are recomputed as the vector means of the data points assigned to them (to enable these mean computations, the data points are supposed to live in a metric space).

Using $\nu_m(x)$ to denote the cluster center m-th closest to a point x in the data space, this update scheme can be written (for n data points $x_1, \ldots, x_n$) as

$$\forall i; 1 \leq i \leq k: \qquad c_i^{t+1} = \frac{\sum_{j=1}^{n} \mathbb{1}(\nu_1^t(x_j) = c_i^t) \cdot x_j}{\sum_{j=1}^{n} \mathbb{1}(\nu_1^t(x_j) = c_i^t)},$$

where the indices t and $t+1$ indicate the update step and the function $\mathbb{1}(\phi)$ yields 1 if ϕ is true and 0 otherwise. Here $\nu_1^t(x_j)$ represents the assignment step and the fraction computes the mean of the data points assigned to center c_i.

It can be shown that this update scheme must converge, that is, must reach a state in which another execution of the update step does not change the cluster centers anymore [14]. However, there is no guarantee that the obtained result is optimal in the sense that it yields the smallest sum of squared distances between the data points and the cluster centers they are assigned to. Rather, it is very likely that the optimization gets stuck in a local optimum. It has even been shown that k-means clustering is NP-hard for 2-dimensional data [10].

Furthermore, the quality of the obtained result can depend heavily on the choice of the initial centers. A poor choice can lead to inferior results due to a local optimum. However, improvements of naïvely sampling uniformly at random from the data points are easily found, for example the Maximin method [8] and the k-means++ procedure [1], which has become the *de facto* standard.

3 Bounds-based Exact *k*-Means Clustering

Some approaches to accelerate the k-means algorithm rely on approximations, which may lead to different results, e.g. [6, 12, 13]. Here, however, I focus on

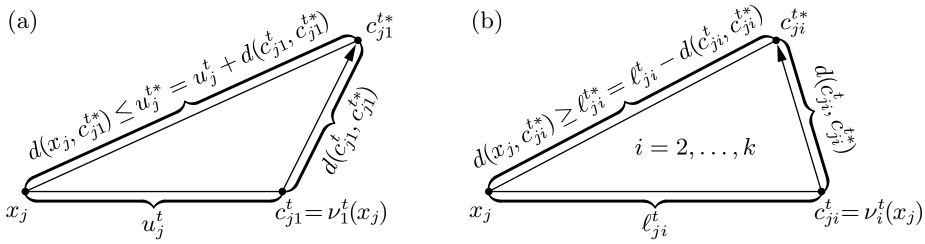

Fig. 1. Using the triangle inequality to update the distance bounds for a data point x_j.

methods to accelerate *exact* k-means clustering, that is, methods that, starting from the same initialization, produce the same result as a naïve implementation.

The core idea of these methods is to compute for each update step the distance each center moved, that is, the distance between the new and the old location of the center. Applying the triangle inequality one can then derive how close or how far away an updated center can be from a data point in the worst possible case. For this we distinguish between the center closest (before the update) to a data point x_j on the one hand and all other centers on the other.

k **Distance Bounds.** The first approach along these lines was developed in [5] and maintains one distance bound for each of the k cluster centers.

For the center closest to a data point x_j an upper bound u_j^t on its distance is updated as shown in Figure 1(a): If we know before the update that the distance between x_j and its closest center $c_{j1}^t = \nu_1^t(x_j)$ is (at most) u_j^t, and the update moved the center c_{j1}^t to the new location c_{j1}^{t*}, then the distance $d(x_j, c_{j1}^{t*})$ between the data point and the new location of this center[1] cannot be greater than $u_j^{t*} = u_j^t + d(c_{j1}^t, c_{j1}^{t*})$. This bound is actually tight if before the update the bound was tight and the center c_{j1}^t moves away from the data point x_j on the straight line through x_j and c_{j1}^t (that is, if the triangle is “flat”).

For all other centers, that is, centers that are *not* closest to the point x_j, lower bounds ℓ_{ji}, $i = 2, \ldots, k$, are updated as shown in Figure 1(b): If we know before the update that the distance between x_j and a center $c_{ji}^t = \nu_i^t(x_j)$, is (at least) ℓ_{ji}^t, and the update moved the center c_{ji}^t to the new location c_{ji}^{t*}, then the distance $d(x_j, c_{ji}^{t*})$ between the data point and the new location of this center cannot be less than $\ell_{ji}^{t*} = \ell_{ji}^t - d(c_{ji}^t, c_{ji}^{t*})$. This bound is actually tight if before the update the bound was tight and the center c_{ji}^t moves towards the data point x_j on the straight line through x_j and c_{ji}^t (“flat” triangle).

These bounds are easily exploited to avoid distance computations for a data point x_j: If we find that $u_j^{t*} < \ell_j^{t*} = \min_{i=2}^k \ell_{ji}^{t*}$, that is, if the upper bound on the distance to the center that was closest before the update (in step t) is less than the smallest lower bound on the distances to any other center, the center that was closest before the update must still be closest after the update (that is,

[1] Note that it may be $c_{j1}^{t*} \neq c_{j1}^{t+1}$ (although equality is not ruled out either), because the update may have changed which cluster center is closest to the data point x_j.

in step $t+1$). Intuitively: even if the worst possible case happens, namely if the formerly closest center moves straight away from the data point and the other centers move straight towards it, no other center can have been brought closer than the one that was already closest before the update.

And even if this test fails, one first computes the actual distance between the data point x_j and c_{j1}^{t*}. That is, one tightens the bound u_j^{t*} to the actual distance and then reevaluates the test. If it succeeds now, the center that was closest before the update must still be closest. Only if the test fails also with the tightened bound, the distances between the data point and the remaining cluster centers have to be computed in order to find the closest center and to reinitialize the bounds (all of which are tight after such a computation).

This scheme leads to considerable acceleration, because the cost of computing the distances between the new and the old locations of the cluster centers as well as the cost of updating the bounds is usually outweighed by the distance computations that are saved in those cases in which the test succeeds.

2 **Distance Bounds.** A disadvantage of the scheme just described is that k bound updates are needed for each data point. In order to reduce this cost, in [7] only two bounds are kept per data point: u_j^t and ℓ_j^t, that is, all non-closest centers are captured by a single lower bound. This bound is updated according to $\ell_j^{t*} = \ell_j^t - \max_{i=2}^k d(c_{ji}^t, c_{ji}^{t*})$. Even though this leads to worse lower bounds for the non-closest centers (since they are all treated as if they moved by the maximum of the distances any one of them moved), the fact that only two bounds have to be updated leads to faster execution, at least in many cases.

YinYang Algorithm. Instead of having either one distance bound for each center (k bounds) or capturing all non-closest centers by a single bound (2 bounds), one may consider a hybrid approach that maintains lower bounds for subsets of the non-closest centers. This improves the quality of bounds over the 2 bounds approach, because bounds are updated only by the maximum distance a center in the corresponding group moved (instead of the global maximum). On the other hand, (considerably) fewer than k bounds have to be updated.

This is the idea of the YinYang algorithm [4], which forms the groups of centers by clustering the initial centers with k-means clustering. The number of groups is chosen as $k/10$ in [4], but other factors may be tried. The groups found initially are maintained, that is, there is no re-clustering after an update.

However, apart from fewer bounds (compared to k bounds) and better bounds (compared to 2 bounds), grouping the centers has yet another advantage: If the bounds test fails, even with a tightened bound u_j^t, the groups and their bounds may be used to limit the centers for which a distance recomputation is needed. Because if the test succeeds for some group, one can infer that the closest center cannot be in that group. Only centers in groups, for which the group-specific test fails, need to be considered for recomputation.

Cluster to Cluster Distances. The described bounds test can be improved by not only computing the distance each center moved, but also the distances between (updated) centers, to find for each center another center that is closest to

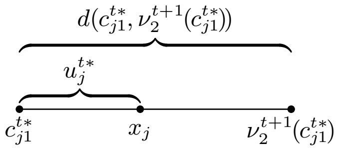

Fig. 2. If $2u^{t*}_j < d(c^{t*}_{j1}, \nu^{t+1}_2(c^{t*}_{j1}))$, then the center c^{t*}_{j1} must still be closest to the data point x_j, due to the triangle inequality.

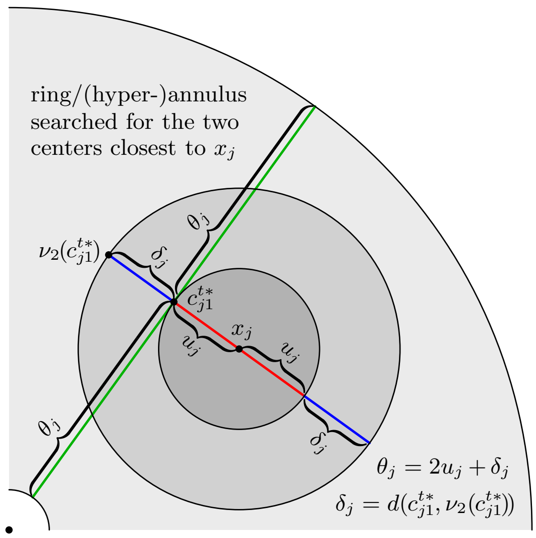

Fig. 3. Annular algorithm [3]: If even after the upper bound u_j for the distance from data point x_j to its (updated) formerly closest center c^{t*}_{j1} has been made tight, the lower bound ℓ_j for distances to other centers is still lower, it is necessary to recompute the two closest centers. Exploiting information about the distance between c^{t*}_{j1} and another center $\nu_2(c^{t*}_{j1})$ closest to it, these two centers are searched in a (hyper-)annulus around the origin (dot in the bottom left corner) with c^{t*}_{j1} in the middle and thickness $2\theta_j$, where $\theta_j = 2u_j + \delta_j$ and $\delta_j = d(c^{t*}_{i1}, \nu_2(c^{t*}_{j1}))$.

it [5]. With my notation I can denote such a center as $\nu^{t+1}_2(c^{t*}_{j1})$, that is, the center that is second closest[2] to the point c^{t*}_{j1}. Knowing the distances $d(c^{t*}_{j1}, \nu^{t+1}_2(c^{t*}_{j1}))$, one can test whether $2u^{t+1}_l < d(c^{t*}_{j1}, \nu^{t+1}_2(c^{t*}_{j1}))$. If this is the case, the center that was closest to the data point x_j before the update must still be closest after, as is illustrated in Figure 2 for the worst possible case (namely x_j, c^{t*}_{ji} and $\nu^{t+1}_2(c^{t*}_{j1})$ lie on a straight line with c^{t*}_{ji} and $\nu^{t+1}_2(c^{t*}_{j1})$ on opposite sides of x_j).

Note that this second test can be used with k as well as with 2 bounds. However, it should also be noted that, although it can lead to an acceleration, if used in isolation it may also make an algorithm slower, because of the $O(k^2)$ distance computations needed to find the k distances $d(c^{t+1}_i, \nu^{t+1}_2(c^{t+1}_i))$.

Annular Algorithm. With the YinYang algorithm an idea appeared on the scene that is at the focus of all following methods: try to limit the centers that need to be considered in the recomputations if the tests fail even with a tightened bound u^{t+1}_j. Especially, if one uses the 2 bounds approach, significant gains may be obtained: all we need to achieve in this case is to find $c^{t+1}_{i1} = \nu^{t+1}_1(x_j)$ and $c^{t+1}_{i2} = \nu^{t+1}_2(x_j)$, that is, the two centers closest to x_j, because these are all that is needed for the assignment step as well as for the (tight) bounds u^{t+1}_j and ℓ^{t+1}_j.

One such approach is the Annular algorithm [3]. For its description, as generally in the following, I drop the time step indices $t+1$ in order to simplify

[2] Note that $\nu^{t+1}_1(c^{t*}_{j1}) = c^{t*}_{j1}$, because a center is certainly the center closest to itself.

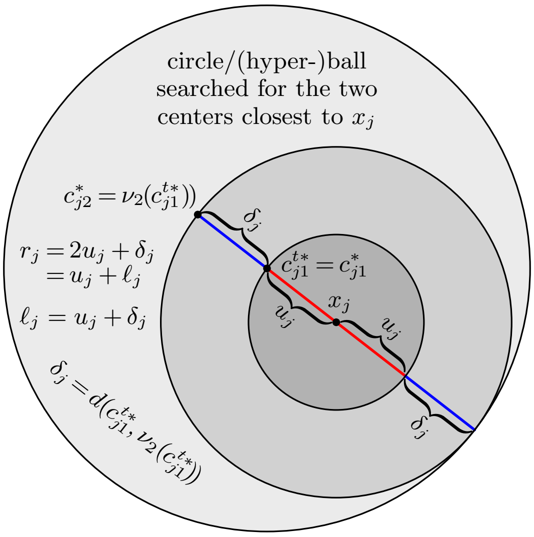

Fig. 4. Exponion algorithm [11]: If even after the upper bound u_j for the distance from a data point x_j to its (updated) formerly closest center c^{t*}_{j1} has been made tight, the lower bound ℓ_j for distance to other centers is still lower, it is necessary to recompute the two closest centers. Exploiting information about the distance between c^{t*}_{j1} and another center $\nu_2(c^{t*}_{j1})$ closest to it, these two centers are searched in a (hyper-)sphere around center c^{t*}_{j1} with radius $r_j = 2u_j + \delta_j$ where $\delta_j = d(c^{t*}_{j1}, \nu_2(c^{t*}_{j1}))$.

the notation. The Annular algorithm relies on the following idea: if the tests described above fail with a tightened bound u_j, we cannot infer that c^{t*}_{ji} is still the center closest to x_j. But we know that the closest center must lie in (hyper-)ball with radius u_j around x_j (darkest circle in Figure 3). Any center outside this (hyper-)ball cannot be closest to x_j, because c^{t*}_{ji} is closer. Furthermore, if we know the distance to another center closest to c^{t*}_{ji}, that is, $\nu_2(c^{t*}_{j1})$, we know that even in the worst possible case (which is depicted in Figure 3: x_j, c^{t*}_{ji} and $\nu_2(c^{t*}_{j1})$ lie on a straight line), the two closest centers must lie in a (hyper-)ball with radius $u_j + \delta_j$ around x_j, where $\delta_j = d(c^{t*}_{i1}, \nu_2(c^{t*}_{j1}))$ (medium circle in Figure 3), because we already know two centers that are this close, namely c^{t*}_{ji} and $\nu_2(c^{t*}_{j1})$. Therefore, if we know the distances of the centers from the origin, we can easily restrict the recomputations to those centers that lie in a (hyper-)annulus (hence the name of this algorithm) around the origin with c^{t*}_{j1} in the middle and thickness $2\theta_j$, where $\theta_j = 2u_j + \delta_j$ with $\delta_j = d(c^{t*}_{i1}, \nu_2(c^{t*}_{j1}))$ (see Figure 3, light gray ring section, origin in the bottom left corner; note that the green line is perpendicular to the red/blue lines only by accident/for drawing convenience).

Exponion Algorithm. The Exponion algorithm [11] improves over the Annular algorithm by switching from annuli around the origin to (hyper-)balls around the (updated) formerly closest center c^{t*}_{j1}. Again we know that the center closest to x_j must lie in a (hyper-)ball with radius u_j around x_j (darkest circle in Figure 4) and that the two closest centers must lie in a (hyper-)ball with radius $u_j + \delta_j$ around x_j, where $\delta_j = d(c^{t*}_{i1}, \nu_2(c^{t*}_{j1}))$ (medium circle in Figure 4). Therefore, if we know the pairwise distances between the (updated) centers, we can easily restrict the recomputations to those centers that lie in the (hyper-)ball with radius $r_j = 2u_j + \delta_j$ around c^{t*}_{j1} (lightest circle in Figure 4).

The Exponion algorithm also relies on a scheme with which it is avoided having to sort, for each cluster center, the lists of the other centers by their distance. For this concentric annuli, one set centered at a each center, are created, with each annulus further out containing twice as many centers as the preceding one. Clearly this creates an onion-like structure, with an exponentially increasing number of centers in each layer (hence the name of the algorithm).

However, avoiding the sorting comes at a price, namely that more centers may have to be checked (although at most twice as many [11]) for finding the two closest centers and thus additional distance computations ensue. In my implementation I avoided this complication and simply relied on sorting the distances, since the gains achievable by concentric annuli over sorting are somewhat unclear (in [11] no comparisons of sorting versus concentric annuli are provided).

Shallot Algorithm. The Shallot algorithm is the main contribution of this paper. It starts with the same considerations as the Exponion algorithm, but adds two improvements. In the first place, not only the closest center c_{j1} and the two bounds u_j and ℓ_j are maintained for each data point (as for Exponion), but also the second closest center c_{j2}. This comes at practically no cost (apart from having to store an additional integer per data point), because the second closest center has to be determined anyway in order to set the bound ℓ_j.

If a recomputation is necessary, because the tests fail even for a tightened u_j, it is *not* automatically assumed that c_{j1}^{t*} is the best center z for a (hyper-)ball to search. As it is plausible that the formerly second closest center c_{j2}^{t*} may now be closer to x_j than c_{j1}^{t*}, the center c_{j2}^{t*} is processed first among the centers c_{ji}^{t*}, $i = 2, \dots, k$. If it turns out that it is actually closer to x_j than c_{j1}^{t*}, then c_{j2}^{t*} is chosen as the center z of the (hyper-)ball to check. In this case the (hyper-)ball will be smaller (since we found that $d(x_j, c_{j2}^{t*}) < d(x_j, c_{j1}^{t*})$). For the following, let p denote the other (updated) center that was not chosen as the center z.

The second improvement may be understood best by viewing the chosen center z of the (hyper-)ball as the initial candidate c_{j1}^* for the closest center in step $t+1$. Hence we initialize $u_j = d(x_j, z)$. For the initial candidate c_{j2}^* for the second closest center in step $t+1$ we have two choices, namely p and $\nu_2(z)$. We choose $c_{j2}^* = p$ if $u_j + d(x_j, p) < 2u_l + \delta_j$ and $c_{j2}^* = \nu_2(z)$ otherwise, and initialize $\ell_j = u_j + d(x_j, p)$ or $\ell_j = 2u_j + \delta_j$ accordingly, thus minimizing the radius, which then can be written, regardless of the choice taken, as $r_j = u_j + \ell_j$.

While traversing the centers in the constructed (hyper-)ball, better candidates may be obtained. If this happens, the radius of the (hyper-)ball may be reduced, thus potentially reducing the number of centers to be processed. This idea is illustrated in Figure 5. Let u_j° be the initial value of u_j when the (hyper-)ball center was chosen, but before the search is started, that is $u_j^\circ = d(x_j, z)$. If a new closest center (candidate) c_{j1}^* is found (see Figure 5(a)), we can update $u_j = d(x_j, c_{j1}^*)$ and $\ell_j = d(x_j, c_{j2}^*) = u_j^\circ$. Hence we can shrink the radius to $r_j = 2u_j^\circ = u_j^\circ + \ell_j$. If then an even closer center is found (see Figure 5(b)), the radius may be shrunk further as u_j and ℓ_j are updated again. As should be clear from these examples, the radius is always $r_j = u_j^\circ + \ell_j$.

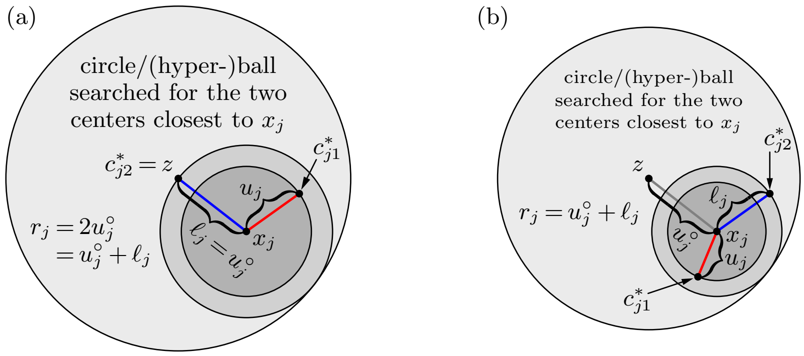

Fig. 5. Shallot algorithm: If a center closer to the data point than the two currently closest centers is found, the radius of the (hyper-)ball to be searched can be shrunk.

A *shallot* is a type of onion, smaller than, for example, a bulb onion. I chose this name to indicate that the (hyper-)ball that is searched for the two closest centers tends to be smaller than for the Exponion algorithm. The reference to an onion may appear misguided, because I rely on sorting the list of other centers by their distance for each cluster center, rather than using concentric annuli. However, an onion reference may also be justified by the fact that my algorithm may shrink the (hyper-)ball radius during the traversal of centers in the (hyper-) ball, as this also creates a layered structure of (hyper-)balls.

4 Experiments

In order to evaluate the performance of the different exact k-means algorithms I generated a large number of artificial data sets. Standard benchmark data sets proved to be too small to measure performance differences reliably and would also not have permitted drawing "performance maps" (see below). I fixed the number of data points in these data sets at $n = 100\ 000$. Anything smaller renders the time measurements too unreliable, anything larger requires an unpleasantly long time to run all benchmarks. Thus I varied only the dimensionality m of the data space, namely as $m \in \{2, 3, 4, 5, 6, 8, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$, and the number k of clusters, from 20 to 300 in steps of 20. For each parameter combination I generated 10 data sets, with clusters that are (roughly, due to random deviations) equally populated with data points and that may vary in size by a factor of at most ten per dimension. All clusters were modeled as isotropic normal (or Gaussian) distributions. Each data set was then processed 10 times with different initializations. All optimization algorithms started from the same initializations, thus making the comparison as fair as possible.

The clustering program is written in C (however, there is also a Python version, see the link to the source code below). All implementations of the different algorithms are entirely my own and use the same code to read the data and to

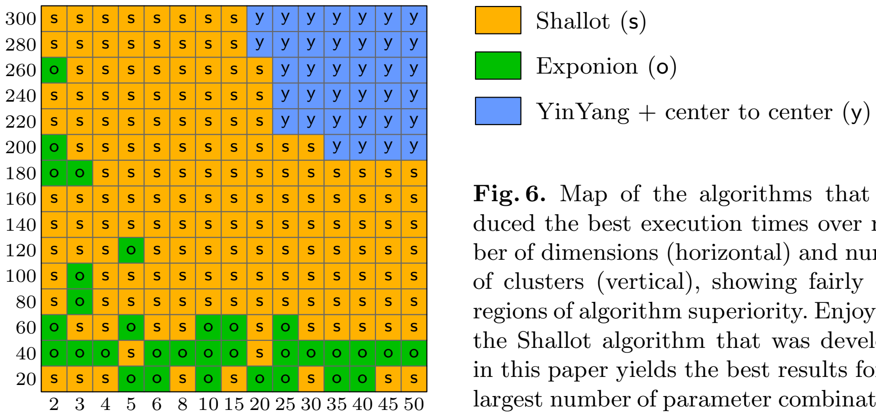

Fig. 6. Map of the algorithms that produced the best execution times over number of dimensions (horizontal) and number of clusters (vertical), showing fairly clear regions of algorithm superiority. Enjoyably, the Shallot algorithm that was developed in this paper yields the best results for the largest number of parameter combinations.

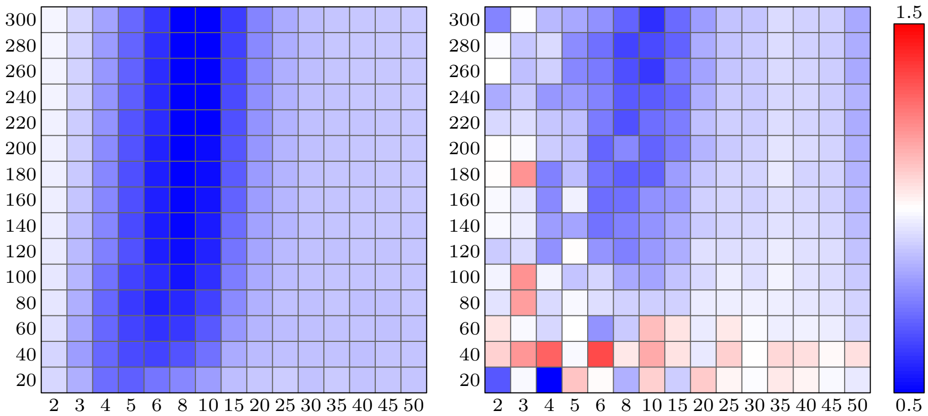

Fig. 7. Relative comparison between the Shallot algorithm and the Exponion algorithm. The left diagram refers to the number of distance computations, the right diagram to execution time. Blue means that Shallot is better, red that Exponion is better.

write the clustering results. This adds to the fairness of the comparison, as in this way any differences in execution time can only result from differences of the actual algorithms. The test systems was an Intel Core 2 Quad Q9650@3GHz with 8 GB of RAM running Ubuntu Linux 18.04 64bit.

The results of these experiments are visualized in Figures 6 to 8. Figure 6 shows on a grid spanned by the number of dimensions (horizontal axis) and the number of clusters inducted into the data set (vertical axis) which algorithm performed best (in terms of execution time) for each combination. Clearly, the Shallot algorithm wins most parameter combinations. Only for larger numbers of dimensions and larger numbers of clusters the YinYang algorithm is superior.

In order to get deeper insights, Figure 7 shows on the same grid a comparison of the number of distance computations (left) and the execution times (right) of the Shallot algorithm and the Exponion algorithm. The relative performance

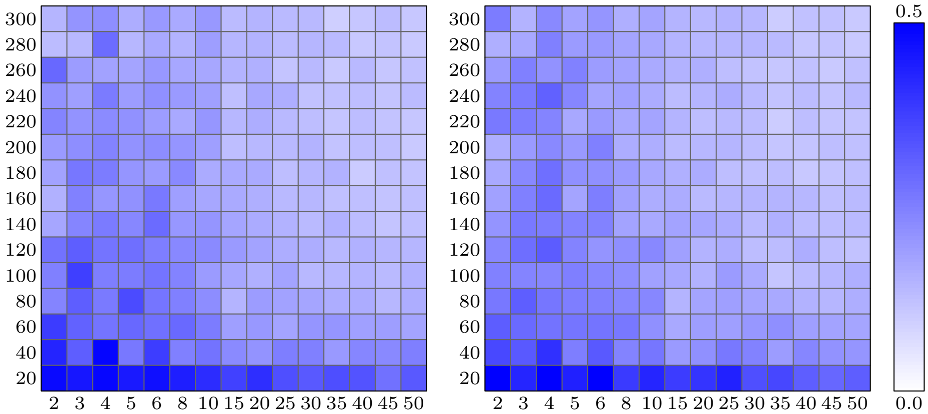

Fig. 8. Variation of the execution times over number of dimensions (horizontal) and number of clusters (vertical). The left diagram refers to the Shallot algorithm, the right diagram to the Exponion algorithm. The larger variation for fewer clusters and fewer dimensions may explain the speckled look of Figures 6 and 7.

is color-coded: saturated blue means that the Shallot algorithm needed only half the distance computations or half the execution time of the Exponion algorithm, saturated red means that it needed 1.5 times the distance computations or execution time compared to the Exponion algorithm.

W.r.t. distance computations there is no question who is the winner: the Shallot algorithm wins all parameter combinations, some with a considerable margin. W.r.t. execution times, there is also a clear region towards more dimensions and more clusters, but for fewer clusters and fewer dimensions the diagram looks a bit speckled. This is a somewhat strange result, as a smaller number of distance computations should lead to lower execution times, because the effort spent on organizing the search, which is also carried out in exactly the same situations, is hardly different between the Shallot and the Exponion algorithm.

The reason for this speckled look could be that the benchmarks were carried out with heavy parallelization (in order to minimize the total time), which may have distorted the measurements. As a test of this hypothesis, Figure 8 shows the standard deviation of the execution times relative to their mean. White means no variation, fully saturated blue indicates a standard deviation half as large as the mean value. The left diagram refers to the Shallot, the right diagram to the Exponion algorithm. Clearly, for a smaller number of dimensions and especially for a smaller number of clusters the execution times vary more (this may be, at least in part, due to the generally lower execution times for these parameter combinations). It is plausible to assume that this variability is the explanation for the speckled look of the diagrams in Figure 6 and in Figure 7 on the right.

Finally, Figure 9 shows, again on the same grid, a comparison of the number of distance computations (left) and the execution times (right) of the Shallot algorithm and the YinYang algorithm (using the test based on cluster to cluster

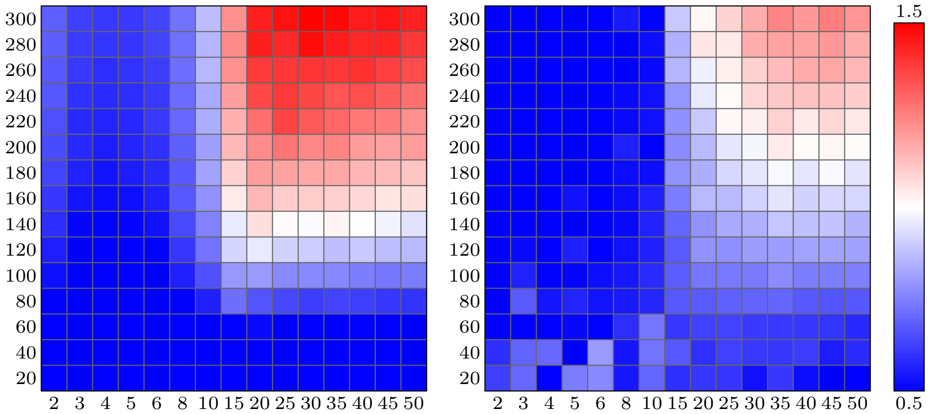

Fig. 9. Relative comparison between the Shallot algorithm and the YinYang algorithm using the cluster to cluster distance test (pure YinYang is very similar, though). The left diagram refers to the number of distance computations, the right diagram to execution time. Blue means that Shallot is better, red that YinYang is better.

distances, although a pure YinYang algorithm performs very similarly). The relative performance is color-coded in the same way as in Figure 7. Clearly, the smaller number of distance computations explains why the YinYang algorithm is superior for more clusters and more dimensions.

The reason is likely that grouping the centers leads to better bounds. This hypothesis is confirmed by the fact that the Elkan algorithm (k distance bounds) always needs the fewest distance computations (not shown as a grid) and loses on execution time only due to having to update so many distance bounds.

5 Conclusion

In this paper I introduced the Shallot algorithm, which adds two improvements to the Exponion algorithm [11], both of which can potentially shrink the (hyper-) ball that has to be searched for the two closest centers if recomputation becomes necessary. This leads to a measurable, sometimes even fairly large speedup compared to the Exponion algorithm due to fewer distance computations. However, for high-dimensional data and large numbers of clusters the YinYang algorithm [4] (with or without the cluster to cluster distance test) is superior to both algorithms. Yet, since clustering in high dimensions is problematic anyway due to the curse of dimensionality, it may be claimed reasonably confidently that the Shallot algorithm is the best choice for standard clustering tasks.

Software. My implementation of the described methods (C and Python), with which I conducted the experiments, can be obtained under the MIT License at

`http://www.borgelt.net/cluster.html`

Complete Results. A table with the complete experimental results I obtained can be retrieved as a simple text table at

`http://www.borgelt.net/docs/clsbench.txt`

More maps comparing the performance of the algorithms can be found at

`http://www.borgelt.net/docs/clsbench.pdf`

References

1. D. Arthur and S. Vassilvitskii. k-Means++: The Advantages of Careful Seeding. *Proc. 18th Annual SIAM Symp. on Discrete Algorithms (SODA'07, New Orleans, LA)*, 1027–1035. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007
2. J.C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms.* Plenum Press, New York, NY, USA 1981
3. J. Drake. *Faster k-Means Clustering,* Master's Thesis, Baylor University, Waco, TX, USA 2013
4. Y. Ding, Y. Zhao, Y. Shen, M. Musuvathi, and T. Mytkowicz. YinYang k-Means: A Drop-in Replacement of the Classic k-Means with Consistent Speedup. *Proc. 32nd Int. Conf. on Machine Learning, (ICML'15, Lille, France)*, 579–587. JMLR Workshop and Conference Proceedings 37, 2015
5. C. Elkan. Using the Triangle Inequality to Accelerate k-Means. *Proc.* 20^{th} *Int. Conf. on Machine Learning (ICML'03, Washington, DC)*, 147–153. AAAI Press, Menlo Park, CA, USA 2003
6. G. Frahling and C. Sohler. A Fast k-Means Implementation Using Coresets. *Proc. 22nd Ann. Symp. on Computational Geometry (SCG'06, Sedona, AZ)*, 135–143. ACM Press, New York, NY, USA 2006
7. G. Hamerly. Making k-Means Even Faster. *Proc. SIAM Int. Conf. on Data Mining (SDM 2010, Columbus, OH)*, 130–140. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA 2010
8. R.J. Hathaway, J.C. Bezdek, and J.M. Huband. Maximin Initialization for Cluster Analysis. *Proc. Iberoamerican Cong. on Pattern Recognition (CIARP 2006, Cancun, Mexico)*, 14–26. Springer-Verlag, Berlin/Heidelberg, Germany 2006
9. S.P. Lloyd. Least Square Quantization in PCM. *Bell Telephone Laboratories Paper.* Bell Laboratories, Murray Hill, NJ, USA 1957; also in: *IEEE Trans. on Information Theory* 28:129–137. IEEE Press, Piscataway, NJ, USA 1982
10. M. Mahajan, P. Nimbhorkar and K. Varadarajan. The Planar k-Means Problem is NP-hard. *Theoretical Computer Science* 442:13–21. Elsevier, Amsterdam, Netherlands 2009
11. J. Newling and F. Fleuret. Fast k-Means with Accurate Bounds. *Proc. 33rd Int. Conf. on Machine Learning (ICML'16, New York, NY)*, 936–944. JMLR Workshop and Conference Proceedings 48, 2016
12. J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object Retrieval with Large Vocabularies and Fast Spatial Matching. *Proc. IEEE Int. Conf. on Computer Vision and Pattern Recognition* (CVPR'07, Minneapolis, MN). IEEE Press, Piscataway, NJ, USA 2007
13. D. Sculley. Web-scale k-Means Clustering. *Proc. 19th Int. Conf. on World Wide Web (WWW'10, Raleigh, NC)*, 1177–1178. ACM Press, New York, NY, USA 2010

14. S.Z. Selim and M.A. Ismail. k-Means-Type Algorithms: A Generalized Convergence Theorem and Characterization of Local Optimality. *IEEE Trans. Pattern Analysis and Machine Intelligence*, (6)1:81–87. IEEE Press, Piscataway, NJ, USA 1984