Faster k-Medoids Clustering: Improving the PAM, CLARA, and CLARANS Algorithms

Erich Schubert · **Peter J. Rousseeuw**

Updated draft, 2019-05-04

Abstract Clustering non-Euclidean data is difficult, and one of the most used algorithms besides hierarchical clustering is the popular algorithm Partitioning Around Medoids (PAM), also simply referred to as k-medoids.

In Euclidean geometry the mean—as used in k-means—is a good estimator for the cluster center, but this does not exist for arbitrary dissimilarities. PAM uses the medoid instead, the object with the smallest dissimilarity to all others in the cluster. This notion of centrality can be used with any (dis-)similarity, and thus is of high relevance to many domains such as biology that require the use of Jaccard, Gower, or more complex distances.

A key issue with PAM is, however, its high run time cost. In this paper, we propose modifications to the PAM algorithm where at the cost of storing $O(k)$ additional values, we can achieve an $O(k)$-fold speedup in the second ("SWAP") phase of the algorithm, but will still find the same results as the original PAM algorithm. If we slightly relax the choice of swaps performed (while retaining comparable quality), we can further accelerate the algorithm by performing up to k swaps in each iteration. With the substantially faster SWAP, we can now also explore alternative (faster) strategies for choosing the initial medoids. We also show how the CLARA and CLARANS algorithms benefit from the proposed modifications.

While we do not further study the parallelization of our approach in this work, it can easily be combined with earlier approaches to use PAM and CLARA on big data (some of which use PAM as a subroutine, hence can immediately benefit from these improvements), where the performance with high k becomes increasingly important.

In experiments on real data with $k = 100$, we observed a $200\times$ speedup compared to the original PAM SWAP algorithm, making PAM applicable to larger data sets as long as we can afford to compute a distance matrix, and in particular to higher k (at $k = 2$, the new SWAP was only 1.5 times faster, as the speedup is expected to increase with k).

Keywords Cluster Analysis · k-Medoids · PAM · CLARA · CLARANS

Erich Schubert
Technische Universität Dortmund, Dortmund, Germany
E-mail: erich.schubert@tu-dortmund.de

Peter J. Rousseeuw
Department of Mathematics, KU Leuven, Leuven, Belgium
E-mail: peter@rousseeuw.net

1 Introduction

Clustering is a common unsupervised machine learning task, in which the data set has to be automatically partitioned into "clusters", such that objects within the same cluster are more similar, while objects in different clusters are more different. There is not (and likely never will be) a generally accepted definition of a cluster, because "clusters are, in large part, in the eye of the beholder" (Estivill-Castro, 2002), meaning that every user may have different enough needs and intentions to want a different algorithm and notion of cluster. And therefore, over many years of research, hundreds of clustering algorithms and evaluation measures have been proposed, each with their merits and drawbacks. Nevertheless, a few seminal methods such as hierarchical clustering, k-means, PAM (Kaufman and Rousseeuw, 1987, 1990, Ch. 2), and DBSCAN (Ester et al., 1996) have received repeated and widespread use. One may be tempted to think that after 60 to 20 years these methods have all been well researched and understood, but there are still many scientific publications trying to explain these algorithms better (e.g., Schubert et al. 2017), trying to parallelize and scale them to larger data sets (e.g., Lijffijt et al. 2015; Yang and Lian 2014), trying to better understand similarities and relationships among the published methods (e.g., Schubert et al. 2018), or proposing further improvements – and so does this paper for the widely used PAM algorithm, also often referred to as k-medoids.

In hierarchical agglomerative clustering (HAC), each object is initially its own cluster. The two closest clusters are then merged repeatedly to build a cluster tree called dendrogram. HAC is a very flexible method: it can be used with any distance or (dis-)similarity, and it allows for different rules of aggregating the object distances into cluster distances, such as the minimum ("single linkage"), average, or maximum ("complete linkage"). Single linkage directly corresponds to the minimum spanning tree of the distance graph. While the dendrogram is a powerful visualization for small data sets, extracting flat partitions from hierarchical clustering is not trivial, and thus users often turn to simpler methods.

Another classic method taught in textbooks is k-means (for an overview of the complicated history of k-means, refer to Bock 2007), where the data is modeled using k cluster means, that are iteratively refined by assigning all objects to the nearest mean, then recomputing the mean of each cluster. This optimization converges because the mean is the least squares estimator of location, and both steps optimize the same quantity, a measure known as sum-of-squared errors (SSQ, also called SSE, and equivalent to WCSS):

$$SSQ := \sum_{i=1}^{k} \sum_{x_j \in C_i} ||x_j - \mu_i||_2^2 \,. \tag{1}$$

In k-medoids, the data is modeled very similarly, but using k representative objects m_i called medoids (chosen from the data set; defined below) that can serve as "prototypes" for the cluster instead of means in order to allow using arbitrary other dissimilarities and arbitrary input domains, using the absolute error criterion ("total deviation", TD) as objective:

$$TD := \sum_{i=1}^{k} \sum_{x_j \in C_i} d(x_j, m_i) \,, \tag{2}$$

which is the sum of dissimilarities of each point $x_j \in C_i$ to the medoid m_i of its cluster. If we use squared Euclidean as distance function (i.e., $d(x, m) = ||x - m||_2^2$), we almost obtain the usual SSQ objective used by k-means, except that k-means is free to choose any $\mu_i \in \mathbb{R}^d$, whereas in k-medoids $m_i \in C_i$ must be one of the original data points. For *squared* Euclidean distances and Bregman divergences, the arithmetic mean is the optimal choice for μ and a fixed cluster assignment. For L_1 distance (i.e, $\sum |x_i - y_i|$), also called Manhattan distance,

the component-wise median is a better choice in $\mathbb{R}^d$ (Bradley et al., 1996, k-medians). For unsquared Euclidean distances,[1] we get the much harder Weber problem (Overton, 1983), which has no closed-form solution (Bradley et al., 1996) (for a recent survey of algorithms for the Weber point see Fritz et al. 2012). For other distance functions, finding a closed form to compute the best m_i would require non-trivial mathematical analysis of each distance function separately. Furthermore, our input data does not necessarily come from a $\mathbb{R}^d$ vector space. In k-medoids clustering, we therefore constrain m_i to be one of our data samples. The medoid of a set C is defined as the object with the smallest sum of dissimilarities (or, equivalently, smallest average) to all other objects in the set:

$$\text{medoid}(C) := \arg\min_{x_i \in C} \sum\nolimits_{x_j \in C} d(x_i, x_j)$$

This definition does not require the dissimilarity to be a metric, and by using $\arg\max$ instead of the $\arg\min$ it can also be applied to similarities. The algorithms discussed below all can trivially be modified to maximize similarities rather than minimizing distances, and none assumes the triangular inequality. Partitioning Around Medoids (PAM, Kaufman and Rousseeuw 1987, 1990, Ch. 2) is the most widely known algorithm to find a good partitioning using medoids, with respect to TD (Equation 2).

2 Partitioning Around Medoids (PAM)

The "Program PAM" (Kaufman and Rousseeuw, 1987, 1990, Ch. 2) consists of two algorithms, BUILD to choose an initial clustering, and SWAP to further improve the clustering towards a local optimum (finding the global optimum of the k-medoids problem is, unfortunately, NP-hard). The algorithms require a dissimilarity matrix (for example computed using the routine DAISY of Kaufman and Rousseeuw 1990), which requires $O(n^2)$ memory and for many popular distance functions in d dimensional data $O(n^2 d)$ time to compute (but potentially much more for expensive distances such as earth movers distance). Computing the distance matrix will therefore in many cases be much of the computational cost already.

2.1 BUILD Initialization Algorithm

In order to find a good initial clustering (rather than relying on a random sampling strategy as commonly used with k-means), BUILD chooses k times the point which yields the smallest distance sum TD (in the first iteration, this means choosing the point with the smallest distance to all others; afterwards adding the point as next medoid, that reduces TD most). We give a pseudocode in Algorithm 1, where we use ΔTD as symbol for the change in TD (which should be negative to be beneficial), and * for the best values found so far. If we cache the nearest medoid in line 13, then BUILD initialization needs $O(n^2 k)$ time, so it already is a fairly expensive algorithm. The motivation here was to find a good starting point, in order to require fewer iterations of the refinement procedure. In the experiments, we will also study whether a clever sampling-based approach similar to k-means++ (Arthur and Vassilvitskii, 2007) that needs only $O(nk)$ time but produces a worse starting point is an interesting alternative.

[1] It is a common misconception that k-means would minimize Euclidean distances. It optimizes the sum of *squared* Euclidean distances, and even then the textbook algorithm may end up slightly off a local optimum, because always assigning a point to its nearest center *can* increase the variance by moving the centers away from other points (Hartigan and Wong, 1979).

Algorithm 1: PAM BUILD: Find initial cluster centers.

```
1  (TD, m_1) ← (∞, null);
2  foreach x_j do                                                    // First medoid
3  |   TD_j ← 0 ;
4  |   foreach x_o ≠ x_j do  TD_j ← TD_j + d(x_o, x_j);
5  |   if TD_j < TD then (TD, m_1) ← (TD_j, x_j);                    // Smallest distance sum
6  for i = 1 ... k − 1 do                                            // Other medoids
7  |   (ΔTD*, x*) ← (∞, null);
8  |   foreach x_j ∉ {m_1, ..., m_i} do
9  |   |   ΔTD ← 0 ;
10 |   |   foreach x_o ∉ {m_1, ..., m_i, x_j} do
11 |   |   |   δ ← d(x_o, x_j) − min_{o∈m_1,...,m_i} d(x_o, o);
12 |   |   |   if δ < 0 then ΔTD ← ΔTD + δ;
13 |   |   if ΔTD < ΔTD* then (ΔTD*, x*) ← (ΔTD, x_j);              // best reduction in TD
14 |   (TD, m_{i+1}) ← (TD + ΔTD*, x*);
15 return TD, {m_1, ..., m_k};
```

Algorithm 2: PAM SWAP: Iterative improvement.

```
1  repeat
2  |   (ΔTD*, m*, x*) ← (0, null, null) ;
3  |   foreach m_i ∈ {m_1, ..., m_k} do                              // each medoid
4  |   |   foreach x_j ∉ {m_1, ..., m_k} do                         // each non-medoid
5  |   |   |   ΔTD ← 0 ;
6  |   |   |   foreach x_o ∉ {m_1, ..., m_k} \ m_i do  ΔTD ← ΔTD + Δ(x_o, m_i, x_j) ;
7  |   |   |   if ΔTD < ΔTD* then (ΔTD*, m*, x*) ← (ΔTD, m_i, x_j) ;
8  |   break loop if ΔTD* ≥ 0 ;
9  |   swap roles of medoid m* and non-medoid x* ;                  // perform best swap
10 |   TD ← TD + ΔTD* ;
11 return TD, M, C;
```

2.2 SWAP Refinement Algorithm

The second algorithm, which is the main focus of this paper, was named SWAP. In order to improve the clustering, it considers all possible changes to the set of k medoids, which effectively means replacing (swapping) some medoid with some non-medoid, which gives $k(n-k)$ candidate swaps. If it reduces TD, the best such change is then applied, in the spirit of a greedy steepest-descent method, and this process is repeated until no further improvements are found. We give a pseudocode of this in Algorithm 2. If we again cache the necessary data to compute the $\Delta(x_o, m_i, x_j)$ function (Equation 4, explained in Section 3) in line 6 efficiently, then the run time of this algorithm is $O(k(n-k)^2)$ for each iteration. While the authors of PAM assumed that only few iterations will be needed (if the algorithm is already initialized well, using the BUILD algorithm above), we do see an increasing number of iterations with increasing amounts of data (but usually we will have fewer than k iterations).

Both the pseudocode for BUILD and SWAP given here omit the details of managing the cached distances. For each object, we need to store the index of the nearest medoid $\mathit{nearest}(o)$ and its distance $d_{\mathit{nearest}}(o)$, and also the distance to the second nearest medoid $d_{\mathit{second}}(o)$ (if we also store the index of the second nearest center, we may be able to avoid some more distance computations). In particular in line 9, when executing the best swap, we need to carefully update the cached values.

2.3 Variants of PAM

The algorithm CLARA (Kaufman and Rousseeuw 1986, 1990, Ch. 3) repeatedly applies PAM on a subsample with $n' \ll n$ objects, with the suggested value $n' = 40 + 2k$. Afterwards, the remaining objects are assigned to their closest medoid. The run with the least TD (on the entire data) is returned. If the sample size is chosen $n' \in O(k)$ as suggested, the run time reduces to $O(k^3)$, which explains why the approach is typically used only with small k (Lucasius et al., 1993). Because CLARA uses PAM internally, it will directly benefit from the improvements proposed in this article.

Lucasius et al. (1993) propose a genetic algorithm to find the best k-medoids partitioning, which will perform a randomized exploration of the search space based on "mutation" of the best solutions found so far. Crossover mutations correspond to taking some medoids from both "parents", whereas mutations replace medoids with random objects. It is not obvious that this will efficiently provide a sufficient coverage of the enormous search space (there are $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ possible sets of medoids) for a large k. In order to benefit from the proposed improvements, a more systematic mutation strategy would need to be adopted, making the method similar to CLARANS below. Wei et al. (2003) found the genetic methods to work only for small data sets, small k, and well separated symmetric clusters, and it was usually outperformed by CLARANS.

The algorithm CLARANS (Ng and Han, 2002) interprets the search space as a high-dimensional hypergraph, where each edge corresponds to swapping a medoid and non-medoid. On this graph it performs a randomized greedy exploration, where the first edge that reduces the loss TD is followed until no edge can be found with $p = 1.25\% \cdot k(n-k)$ attempts. In Section 3.6 we will outline how our approach can be used to explore k edges at a time efficiently; this will allow exploring a larger part of the search space in similar time, but we expect the savings to be relatively small compared to PAM.

Reynolds et al. (2006) discuss an interesting trick to speed up PAM. They show how to decompose the change in the loss function into two components, where the first depends only on the medoid removed, the second part only on the new point. This decomposition forms the base for our approach, and we will thus discuss it in Section 3 in more detail.

Park and Jun (2009) propose a "k-means like" algorithm for k-medoids (which actually was already considered by Reynolds et al. 2006 before), where in each iteration the medoid is chosen to be the object with the smallest distance sum to other members of the cluster, then each point is assigned to the nearest medoid until TD no longer decreases. This is, unfortunately, not very effective at improving the clustering: new medoids are only chosen from within the cluster, and *have* to cover the entire current cluster. This misses many improvements where cluster members can be reassigned to *other* clusters with little cost; such improvements are, however, found by PAM. Furthermore, the means used in k-means change with every point we move to a different cluster, but the medoids will very often remain the same, they are too coarse for this optimization strategy. In our experiments this approach produced noticeably worse results than PAM, in line with the earlier observations by Reynolds et al. (2006). The paper also contributes an $O(n^2)$ initialization, that unfortunately tends to choose all initial medoids close to the center of the data set. Choosing cluster medoids with a k-means like strategy will take $O(n^2)$ time because we have to assume the clusters to be unbalanced, and contain up to $O(n)$ objects; nevertheless this is k times faster than PAM. But because it does *not* consider reassigning points to other clusters when choosing the new medoid, this approach will miss many improvements found by PAM. This reduces the number of iterations, but also produces worse results.

Lijffijt et al. (2015) propose to use the k-means++ (Arthur and Vassilvitskii, 2007) initialization with PAM and CLARA, but give the resulting complexity incorrectly, missing a factor of k. They do not study the effect of replacing the initialization, which requires more iterations to converge (and hence, will be slower with original PAM), as we will discuss later in Section 3.4, and demonstrate in Section 4.2.

Since PAM needs $O(n^2)$ memory for the distance matrix, it is not usable on big data. Therefore, people have proposed various approximations to PAM, such as CLARA and CLARANS discussed before. Yang and Lian (2014) parallelize the "k-means like" variant with map-reduce, parallelizing over the cluster in the reduce step. When cluster sizes vary substantially, this needs $O(n^2)$ memory in the reducer, and may yield next to no speedup in the worst case. CLARA can be trivially parallelized by randomly partitioning the data, then running PAM on each partition (Kaufman et al., 1988). This approach will obviously benefit from our improvements the same way as CLARA and PAM benefit. A recent example is PAMAE (Song et al., 2017), which essentially is CLARA with an additional refinement step: it draws random samples and runs any k-medoids approach on each; chooses the best medoids found, and refines them with a single iteration of a approximate parallel version of the "k-means like" update. Papers have rarely considered using large k values, although this makes sense in the context of data approximation, where you want to reduce the data set to k representative samples. Many of the attempts at distributing and parallelizing PAM employ PAM as a subroutine, and hence can trivially integrate our improvements.

3 Finding the Best Swap

We focus on improving the original PAM algorithm here, which is a commonly used subroutine even in the faster variants such as CLARA (which uses PAM as a subroutine, and hence directly benefits from any improvement to PAM). We also discuss how we can obtain similar improvements for CLARANS in Section 3.6.

The algorithm SWAP evaluates every swap of each medoid m_i with any non-medoid x_j. Recomputing the resulting TD using Equation 2 every time requires finding the nearest medoid for every point, which causes many redundant computations. Instead, PAM only computes the *change* in TD for each object x_o if we swap m_i with x_j separately:

$$\Delta TD = \sum_{x_o} \Delta(x_o, m_i, x_j) \tag{3}$$

In the function $\Delta(x_o, m_i, x_j)$ we can often detect when a point remains assigned to its current medoid (if $c_k \neq c_i$, and this distance is also smaller than the distance to x_j), and then immediately return 0. Because of space restrictions, we do not repeat the original "if" statements used in (Kaufman and Rousseeuw, 1990, Ch. 2), but instead condense them directly into the following equation:

$$\Delta(x_o, m_i, x_j) = \begin{cases} \min\{d(x_o, x_j), d_s(o)\} - d_n(o) & \text{if } i = \textit{nearest}(o) \\ \min\{d(x_o, x_j) - d_n(o), 0\} & \text{otherwise} \end{cases} \tag{4}$$

where $d_n(o)$ is the distance to the nearest medoid of o, and $d_s(o)$ is the distance to the second nearest medoid. Computing them on the fly would increase the cost of SWAP by a factor of $O(k)$, but we can cache these values, and only update them when performing a swap.

Reynolds et al. (2006) note that we can decompose ΔTD into: (i) the loss of removing medoid m_i, and assigning all of its cluster members to the next best alternative, which can be computed as $\sum_{o \in C_i} d_s(o) - d_n(o)$ (ii) the (negative) loss of adding the replacement medoid

Algorithm 3: FastPAM1: Improved SWAP algorithm

```
1  repeat
2      (ΔTD*, m*, x*) ← (0, null, null) ;                    // Empty best candidate storage
3      foreach x_j ∉ {m_1, ..., m_k} do
4          d_j ← d_nearest(x_j) ;                              // Distance to current medoid
5          ΔTD ← (−d_j, −d_j, ..., −d_j) ;                     // Loss change for making j a medoid
6          foreach x_o ≠ x_j do
7              d_oj ← d(x_o, x_j) ;                            // Distance to new medoid
8              (n, d_n, d_s) ← (nearest(o), d_nearest(o), d_second(o)) ;   // Cached values
9              ΔTD_n ← ΔTD_n + min{d_oj, d_s} − d_n ;          // Loss change for current
10             if d_oj < d_n then                              // Reassignment check
11                 foreach m_i ∈ {m_1, ..., m_k} \ m_n do
12                     ΔTD_i ← ΔTD_i + d_oj − d_n;             // Update loss change
13         i ← arg min ΔTD_i ;                                 // Choose best medoid i
14         if ΔTD_i < ΔTD* then (ΔTD*, m*, x*) ← (ΔTD_i, m_i, x_j) ;   // Remember best
15     break loop if ΔTD* ≥ 0;
16     swap roles of medoid m* and non-medoid x* ;
17     TD ← TD + ΔTD* ;
18 return TD, M, C;
```

x_j, and reassigning all objects closest to this new medoid. Since (i) does not depend on the choice of x_j, we can make the loop over all medoids m_i outermost, reassign all its points to the second nearest medoid (cache the distance to the now nearest neighbor), and compute the resulting loss. We then iterate over all non-medoids and compute the benefit of using them as the missing medoid instead. In the Δ function, we no longer have to consider the second nearest now (we virtually removed the old medoid already). The authors observed roughly a two-fold speedup using this approach, and so do we in our experiments.

Our approach is based on a similar idea of exploiting redundancy in these computations (by caching shared computations), but we instead move the loops over the medoids m_i into the *innermost* for loop. The reason for this is to further remove redundant computations. This becomes apparent when we realize that in Equation 4, the second case does not depend on the current medoid i. If we transform the second case into an if statement, we can often avoid to iterate over all k medoids.

3.1 Making PAM SWAP faster: FastPAM1

Algorithm 3 shows the improved SWAP algorithm. In lines 4–5 we compute the benefit of making x_j a medoid. As we do not yet decide which medoid to remove, we use an array of ΔTD for each possible medoid to replace. We can now for each point compute the benefit when removing its current medoid (line 9), or the benefit if the new medoid is closer than the current medoid (line 10), which corresponds to the two cases in Equation 4. The interesting property is now since the second case does not depend on i, we can replace the $\min$ statement with an if conditional *outside* of the loop in lines 10–12. After iterating over all points, we choose the best medoid, and remember the overall best swap. If we always prefer the smaller index i on ties, we choose *exactly the same* swap as the original PAM algorithm.

3.2 Benefits and Costs

If we assume that the new medoid is closest in $O(1/k)$ cases on average (this assumes a somewhat balanced cluster size distribution), then we can compute the change for all k

Algorithm 4: FastPAM2: SWAP with multiple candidates

```
1  repeat
2      foreach x_o do compute nearest(o), d_nearest(o), d_second(o) ;
3      ΔTD*, x* ← [0,...,0], [null,...,null] ;          // Empty best candidates array
4      foreach x_j ∉ {m_1,...,m_k} do
5          d_j ← d_nearest(x_j) ;                        // Distance to current medoid
6          ΔTD ← (−d_j, −d_j, ..., −d_j) ;               // Loss change for making j a medoid
7          foreach x_o ≠ x_j do
8              d_oj ← d(x_o, x_j) ;                      // Distance to new medoid
9              (n, d_n, d_s) ← (nearest(o), d_nearest(o), d_second(o)) ;   // Cached
10             ΔTD_n ← ΔTD_n + min{d_oj, d_s} − d_n ;    // Loss change for current
11             if d_oj < d_n then                        // Reassignment check
12                 foreach m_i ∈ {m_1,...,m_k} \ m_n do
13                     ΔTD_i ← ΔTD_i + d_oj − d_n;       // Update loss change
14         foreach i where ΔTD_i < ΔTD*_i do
15             (ΔTD*_i, x*_i) ← (ΔTD_i, x_j) ;           // Remember the best swap for i
16     break loop if min ΔTD* ≥ 0 ;                      // At least one improvement was found
17     while i ← arg min ΔTD* and ΔTD*_i < 0 do          // Execute all improvements
18         swap roles of medoid m_i and non-medoid x*_i ;
19         TD ← TD + ΔTD*_i ;
20         ΔTD*_i ← 0 ;                                  // Prevent further processing
21         foreach j where ΔTD*_j < 0 do                 // Recompute TD for remaining swaps
22             ΔTD ← 0 ;
23             foreach x_o ∉ {m_1,...,m_k} \ m_j do ΔTD ← ΔTD + Δ(x_o, m_j, x*_j) ;
24             if ΔTD ≤ τ · ΔTD*_j then ΔTD*_j ← ΔTD ;   // Tolerance check
25             else ΔTD*_j ← 0;                          // Skip otherwise
26 return TD, M, C;
```

medoids with $O(k \cdot 1/k) = O(1)$ effort, by saving the innermost loop in line 12. Therefore, we expect a typical speedup on the order of $O(k)$ compared to the original PAM SWAP (but it may be hard to guarantee this for any useful assumption on the data distribution; the worst case supposedly remains unaffected) at the slight cost of temporarily storing one ΔTD for each medoid m_i (compared to the cost of storing the distance matrix and the distances to the nearest and second nearest medoids, the cost of this is negligible).

3.3 Swapping Multiple Medoids: FastPAM2

A second technique to make this second stage of PAM faster is based on the following observation: PAM will always identify the *single* best swap, then restart search; whereas the classic k-means reassigns all points, and updates all means in each iteration. Choosing the best swap has the benefit that this makes the algorithm independent of the data order (Kaufman and Rousseeuw, 1990, Ch. 2) as long as there are no ties, and it also means we need to execute fewer swaps than if we would greedily perform any swap that yields an improvement (where we may end up replacing the same medoid several times).

But on the other hand, in particular for large k, we can assume that many clusters will be independent, and we could therefore update the medoids of these clusters in the same iteration. Naively assuming that each medoid would be swapped in each iteration, this would allow us to reduce the number of iterations by k.

Based on this observation, we propose to consider the best swap for *each* medoid, and not only the single best swap, i.e., perform up to k swaps. This is a fairly simple modification shown in Algorithm 4, as we can simply store an array of swap candidates $(\Delta TD^*_i, x^*_i)$ in line 3, storing one best candidate for each current medoid m_i, and update these in line 15. After evaluating all possible swaps, we find the best swap within these up to k candidates

(if we did not find a candidate, the algorithm has converged). We perform the best of these swaps in line 18, mark it as done. Then we have to recompute in line 23 for each remaining swap candidate if it still improves the clustering, otherwise the swap is not performed. At this point, in line 24, we explore two alternatives: (a) "strict" ($\tau = 1$): only swaps are executed that appear to be independent of the previous swaps (because their ΔTD has not changed) and (b) "greedy" ($\tau = 0$): execute any swap that still yields a benefit.

The benefits of this strategy are, unsurprisingly, much smaller than the first improvement. In early iterations we see multiple swaps being executed, but in the later iterations it is common that only few medoids change at all. Nevertheless, this simple modification does yield another measurable performance improvement. However—in contrast to the first improvement—this no longer guarantees to yield the exact same result (because the additional swaps may occasionally be worse than the swaps found when rescanning, or may be executed in a slightly different order). From a theoretical point of view, both the original PAM, and FastPAM2 perform a steepest descent optimization strategy; where PAM only permits descends consisting of a single swap, whereas FastPAM2 can perform multiple swaps at once as long as they use different medoids. Therefore, we argue that both are able to find results of equivalent quality.[2] In our experiments, even the "greedy" strategy would often find slightly better results than PAM, and faster.

3.4 Faster Initialization with Linear Approximative BUILD (LAB): FastPAM

With these optimizations to the SWAP algorithm, that reduce the run time from $O(k(n-k)^2)$ to $O((n-k)^2)$, the main bottleneck of PAM suddenly becomes the first algorithm, BUILD. In the experiments below on the plant species data at $k = 200$, using the R implementation, PAM would spend 99% of the run time in SWAP. With above optimizations this reduces to about 15%. About 16% is the time to compute the distance matrix, and 69% of the time is spent in BUILD. The run time of BUILD is in $O(kn^2)$, so for large k this is not unexpected to happen. But since we were able to make SWAP much faster, we can now afford to begin with slightly worse starting conditions, even if we need more iterations of SWAP afterwards.

A very elegant way of choosing starting conditions in k-means is known as k-means++ (Arthur and Vassilvitskii, 2007). The beautiful idea of this approach is to choose additional seeds with the probability proportional to their distance to the nearest seed (the first seed is picked uniformly). If we assume there is a cluster of points and no seed nearby, the probability mass of this cluster is substantial, and we are likely to place the next seed there; afterwards the probability mass of this cluster reduces. Furthermore, this initialization is (in expectation) $O(\log k)$ competitive to the optimal solution, so it will theoretically generate good starting conditions. But as seen in our experiments, this guarantee is pretty loose; and BUILD empirically produces much better starting conditions than k-means++ (we are not aware of a detailed theoretical analysis). But it is easy to see that in BUILD each medoid is chosen as a current optimum with respect to TD; whereas k-means++ picks the first point randomly, and subsequent points are (in expectation) random points from different clusters, but k-means++ makes no effort to find the medoid (which is not that important for seeding k-means, where the mean is likely in between data points anyway). Therefore, we must expect that with k-means++ we need around k swaps just to pick the medoid of each cluster (and hence, k iterations of original PAM SWAP, although much fewer with FastPAM2, experimentally confirmed later in Section 4.2). Because a single iteration of swap used to

[2] Neither can guarantee to find the global optimum, which would be NP-hard.

Algorithm 5: FastPAM LAB: Linear Approximate BUILD initialization.

```
(TD, m_1) ← (∞, null);
S ← subsample of size 10 + ⌈√n⌉ from X ;                                  // Subsample
foreach x_j ∈ S do                                                        // First medoid
    TD_j ← 0 ;
    foreach x_o ∈ S ∧ x_o ≠ x_j do TD_j ← TD_j + d(x_o, x_j);
    if TD_j < TD then (TD, m_1) ← (TD_j, x_j);                            // Smallest distance sum
for i = 1 ... k − 1 do                                                    // Other medoids
    (ΔTD*, x*) ← (∞, null);
    S ← subsample of size 10 + ⌈√n⌉ from X \ {m_1, ..., m_i} ;            // Subsample
    foreach x_j ∈ S do
        ΔTD ← 0 ;
        foreach x_o ∈ S ∧ x_o ≠ x_j do
            δ ← d(x_o, x_j) − min_{o∈m_1,...,m_i} d(x_o, o);
            if δ < 0 then ΔTD ← ΔTD + δ;
        if ΔTD < ΔTD* then (ΔTD*, x*) ← (ΔTD, x_j);                       // best reduction in TD
    (TD, m_{i+1}) ← (TD + ΔTD*, x*);
return TD, {m_1, ..., m_k};
```

take as much time as BUILD, the k-means++ initialization only begins to shine if we use FastPAM1 to reduce the cost of iterating together with the FastPAM2 strategy of doing as many swaps as possible in each iteration. Lijffijt et al. (2015) previously proposed to use k-means++ for PAM and CLARA; but their complexity analysis misses a factor of k for SWAP, and their experiments also only used small k. Our experiments (in Section 4.2) show that k-means++ initialization takes *many more iterations* to converge than with the original BUILD initialization; so without the improvements introduced in this article, it is usually not beneficial to use k-means++ with the original SWAP algorithm for speed (the benefit of randomness, the ability to get different results, remains; and so do the theoretical guarantees).

We originally experimented with k-means++ initialization, but eventually settled for a different strategy we call LAB (Linear Approximative BUILD). What we title "FastPAM" then is the combination of LAB with the optimizations of FastPAM2. As the name indicates, LAB is a linear approximation of the original PAM BUILD (c.f., Algorithm 1). In order to achieve linear runtime in n, we simply subsample the data set. Before choosing each medoid, we sample $10 + \lceil\sqrt{n}\rceil$ points from all non-medoid points. From this subsample we choose the one with the largest decrease ΔTD with respect to the current subsample only. Results were slightly better with sampling k times, and not just once; since each object has k chances of being in the sample, and if we draw a bad sample it only affects a single medoid. A pseudocode of LAB is given as Algorithm 5.

Clearly, this algorithm reduces the runtime of BUILD to $O(kn)$. In the experiments in Section 4.2, the results with LAB were significantly better than with k-means++.

3.5 Integration in CLARA: FastCLARA

Since CLARA (Kaufman and Rousseeuw 1986, 1990, Ch. 3) uses PAM as a subroutine, we can trivially use our improved FastPAM with CLARA. In the experiments (and the implementations provided as open-source) we denote this variant as FastCLARA.

3.6 Wider Exploration in CLARANS: FastCLARANS

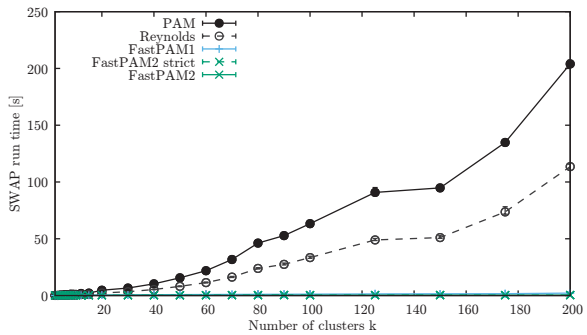
CLARANS (Ng and Han, 2002) uses a randomized search instead of considering all possible swaps. For this, it chooses a random pair of a non-medoid object and a medoid, computes whether this improves the current loss, and then greedily performs this swap. Adapting the idea from FastPAM1 to the random exploration approach of CLARANS, we pick only the non-medoid object at random, but consider all medoids for swapping at a similar cost to looking at a single medoid. This means we can either explore k times as many edges of the graph, or we can reduce the number of samples to draw by a factor of k. In our experiments we opted for the second choice, to make the results comparable to the original CLARANS in the number of edges considered; but as the edges chosen involve the same non-medoids, we expect a slight loss in quality that should be easily countered by increasing the subsampling rate of non-medoids. By varying the subsampling rate parameter, the user can easily control the tradeoff between computation time and exploration.

4 Experiments

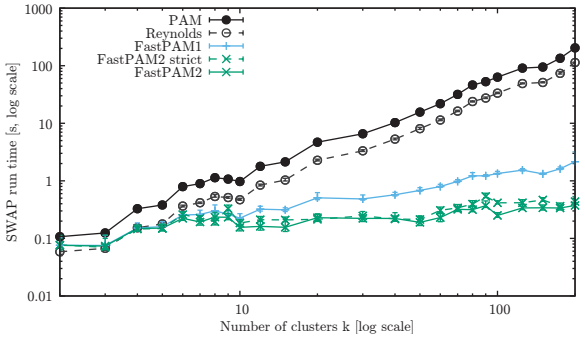
Theoretical considerations show that we must expect an $O(k)$ speedup of FastPAM1 over the original PAM algorithm, so our experiments primarily need to verify that there is no trivial error (in contrast to much work published in recent years, the speedup is not just empirical). Nevertheless constant factors and implementation details can make a big difference (Kriegel et al., 2017), and we want to ensure that we do not pay big overheads for theoretical gains that would only manifest for infinite data.[3] Because of constant factors, it could for example be possible that we need a certain minimum k for this approach to be beneficial over the original PAM. For FastPAM2 we do not have such a theoretical argument for an additional speedup over FastPAM1; and the speedup is expected to be a small factor due to the reduction in iterations necessary. In contrast to FastPAM1, it does not guarantee the exact same results; therefore we also want to verify that they are of equivalent quality. LAB, the third component for FastPAM, yields worse starting conditions. These should not affect the final result much, but will require additional iterations of SWAP. We observed increased runtimes when using k-means++ for PAM initialization, therefore it needs to be verified experimentally that LAB does not require excessive additional iterations.

As primary data set for our experiments, we use the "One-hundred plant species leaves" data set (texture features only) from the well-known UCI repository (Dheeru and Karra Taniskidou, 2017). We chose this data set because it has 100 classes, and 1600 instances, a fairly small size that PAM can still easily handle. Naively, one would expect that $k = 100$ is a good choice on this data set, but some leaf species are likely not distinguishable by unsupervised learning. We used the ELKI open-source data mining toolkit (Schubert et al., 2015) in Java to develop our version. For comparison, we also ported FastPAM2 to the R `cluster` package, which is based on the original PAM source code and written in C. The source codes of both versions will be contributed to these projects. Experiments were run on an Intel i7-7700 at 3.6 GHz with turbo boost disabled. We perform 25 runs, and plot the average, minimum and maximum. Both implementations show similar behavior, so we are confident that the results are not just due to implementation differences (Kriegel et al., 2017), and we verify the results on additional data sets.

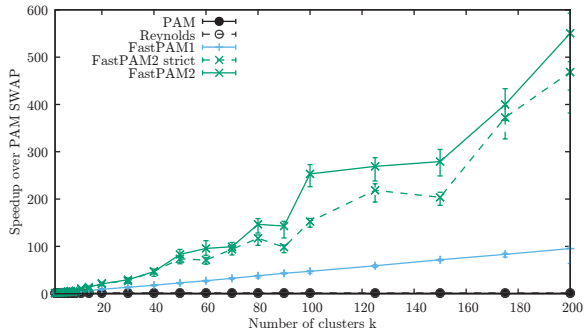
[3] Clearly, our $O(k)$ fold speedup must be immediately measurable, not just asymptotically, because the constant overhead for maintaining the fixed array cache is small.



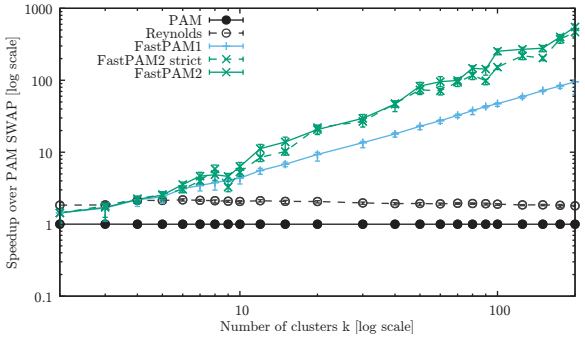
(a) Run time in linear space [ELKI]



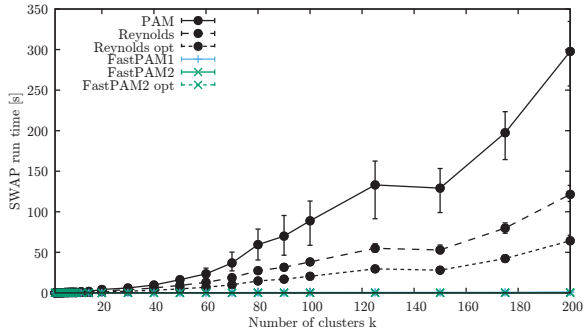
(b) Run time in log-log space [ELKI]



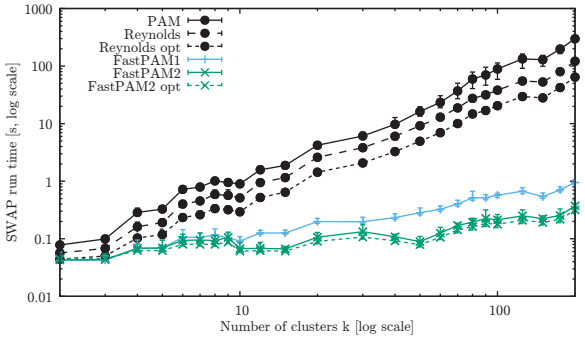
(c) Speedup in linear space [ELKI]



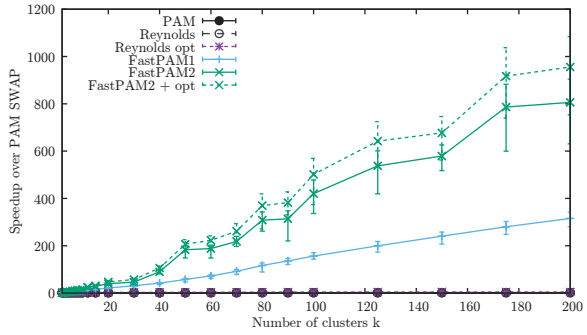
(d) Speedup in log-log space [ELKI]



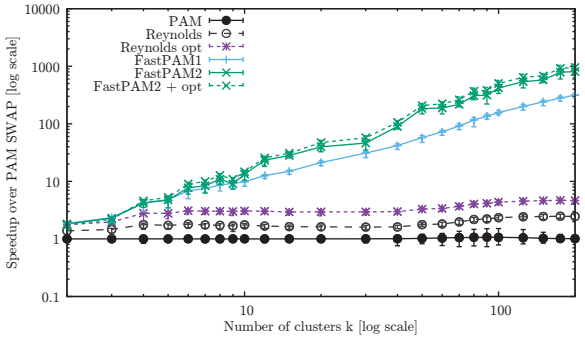
(e) Run time in linear space [R]



(f) Run time in log-log space [R]



(g) Speedup in linear space [R]

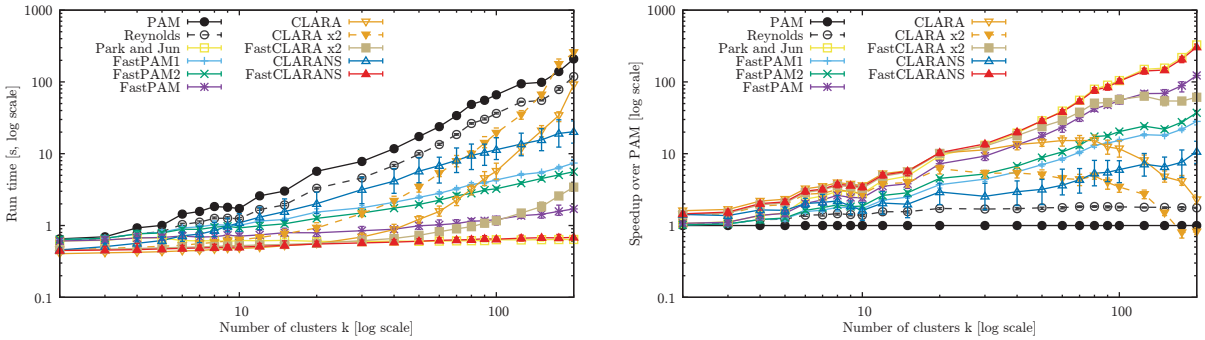


(h) Speedup in log-log space [R]

Fig. 1: Run time comparison of PAM SWAP (SWAP only, without DAISY, without BUILD)

4.1 Run Time Speedup

In Figure 1, we vary k from 2 to 200, and plot the run time of the PAM SWAP phase *only* (the cost of computing the distance matrix and the BUILD phase is not included), using the original PAM, the Reynolds version, as well as the proposed improvements. Figure 1a shows the run time in linear space, to visualize the drastic run time differences observed. Reynolds' was quite consistently two times faster than the original PAM; but our proposed methods were faster by a factor that grows approximately linearly with the number of clusters k. In



(a) Run time in log-log space [ELKI]

(b) Speedup in log-log space [ELKI]

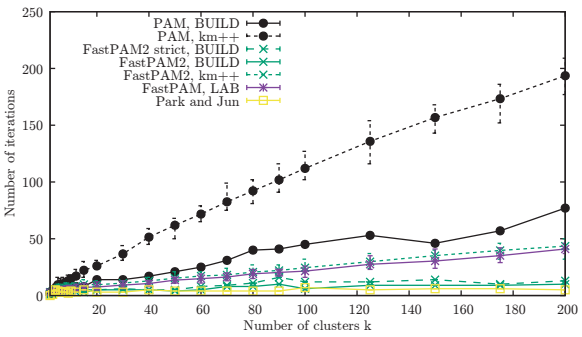
Fig. 2: Run time comparison of different variations and derived algorithms.

log-log-space, Figure 1b, we can differentiate the three variants studied. While the "greedy" variant is slightly faster than the "strict" variant of FastPAM2, the difference between these two is not very large compared to the main contribution of this paper. (Because this plot only includes the SWAP phase, LAB is not used here, all methods are initialized with BUILD.)

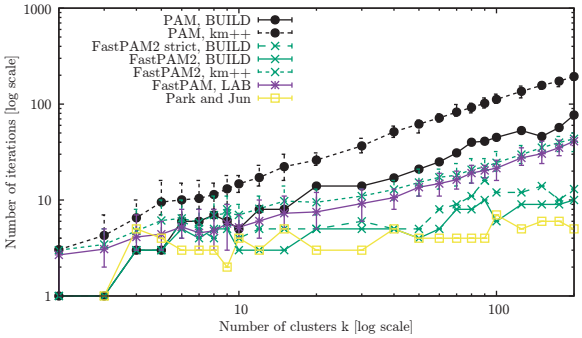
In Figure 1c we plot the speedup over PAM. Reynolds' SWAP clearly was about twice as fast as the original PAM. The FastPAM1 improvement gives an empirical speedup factor of about $\frac{1}{2}k$, while the second improvement contributed an additional speedup of about 2-2.5$\times$ by reducing the number of iterations. Because of the multiplicative effect of these savings, the linear plot in Figure 1c gives the false impression that this second contribution yields the larger benefit. The logspace plot in Figure 1d more accurately reflects the contribution of the two factors, resulting in a speedup of over 250 times at $k = 150$; while at $k = 2$ and $k = 3$ the speedup was just $1.4\times$ respectively $1.75\times$, and less than our implementation of Reynolds (in R, as seen in Figure 1h, the difference at $k = 2, 3$ is negligible; so this is probably only an implementation difference). In Figure 1e to Figure 1h we provide the results using the R implementation, which clearly exhibit similar behavior. We only implemented the "greedy" $\tau = 0$ version in R; but we additionally include versions of Reynolds (`pamonce=2` of the existing package) and of our approach that optimize the traversal of the distance matrix. In the most extreme case tested, a speedup of about $1000\times$ at $k = 200$ is measured – but since the speedup is expected to depend on $O(k)$, the exact values are meaningless, furthermore, we excluded the distance matrix computation and initialization in this first experiment.

In Figure 2, we study the run time of approximations to PAM (including the distance matrix computation and initialization time now). We only present the log-log space plots, because of the extreme differences.

The run time of CLARA, as k increases, approaches the run time of PAM. This is expected, because the subsample size for CLARA is chosen as $40 + 2k$, and necessary because the subsample size needs to be sufficiently larger than k (the recommendation of Lijffijt et al. 2015 of always using 80 samples is inappropriate for larger k). For CLARA x2 we also evaluate doubling this value to $80 + 4k$, and we also double the number of restarts from 5 to 10. CLARA x2 is thus expected to take 8 times longer than CLARA, but should give better results. FastCLARA is CLARA using our FastPAM approach, and performs much better, but for large k also eventually becomes slower than FastPAM. The run time of CLARANS on this data set (see later for CLARANS problems) is in between the original PAM and CLARA, and with our optimizations FastCLARANS becomes the fastest method tested (at similar quality to CLARANS, and with the same problems). Park and Jun's (2009) approach is similarly fast to FastCLARANS for large k, but its quality is quite poor, as we will see and discuss in Section 4.3.

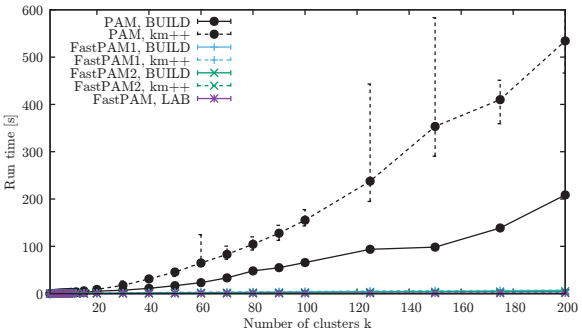


(a) Iterations in linear space [ELKI]

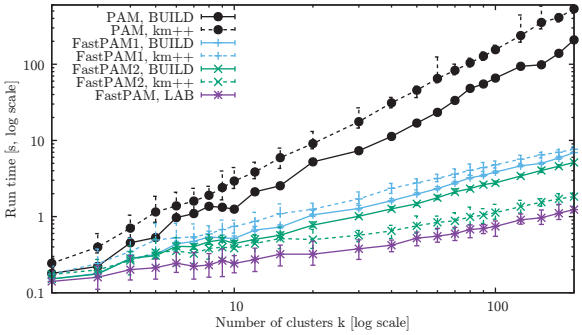


(b) Iterations in log-log space [ELKI]

Fig. 3: Number of iterations for PAM vs. FastPAM2 and BUILD vs. LAB initialization



(a) Runtime with BUILD vs. k-means++ initialization in linear space [ELKI]



(b) Runtime with BUILD vs. k-means++ initialization in log-log space [ELKI]

Fig. 4: Runtime impact of k-means++ and LAB initialization

4.2 Number of Iterations

We are not aware of theoretical results on the number of iterations needed for PAM. Based on results for k-means, we must assume that the worst case is superpolynomial like k-means (Arthur and Vassilvitskii, 2006), albeit in practice a "few" iterations are usually enough. Because of this, we are also interested in studying the number of iterations, depending on the choice of k and the initialization method.

Figure 3 shows the number of iterations needed with different methods, both in linear space and log space. In line with previous empirical results, only few iterations are necessary. Because PAM only performs the best swap in each iteration, a linear dependency on k is to be assumed; interestingly enough we usually observed much less than k iterations, so many medoids remain unchanged from their initial values (note that this may be due to the rather small data set size, too). The k-means++ initialization required roughly 2-4$\times$ as many iterations for PAM; with the original algorithm where each iteration would cost about as much as the BUILD initialization, this choice (although suggested by Lijffijt et al. 2015) is detrimental even for small k. With the improvements of this paper, these additional iterations are cheaper than the rather slow BUILD initialization by a factor of $O(k)$ now, hence we can now begin with a worse but cheaper starting point. Furthermore, the FastPAM2 greedy approach which performs up to k swaps in each iteration does reduce the number of iterations substantially (the "greedy" version requires slightly fewer iterations than the "strict" version, as expected). FastPAM2 with BUILD performed the second-lowest numbers of iterations. Our proposed LAB initialization of FastPAM saves a few extra iterations compared to the k-means++ strategy, at better initial quality, and hence is measurably faster in the end. Park and Jun (2009) at first seems to perform very well in this figure, with slightly fewer iterations than

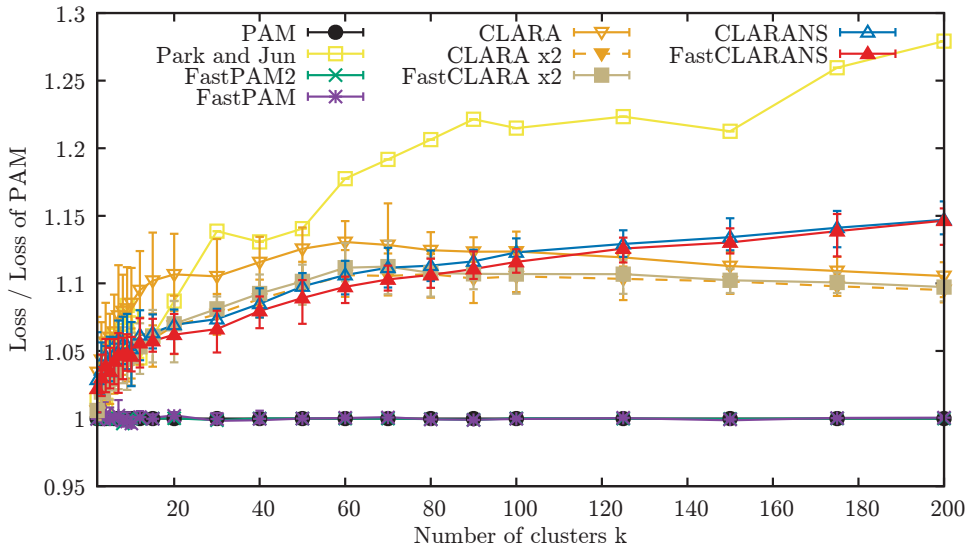


Fig. 5: Loss (TD) compared to PAM

FastPAM2 with BUILD. Unfortunately, this is because the "k-means style" algorithm misses many possible improvements to the clustering, and hence produces much worse results as we will observe in the next experiments.

In Figure 4 we revisit the runtime experiment, and focus on initialization. As we can see, the increased number of iterations hurts runtime with the original PAM algorithm as well as its Reynolds variant substantially (the reasons for this are explained in Section 3.4); for FastPAM1, the use of k-means++ only comes at a small performance penalty (while it still needs as many iterations as the original PAM, these have become $O(k)$ times faster, and the initialization cost begins to matter much more), and with FastPAM2's ability to perform multiple swaps per iteration, a linear-time initialization such as k-means++ or the proposed LAB clearly becomes the preferred initialization method, in particular for large k.

4.3 Quality

Any algorithmic change and optimization comes at the risk of breaking some things, or negatively affecting numerics (see, e.g., Schubert and Gertz 2018 on how common numerical issues are even with basic statistics such as variance in SQL databases). In order to check for such issues, we made sure that our implementations pass the same unit tests as the other algorithms in both ELKI and R. We do not expect numerical problems, and Reynolds' variant and FastPAM1 are supposed to give the same result (and do so in the experiments, so we exclude them from the plot). The FastPAM2 algorithm is greedy in performing swaps, and may therefore converge to a different solution, but that should be of the same quality, which we will verify now.

In Figure 5 we visualize the loss, i.e., the objective TD of Equation 2, of different approximations compared to the solution found by the original PAM approach (which is not necessarily the global minimum). For large k, the solution found by the approach of Park and Jun (2009) is over 25% worse here, for the reasons discussed before (worse initialization, misses many improvements). Our strategy FastPAM2 even with the "greedy" approach gives results comparable to PAM as expected (sometimes slightly better, sometimes slightly worse). The cheaper LAB initialization (full FastPAM) does not cause a noticeable loss in quality either, but further improves the total run time. CLARA (which only uses a subsample of the data) finds considerably worse results. By doubling the subsample size to $80 + 4k$ and doing

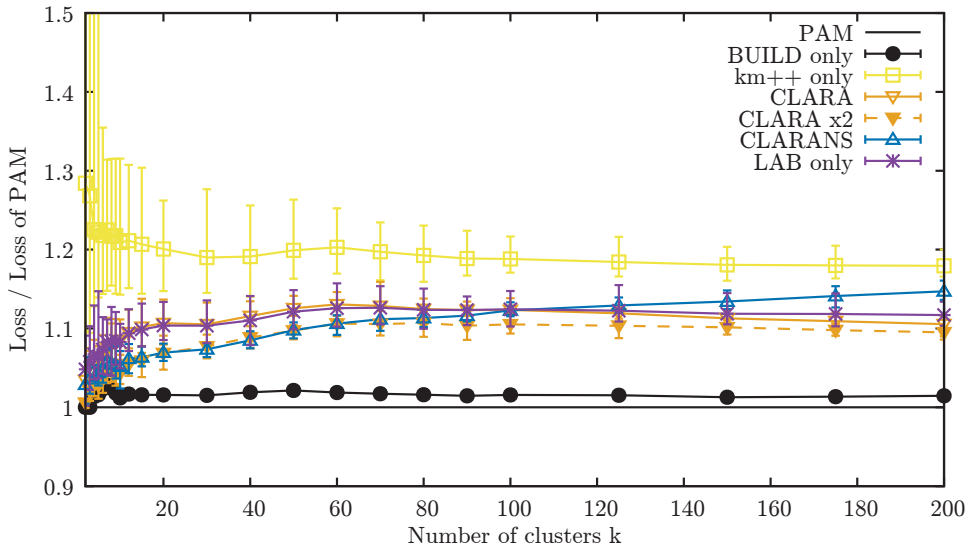


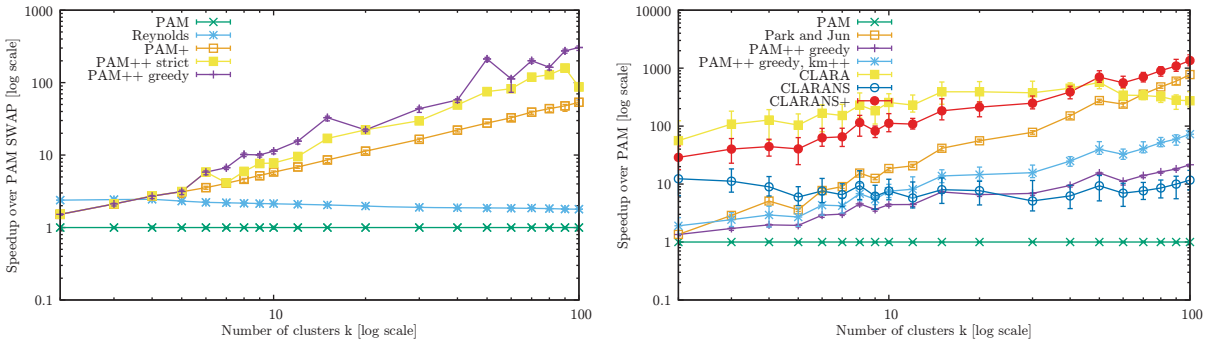
Fig. 6: Loss (TD) of k-means++ vs. BUILD initialization compared to PAM

twice as many restarts (CLARA x2) the results only improve slightly for large k (but much more for small k). CLARA x2 is until about $k = 70$ as good as CLARANS here, but faster; for larger k it becomes even better than CLARANS, but also slower. FastCLARA has the same quality as CLARA x2 (we use the x2 parameters, too), but it was much faster. FastCLARANS is slightly better than CLARANS, and was considerably faster. All the CLARANS results degrade with increasing k, so it may become necessary to increase the subsample size there, which will increase the run time (it is up to the user to choose his preferences, quality or runtime). In conclusion, all our "Fast" approaches perform as well as their older counterparts, but are $O(k)$ times faster.

In Figure 6, we evaluate the quality of LAB, k-means++, and BUILD initialization compared to the converged PAM result. As seen in the previous experiments, all three initializations will yield similar results after PAM, but we can compare the quality of the initial medoids to the full PAM result. As we can see, the BUILD approach produces the best initial results (and as noted by Kaufman and Rousseeuw 1987, 1990, Ch. 2, the BUILD result may be usable without further refinement). While k-means++ offers some theoretical advantages (c.f., Section 3.4), the initial result is quite bad as this strategy only attempts to pick a random point from each cluster, and not the medoids. Our proposed LAB initialization is in between k-means++ and BUILD, and by itself performs similar to CLARA. As it only considers a subset of the data, its medoids will be worse than BUILD; but because it chooses the best medoid of the sample it performs better than k-means++. Because it reduces the runtime for $O(n^2k)$ to $O(nk)$ it is the preferred choice for FastPAM nevertheless.

4.4 Second Dataset

We also verified our results on the "Optical Recognition of Handwritten Digits" data set from UCI (Dheeru and Karra Taniskidou, 2017) with $n = 5620$ instances, $d = 64$ variables, and 10 natural classes. An example of the results is shown in Figure 7 and also show an $O(k)$ speedup compared to PAM (we observe a $1000\times$ speedup at $k = 200$, but the more reasonable choice of k would be 10 here, where the speedup is only about $10\times$). If a loss in quality is acceptable, FastCLARA and FastCLARANS again are interesting alternatives outperforming their non-fast versions. Clearly, the benefits on this data set are similar, and support our theoretical analysis.



(a) SWAP Speedup [log-log]

(b) Overall speedup [log-log]

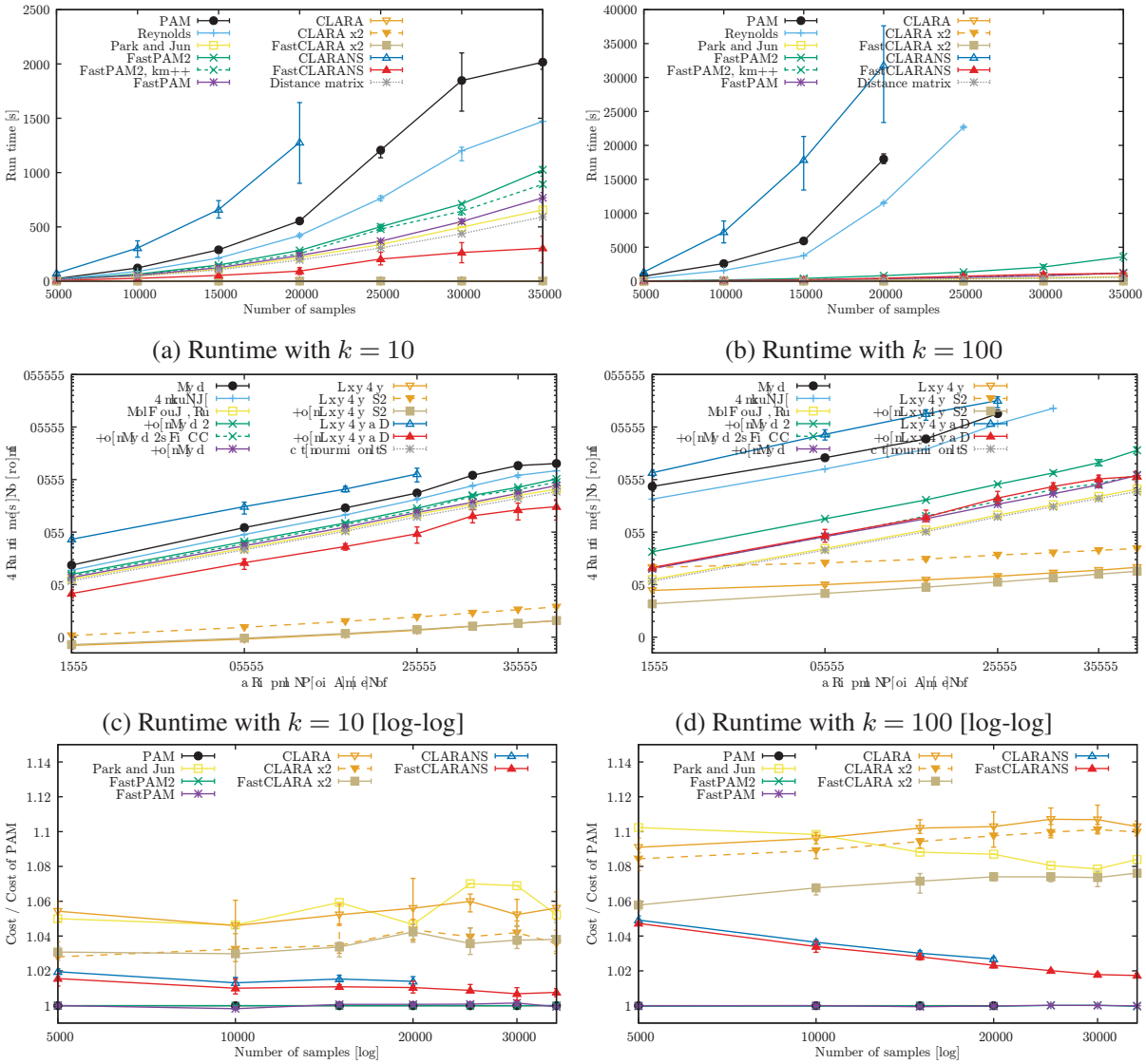
Fig. 7: Results on Optical Digits data using ELKI

4.5 Scalability Experiments

Just as PAM, our method also requires the entire distance matrix to be precomputed. This will require $O(n^2)$ time and memory, making the method as-is unsuitable for big data (for real big data problems, it will however often be good enough to cluster a sample that still fits into memory – for example with a mean, the precision only improves with $\sqrt{n}$, so adding more data eventually does barely improve the results). Our improvements focus on reducing the dependency on k, but we nevertheless experimented with scalability in n, too (and we already included FastCLARA and FastCLARANS in the previous experiments). The behavior of the PAM variants is as expected $O(n^2)$, but we see nevertheless quite big differences between PAM, FastPAM, and sampling-based approaches. In this experiment, we use the well-known MNIST data set from the UCI repository (Dheeru and Karra Taniskidou, 2017), which has 784 variables (each corresponding to a pixel in a 28×28 grid) and 60.000 instances. We used the first $n = 5000, 10000, \ldots, 35000$ instances with a time limit of 6 hours and compare $k = 10$ and $k = 100$. The high number of variables makes this data set expensive for CLARANS, because it computes distances more than once.

The problem of quadratic increase in runtime is best seen in the linear scale plots Figure 8a and Figure 8b. As a reference, we also give the time needed just for computing the distance matrix as dotted line, which is also quadratic. Except for CLARA and FastCLARANS, the runtime is dominated by computing the distance matrix (and hence CLARA, which uses a constant-size sample independent of n, shines for large n). The original CLARANS suffers from excessive distance re-computations. The authors assumed that distances are cheap to compute, and noted that it may be necessary to cache the distances in one way or another. FastCLARANS reduces the number of distance computations of CLARANS by a factor of $O(k)$, and is still cheaper than the full distance matrix here. For more expensive distances such as dynamic time warping, FastPAM will outperform FastCLARANS, and it will almost always give better results. For $k = 10$, only CLARANS, PAM and Reynolds' variant are problematic at this data size, but at $k = 100$ the benefits of our improvements become very noticeable. The CLARA methods are squeezed to the axis in the linear plot, and hence we also provide log-log plots in Figure 8c for $k = 10$ and Figure 8d for $k = 100$. For $k = 10$, the lines of CLARA and FastCLARA x2 almost coincide by chance (note that FastCLARA x2 produces a result comparable to the slower CLARA x2 method; expected to be 8 times slower), but at $k = 100$ it is faster than CLARA, demonstrating that our improvements also accelerate CLARA by a factor of $O(k)$.

While the scalability in n is quadratic as expected, we observe that if you can afford to compute the pairwise distance matrix, then you will *now* also be able to run FastPAM. For $k = 10$, the additional runtime of FastPAM was about 30% the runtime of computing



(a) Runtime with $k = 10$

(b) Runtime with $k = 100$

(c) Runtime with $k = 10$ [log-log]

(d) Runtime with $k = 100$ [log-log]

(e) Quality compared to PAM $k = 10$

(f) Quality compared to PAM $k = 100$

Fig. 8: Results on MNIST data using ELKI

the distance matrix computation, and at $k = 100$ FastPAM took about as much time as the distance matrix. Hence, if you can compute the distance matrix, you can also run FastPAM for reasonable values of $k \ll n$, and the main scalability problem will often be the memory consumption of the distance matrix. Without our optimizations, the cost of PAM would have been many times higher.

If computing the distance matrix is prohibitively expensive, it may still be possible to use FastCLARA (CLARA with our improved FastPAM on the individual samples); which is $O(k)$ times faster than original CLARA, and will scale linearly in n. But as seen in Figure 8e and Figure 8f, CLARA will usually give worse results (about 10% in our experiments). For many users this difference will be acceptable, as a clustering result is never "perfect". For large data sets, FastCLARANS will usually give better results, unless the sample size of CLARA is increased considerably. But on the other hand, FastCLARANS is only advisable for inexpensive distance functions such as (low-dimensional) Euclidean distance, and would require using a non-trivial distance cache for good performance otherwise.

5 Conclusions

In this article we proposed a modification of the popular PAM algorithm that typically yields an $O(k)$ fold speedup, by clever caching of partial results in order to avoid recomputation. This caching was enabled by changing the nesting order of the loops in the algorithm, showing once more how much seemingly minor looking implementation details can matter (Kriegel et al., 2017). As a second improvement, we propose to find the best swap for each medoid, and execute as many as possible in each iteration, which reduces the number of iterations needed for convergence without loss of quality, as demonstrated in the experiments, and as supported by theoretical considerations.

The surprisingly large speedups obtained with this approach enable the use of this classic clustering method on much larger data, in particular with large k. Even such seemingly minor changes in such an algorithm can make a big difference. It is hard to devise such things on the drawing board – such solutions more naturally arise when trying to low-level optimize the code, such as when and when not to allocate memory for buffers, and trying to avoid recomputing the same values repeatedly. Today's compilers are reasonably good at performing local optimization (at least when it does not affect numerical precision, Schubert and Gertz 2018), but will not introduce an additional array to cache such values. With the faster refinement procedure, it now pays off to use cheaper initialization methods with PAM. We propose LAB initialization, a linear-time approximation of the original PAM BUILD algorithm.

Methods based on PAM, such as CLARA, CLARANS, and the many parallel and distributed variants of these algorithms for big data, all benefit from this improvement, as they either use PAM as a subroutine (CLARA), or employ a similar swapping method (CLARANS) that can be modified accordingly as seen in Section 3.6.

The proposed methods are included in the open-source ELKI (Schubert et al., 2015) framework in version 0.7.5 (Schubert and Zimek, 2019), and FastPAM2 is included in the R `cluster` package version 2.0.9 (LAB, FastCLARA, and FastCLARANS are, however, not implemented for R yet, only in ELKI), to make it easy for others to benefit from these improvements. With the availability in two major clustering tools, we hope that many users will find using PAM, CLARA, CLARANS, and later derived methods, possible on much larger data sets with higher k than before.

References

Arthur D, Vassilvitskii S (2006) How slow is the *k*-means method? In: ACM Symposium on Computational Geometry, pp 144–153, DOI 10.1145/1137856.1137880

Arthur D, Vassilvitskii S (2007) k-means++: the advantages of careful seeding. In: ACM-SIAM SODA, pp 1027–1035

Bock H (2007) Clustering methods: A history of k-means algorithms. In: Brito P, Cucumel G, Bertrand P, Carvalho F (eds) Selected Contributions in Data Analysis and Classification, Springer, pp 161–172, DOI 10.1007/978-3-540-73560-1_15

Bradley PS, Mangasarian OL, Street WN (1996) Clustering via concave minimization. In: NIPS, pp 368–374

Dheeru D, Karra Taniskidou E (2017) UCI machine learning repository. URL `http://archive.ics.uci.edu/ml`

Ester M, Kriegel H, Sander J, Xu X (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. In: KDD, pp 226–231

Estivill-Castro V (2002) Why so many clustering algorithms: a position paper. SIGKDD Explorations 4(1):65–75

Fritz H, Filzmoser P, Croux C (2012) A comparison of algorithms for the multivariate l_1-median. Computational Statistics 27(3):393–410, DOI 10.1007/s00180-011-0262-4

Hartigan JA, Wong MA (1979) Algorithm as 136: A k-means clustering algorithm. Journal of the Royal Statistical Society Series C (Applied Statistics) 28(1):100–108

Kaufman L, Rousseeuw PJ (1986) Clustering large data sets. In: Pattern Recognition in Practice, Elsevier, pp 425–437, DOI 10.1016/b978-0-444-87877-9.50039-x

Kaufman L, Rousseeuw PJ (1987) Clustering by means of medoids. In: Dodge Y (ed) Statistical Data Analysis Based on the L_1 Norm and Related Methods, pp 405–416

Kaufman L, Rousseeuw PJ (1990) Finding Groups in Data: An Introduction to Cluster Analysis. John Wiley&Sons, DOI 10.1002/9780470316801

Kaufman L, Hopke PK, Rousseeuw P (1988) Using a parallel computer system for statistical resampling methods. Computational Statistics Quarterly 2:129–141

Kriegel H, Schubert E, Zimek A (2017) The (black) art of runtime evaluation: Are we comparing algorithms or implementations? Knowl Inf Syst 52(2):341–378, DOI 10.1007/s10115-016-1004-2

Lijffijt J, Papapetrou P, Puolamäki K (2015) Size matters: choosing the most informative set of window lengths for mining patterns in event sequences. Data Min Knowl Discov 29(6):1838–1864, DOI 10.1007/s10618-014-0397-3

Lucasius C, Dane A, Kateman G (1993) On k-medoid clustering of large data sets with the aid of a genetic algorithm: background, feasiblity and comparison. Analytica Chimica Acta 282(3):647 – 669, DOI 10.1016/0003-2670(93)80130-D

Ng RT, Han J (2002) CLARANS: A method for clustering objects for spatial data mining. IEEE TKDE 14(5):1003–1016, DOI 10.1109/TKDE.2002.1033770

Overton ML (1983) A quadratically convergent method for minimizing a sum of euclidean norms. Math Program 27(1):34–63, DOI 10.1007/BF02591963

Park H, Jun C (2009) A simple and fast algorithm for k-medoids clustering. Expert Syst Appl 36(2):3336–3341, DOI 10.1016/j.eswa.2008.01.039

Reynolds AP, Richards G, de la Iglesia B, Rayward-Smith VJ (2006) Clustering rules: A comparison of partitioning and hierarchical clustering algorithms. J Math Model Algorithms 5(4):475–504, DOI 10.1007/s10852-005-9022-1

Schubert E, Gertz M (2018) Numerically stable parallel computation of (co-)variance. In: SSDBM, pp 10:1–10:12, DOI 10.1145/3221269.3223036

Schubert E, Zimek A (2019) ELKI: A large open-source library for data analysis - ELKI release 0.7.5 "Heidelberg". arXiv:1902.03616

Schubert E, Koos A, Emrich T, Züfle A, Schmid KA, Zimek A (2015) A framework for clustering uncertain data. PVLDB 8(12):1976–1979, DOI 10.14778/2824032.2824115

Schubert E, Sander J, Ester M, Kriegel H, Xu X (2017) DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. ACM Trans Database Syst 42(3):19:1–19:21

Schubert E, Hess S, Morik K (2018) The relationship of DBSCAN to matrix factorization and spectral clustering. In: LWDA, CEUR Workshop Proceedings, vol 2191, pp 330–334

Song H, Lee J, Han W (2017) PAMAE: parallel *k*-medoids clustering with high accuracy and efficiency. In: KDD, pp 1087–1096, DOI 10.1145/3097983.3098098

Wei C, Lee Y, Hsu C (2003) Empirical comparison of fast partitioning-based clustering algorithms for large data sets. Expert Syst Appl 24(4):351–363, DOI 10.1016/S0957-4174(02)00185-9

Yang X, Lian L (2014) A new data mining algorithm based on MapReduce and hadoop. IJSIP 7(2):131–142, DOI 10.14257/ijsip.2014.7.2.13