Resilient K-d Trees: K-Means in Space Revisited

Fabian Gieseke
Technische Universität Dortmund
Dortmund, Germany
fabian.gieseke@cs.tu-dortmund.de

Gabriel Moruz*
Goethe University Frankfurt am Main
Frankfurt am Main, Germany
gabi@cs.uni-frankfurt.de

Jan Vahrenhold
Technische Universität Dortmund
Dortmund, Germany
jan.vahrenhold@cs.tu-dortmund.de

***Abstract*—We develop a k-d tree variant that is resilient to a pre-described number of memory corruptions while still using only linear space. We show how to use this data structure in the context of clustering in high-radiation environments and demonstrate that our approach leads to a significantly higher resiliency rate compared to previous results.**

I. Introduction

Onboard data analysis on spacecrafts has become more and more important in recent years. For instance, the EO-1 Earth Orbiter already makes use of automatic classification approaches to analyze hyperspectral image data such that additional follow-up observations of interesting regions can be made [1]. Onboard autonomy will be an important issue for future missions in general and in particular for those involving large one-way communication times like Jupiter and Saturn missions [2], [3]. Hence, machine learning algorithms like clustering or classification seem to have the potential to dramatically reduce unnecessary idle times by prioritizing data and/or by creating bandwidth-saving compressed "previews" for transmission to earth.

Spacecraft systems differ significantly from standard PCs in terms of computational power, memory, storage space, power consumption, and operating systems. Also, these systems operate in high-radiation environments with interference caused by cosmic rays and alpha particles [2], [4]. Both influences can severely affect the computations onboard these spacecrafts by memory corruptions (*bit-flips*) in the various layers of the systems' memory hierarchy. For modern (spacecraft) systems, the influence of memory corruptions in the CPU (i.e., registers and cache) is addressed by using fully radiation-hardened hardware components [2]. For the remaining parts of the memory hierarchies, software (e.g., bit-encoding schemes) and/or hardware-based concepts are used to reduce the effect of corruptions. However, both approaches lead to higher costs, an increase of mass and/or a reduction in capability and speed in most cases [2].

An alternative approach are *resilient* algorithms which can guarantee reasonable outputs in the presence of memory corruptions and exhibit only a small overhead in space and runtime. These algorithms are usually analyzed in the *faulty-memory RAM model* [5] which extends the RAM model in the sense that memory cells may get corrupted.

Recently, the effect of radiation on the well-known k-means clustering approach along with two of its variants was investigated by Wagstaff and Bornstein [2], [3]. Their results indicate that the k-d tree-based variant, which is superior in an radiation-free environment, exhibits an inferior clustering performance compared to the standard approach when the data cells are exposed to memory corruptions. In this paper, we propose a resilient version of the classical k-d tree data structure and show how to use it to obtain a resilient k-means variant. Our experiments indicate that the resulting clustering approach has a superior clustering performance for real-world data compared to the standard k-means approaches even in the presence of massive memory corruptions.

II. Background

A. K-Means Clustering and K-d Tree-Based Filtering

The k-means clustering approach is a local search strategy which aims at finding an assignment of all input points to a predefined number k of clusters [6]. The algorithm maintains a set of k candidate centers as the mean of the input points closest to them and iterates adjusting the centers until the algorithm converges. After initializing the set of candidate centers by k points randomly sampled from the input, the algorithm proceeds in iterations. In each iteration, every point is assigned to its nearest candidate center, and at the end of the iteration each candidate center is updated to the mean of all points assigned to it. One of the bottlenecks of a straightforward k-means implementation is the recurrent computation of nearest neighbors. We build upon Kanungo *et al.*'s [7] k-d k-means algorithm which first builds a k-d tree for the input and uses it to find nearest neighbors.

K-d Trees: A k-d tree [8] is a binary search tree for a set of multidimensional points.[1] Each internal node v in a k-d tree for a set of d-dimensional points corresponds to a d-dimensional box that contains all points stored in the subtree rooted at v. Splitting such boxes is done in a level-wise round-robin manner, i.e., for a node v on the ith level

*Supported in part by MADALGO – Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation, and by DFG grant ME 3250/1-2.

[1]The parameter k in "k-d tree" denotes the dimensionality of the input and is not to be confused with the number k of cluster centers in k-means.

Algorithm 1 Filter (=remove) candidate centers not relevant for the points stored in the (sub-)tree rooted at v [7].

```
1:  procedure FILTER(Node v, Points candidateCenters)
2:      if v is a leaf node then
3:          z* := candidate center closest to v's point.
4:          Update (weighted) center z* by v's point.
5:          Increment weight of z* by one.
6:      else
7:          z* := candidate center closest to center of v's cell.
8:          for each z ∈ candidateCenters do
9:              if z* strictly closer to v's cell than z then
10:                 Remove z from candidateCenters.
11:         if |candidateCenters| = 1 then
12:             Update z* by weighted center stored in v.
13:             Increment weight of z* by size of v's subtree.
14:         else
15:             FILTER(v.leftChild, candidateCenters);
16:             FILTER(v.rightChild, candidateCenters);
```

from the top ($i = 0, \ldots$) we use the median of the points' coordinates in the $(i \mod d) + 1$-th dimension to partition the point set into two sets assigned to the children of v. Thus a k-d tree is a complete binary tree except for possibly the last level. A k-d tree uses linear space and can be built top-down using median-finding in $O(n \log n)$ time.

Filtering Approach: The algorithm by Kanungo *et al.* [7] first builds a k-d tree on top of the points to be clustered and then assigns the points to the corresponding cluster using Algorithm 1 (FILTER), where the set of candidate centers is filtered starting at the root of the k-d tree.

Instead of using two nested loops to determine closest candidate centers, each iteration of the k-d k-means algorithm invokes FILTER to reduce the set of candidate centers that might be of relevance for subsets of the input. The algorithm exploits the k-d tree hierachy and that candidate centers "far away" from the box of an internal node v are guaranteed to be "far away" from any input point stored in the subtree rooted at v. The algorithm recursively traverses the tree while maintaining a set of candidate centers. At each node v, all candidates strictly farther away from v's box than the candidate closest to the *center* of v's box are pruned since no point stored in the subtree rooted at v can possibly be assigned to them (l. 7–10). If only one candidate is left, all points stored in the subtree are assigned to it (l. 12–13), else the algorithm recurses (l. 15–16). When reaching a leaf, its point is checked against all remaining candidates (l. 3–5).

To facilitate the analysis, k-d k-means uses a *longest-side k-d tree* [9] where the splitting hyperplane is always orthogonal to the longest side of a node's box. Clearly, this does not affect the tree's depth or the space requirement.

B. Faulty-Memory RAM Model

The *faulty-memory RAM model* [5] in an extension of the RAM model where memory corruptions may happen during the execution of an algorithm with no restrictions when and where these corruptions occur. A (small) constant number of "safe" memory cells (CPU registers) is available. We have efficient resilient solutions for sorting, searching, and priority queues [5], [10]–[12].

Algorithm 2 Retrieve a value reliably stored in an array A.

```
1:  function GETVALUE(Array A[0 ... 2δ])
2:      confidence := 0;              ▷ Start with zero confidence.
3:      for i := 0 to 2δ do           ▷ Check all 2δ + 1 values.
4:          if confidence = 0 then
5:              candidate := A[i];    ▷ Declare a new
6:              confidence := 1;      ▷ candidate.
7:          else if A[i] = candidate then
8:              confidence++;         ▷ Increase confidence.
9:          else
10:             confidence−−;         ▷ Decrease confidence.
11:     return candidate;
```

To ensure a worst-case setting while at the same time allowing for an analysis at all, the corruptions are assumed to be performed by an adversary with full knowledge of the algorithm and the goal to inflict maximum damage but algorithms are provided with an upper bound δ on the amount of memory corruptions that may occur. In this model, an algorithm is said to be *resilient* if it produces a correct result for the uncorrupted data. For instance, a resilient sorting algorithm ensures that all uncorrupted values appear in sorted order in the output, whereas corrupted values may appear anywhere and in any order in the output.

In the following we denote by *reliable value* a value stored safely in unsafe memory. This is achieved at the cost of $\Theta(\delta)$ overhead in space and time for both reading and writing the value. The value is written by replicating it $2\delta + 1$ times. Since there can be at most δ corruptions, a majority vote algorithm [13] (Algorithm 2) that maintains a candidate and a confidence can be used to retrieve a reliable value.[2]

Since we have only $O(1)$ safe memory cells, some techniques may not be used as in the RAM model. For instance, recursion is largely prohibited, since it requires a recursion stack, usually of non-constant size. For data structures, (non-reliable) pointers may not be used, since a pointer corruption leads to the loss of data stored in the given data structure.

III. K-D K-MEANS MADE RESILIENT

In this section, we develop a resilient k-d tree and use it to resiliently filter candidate centers without assuming critical data to be protected from corruptions. We use only a constant number of safe cells. Since all comparisons between data points are done on a coordinate-by-coordinate basis, we do not need to assume the dimensionality to be bounded by δ.

A. Resilient K-d Tree

Structure: A resilient k-d tree consists of a *top tree* and a number of *leaf structures*—see Figure 1. The top tree is

[2]To apply this replication to all data items would increase both space and time by a factor of $\Theta(\delta)$ and thus is infeasible.

a complete binary tree and corresponds to the top part of a classical k-d tree where each piece of data stored at a node is a reliable value (see Section II-B), and each leaf structure stores a number of input points. Each internal node v reliably stores the corresponding d-dimensional box, the weighted centroid, and the number of points stored in the subtree rooted at v. Since the top tree is a complete binary tree, we can avoid the use of child pointers by storing the nodes in breadth-first-search order in an array. More precisely, the children of a node stored at index i are stored at indices $2i+1$ and $2i+2$. Each node in the last level of the top tree has a corresponding leaf structure. The leaf structures are stored in an array as well, and for each leaf space is allocated to accommodate at most $b\delta$ input points, for some predefined constant b, along with a (cluster assignment) label for each point. Since leaf structures and leaves of the top tree correspond, we also have a bounding box (reliably stored at the leaf of the top tree) for the points in each leaf structure.

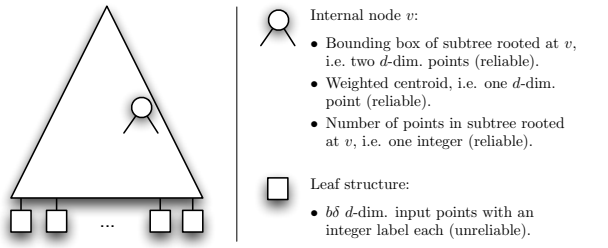


Figure 1. Resilient k-d tree. The top tree is a complete k-d tree with $n/(b\delta)$ leaves corresponding to a leaf structure with $b\delta$ points each.

We do not need pointers for linking the top tree to the leaf structures: Since the top tree is complete, the last level $l_{\max}$ contains $2^{l_{\max}}$ nodes, which—due to the breadth-first-search layout—are stored at the indices $[2^{l_{\max}}-1,\dots,2^{l_{\max}+1}-2]$ in the top tree. Thus, the leaf structure corresponding to one of these nodes stored at index $i \in [2^{l_{\max}}-1,\dots,2^{l_{\max}+1}-2]$ is stored at index $i-(2^{l_{\max}}-1)$ in the array of leaf structures.

Lemma III.1. *Given $b \geq 1$, a resilient k-d tree for n d-dimensional points stores at most $2n + 12n(2\delta+1)/(b\delta)$ d-dimensional points and $2n + 4n(2\delta+1)/(b\delta)$ integers in the faulty-memory RAM model with parameter δ.*

The parameter b induces a time/space tradeoff and interpolates between the k-d k-means and the k-means algorithm. When b increases, the size of a leaf structure increases and the size of the top tree and the total space requirement decreases. When the leaf structures have many points, however, the top k-d tree has few nodes and thus the chances of filtering candidates before reaching the leaf structure are diminished. In the extreme case ($b = n/\delta$), we have exactly one leaf structure, and the resulting algorithm is k-means. Therefore, the constant b induces a tradeoff between space usage and the likelihood of efficiently filtering candidates.

Lemma III.2. *Assuming that $\delta \geq 3$, the space requirement for the resilient k-d tree is at most four times the space requirement for the classic k-d tree as long as $b \geq 14$.*

Construction: The construction of the resilient k-d tree is an adaptation from the classical k-d tree construction algorithm. We start with a d-dimensional box containing all the input points and store it at the root of the tree, together with the weighted centroid. For each node, the box is split in two orthogonal to its longest side (since we reliably store the box for each node, we can easily retrieve its longest side), and the resulting sub-boxes are associated with the children. For each node, we reliably store the corresponding box, the number of input points in this box, and their weighted centroid. We stop splitting when the number of points in the box drops below $b\delta$, and store the points in a leaf structure. Since no efficient resilient median selection algorithm has been proposed so far, we have to rely on resilient sorting for splitting the boxes. We obtain the following lemma:

Lemma III.3. *The resilient k-d tree can be constructed in $O(n\log^2 n+\delta^2)$ time. It supports resilient orthogonal range queries in $O(\sqrt{n\delta}+t)$ time for reporting t points.*

B. Resilient K-d K-Means

In this section, we discuss three major adaptations needed to integrate the resilient k-d tree into k-d k-means.

Protecting the flow control: A defining feature of the faulty-memory model is that, with the exception of $O(1)$ "safe" memory cells, any memory location can be corrupted at any time. This implies that the recursion stack of any recursive algorithm is prone to corruptions which result in faulty computations or segmentation faults. Since the recursion stack used in FILTER has a depth of $O(\log n/(b\delta))$, it cannot be stored in the safe memory. The breadth-first-search layout of the tree allows for a pointerless navigation (the children of node i are stored at index $2i+1$ and $2i+2$). Since each left child of a node has an odd-numbered index and that each right child has an even-numbered index, returning from a "recursive call" is straightforward, and we can deduct whether the child returned from is a left or a right child.

Protecting the candidate set: Since corrupting a single candidate center may severely affect the final output, we store the set of candidate centers reliably. While filtering the set of candidate centers we need to traverse the tree up and down and thus we need to be able to (reliably) reconstruct the candidate centers possibly filtered out at the parent of the current node. For each node v on the current recursion path, we reliably store a bit vector I_v of size $\lceil \log_2 k \rceil$ words (k bits) indicating whether i-th candidate center is relevant for v. We then explicitly maintain a (recursion) stack of reliably stored bit vectors of size $\lceil \log_2 k \rceil$ each. The space requirement per level is $\lceil \log_2 k \rceil (2\delta+1)$ integers. We note that we could further reduce the space requirement to one reliably stored integer per level, i.e., to $(2\delta+1)$ integers per

level, using techniques from in-place algorithms [14], but since in the settings we consider the value of k is small, the asymptotic gain in performance is most likely to be offset by the slightly more complicated in-place algorithm.

Checking consistency during the filtering phase: In an internal node, every piece of information required is read and written reliably. To avoid corruptions in the leaf structures to severely affect the candidate centers, we check whether p lies inside the (reliably stored) box associated to the leaf before assigning it to a center: for a point $p = (p_1, \ldots, p_d)$ and a box B defined by its "lower left" corner $b_1 = (b_{11}, \ldots, b_{1d})$ and "upper right" corner $b_2 = (b_{21}, \ldots, b_{2d})$, we check whether $p_i \in [b_{1i}, b_{2i}]$, for each $i \in \{1, \ldots, d\}$. If p_i is out of range it must have been corrupted and we reset it to $(b_{1i} + b_{2i})/2$. This way the intrinsic structuring of the point set in a k-d tree allows for error detection and helps further reducing the effect of memory corruptions on the output.

IV. Experiments

For the experimental evaluation, we implemented the standard k-means algorithm and its variant based on k-d trees along with our resilient k-d tree-based approach. The purpose of the remainder of this section is to analyze the influence of memory corruptions on these implementations. Following Wagstaff and Bornstein [2], we use two real-world data sets to investigate the behavior of all approaches given certain setups for the involved parameters.

A. Experimental Setup

The experimental results were obtained on a standard Desktop PC having an Intel Dual Core CPU (only one core was used by our algorithms) at 2.66 GHz and 4 GB RAM running Debian Linux, kernel version 2.6.26. All algorithms as well as the testbed allowing for injecting memory corruptions were implemented in `C` and were compiled using the `gcc` compiler version 4.3.2 with optimization level `-O3`.

Data Sets: We test all approaches on multi-spectral satellite data obtained from the Hyperion instrument onboard the EO-1 Earth orbiter. The instrument collects data at 242 wavelengths but onboard computations can only be done on a selectable subset of up to 12 bands [1], [2]. For our experiments, we consider the same 11 bands used by the onboard pixel SWIL classifier described in [1]. We consider two particular data sets which are based on observations of the Quinghai Province in China made on October 3, 2002.[3] The first one (`Quinghai Large`) contains $n = 179.200$ patterns (i.e., pixels) each having $d = 11$ features (bands). The second one (`Quinghai Small`) is a subset of the first one and contains $n = 1.600$ patterns with $d = 11$ features.

[3]Both data sets were kindly provided by the authors of [2], [3].

Clustering Evaluation: To evaluate the clustering performance, we use "grounded truth" labels. For both data sets these labels were manually obtained for all pixels. The quality of a computed partition with respect to the true labels is then quantified with the *Adjusted Rand Index* (ARI) [15], which is a measure for the agreement of two given partitions of a set. A value of 1.0 indicates a perfect agreement, a value of 0.0 is obtained in expectation by random partitions, and negative values stem from anti-correlated partitions.

Parameters: We set the number of designated clusters k to the true number of classes for each data set (i.e., $k = 3$ for `Quinghai Large` and $k = 2$ for `Quinghai Small`). Since all convergence guarantees of the k-means clustering approach are invalid due to (possible) data corruption, we fix the number of iterations to 30. The k-means approach in general is susceptible to the problem of local optima. In addition, the memory corruptions can lead to unstable results. We thus average the clustering results over 30 runs. For our resilient k-d k-means variant, unless noted otherwise, we fix $\delta = 10$ and $b = 5$. The memory corruptions can lead to unpredictable behaviors and hence to unlimited practical runtimes. Thus, we restrict all algorithms to finish within a user-defined time interval which we set to 500 seconds.

Memory Manager: The testbed for performing memory corruptions is based on a *memory manager*. The memory manager is used to allocate corruptible cells, i.e., bytes of memory which can be affected by bit flips. The $\mathcal{O}(1)$ corruption-free cells are obtained using the standard `C`-routines for memory administration. In addition to the allocation of sufficient bytes of memory, the memory manager also maintains an auxiliary data structure which can be used to perform memory corruptions at arbitrary positions of the space allocated through it.[4] A memory corruption consists of a single bit flip. Since all algorithms make use of corruptible indices, access to arbitrary memory positions might occur. In line with [2], we do not corrupt the program code.

Implementation Issues: To ensure a corruption-free execution, we endow non-resilient algorithms with a `getIndex`-routine which prevents out-of-bounds accesses to memory locations. The access of a memory cell via a corrupted index is safeguarded by this routine as follows: if a non-valid (corrupted) index is used to access a memory location, the routine redirects the access by setting the index to a predefined valid value). We note that this routine is an added benefit for the k-means and non-resilient k-d k-means algorithms and thus makes them stronger competitors. Since Wagstaff and Bornstein [2] observed that the k-d k-means algorithm is most vulnerable if the k-d tree is affected by bit flips, we further improve the non-resilient k-d k-means using a pointerless k-d tree, i.e., we embed it in breadth-first-search order in a node array.

[4]Due to intellectual property issues, the authors of [2] were unable to provide us with the BITFLIPS radiation simulator used in their experiments.

Memory Corruptions: Following Wagstaff and Bornstein [2], we consider a *radiation rate* which determines the amount of *single-event upsets* (SEUs) per byte and per second. Hence, the amount of memory corruptions is proportional to the radiation rate, the execution time, and the allocated space. This means that one might estimate wrongly the δ parameter in a practical scenario. More precisely, even if our algorithm is guaranteed to be resilient against a certain number of memory corruptions, it might face a much larger number of corruptions. In a real-world scenario, memory corruptions can happen at any time. To simplify the implementation, we inject memory corruptions after each iteration of all k-means approaches (based on the iteration's runtime and global space usage). Finally, we do not perform memory corruptions during the building phases of the k-d tree-based approaches. This decision is justified foremost by the insignificant relative running time of the building phase: profiling our code showed that we spend consistently (and almost always significantly) less than 6% of the running time on building the (resilient) k-d trees. This implies that an insignificant number of corruptions is induced in the building. Furthermore, most corruptions in the resilient k-d tree can be recovered from by exploiting the hierarchical structure and the reliably stored boxes during a linear-time traversal of the tree. Since such a recovery is not possible in the classic k-d tree, not injecting errors in the building phase actually helps our algorithm's competitor.

B. Results

Clustering Accuracy: Figure 2 shows the clustering performance of all implementations on the two data sets given varying rates of radiation (we report averaged results with standard deviation; note that the x-axis are different). On both data sets, the resilient version shows a superior clustering performance. This might be due to the fact that both k-d tree variants are well-suited for data sets with a large amount of patterns in a low-dimensional feature space Here, the runtime depends on the efficiency of the pruning approach which takes place in the "upper region" of the associated k-d tree. Since this information is stored resiliently for the `resilient k-d k-means` approach, corruptions should have less influence on the final clustering.

A comparison of the clustering results for the `Quinghai` data sets indicates that the influence of the radiation rate increases with increasing size of the data set. An explanation might be that even a single bit flip can have a significant influence on the clustering result, e.g., if the exponent of a floating point representation is corrupted—see Figure 3. Since the probability of corruptions increases with the size and dimensionality of the data set, we expect worse results for clustering larger data sets with non-resilient algorithms.

In the remaining experiments, we attempt to give more insights on the resilient k-d k-means implementation. To ensure that the experimental results are as insightful as

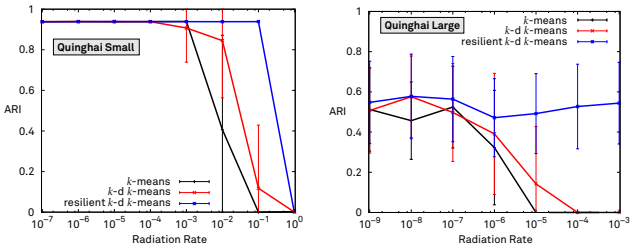
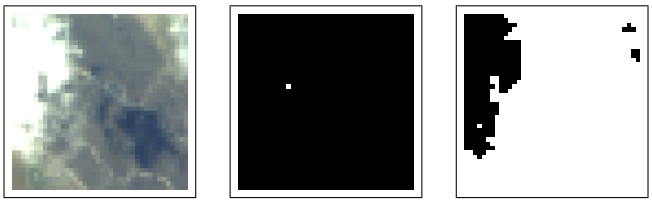


Figure 2. Clustering performance (adjusted rand index) of all approaches with respect to memory corruptions (determined by the radiation rate).



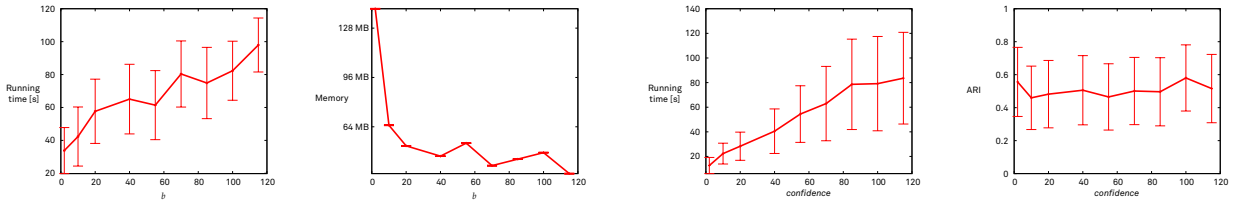
(a) RGB (b) `(k-d) k-means` (c) `res. k-d k-means`

Figure 3. Clustering on `Quinghai Small` for a radiation rate of 10^{-2}. For `k-means` and `k-d k-means`, one of the two final centroids has only a single training pattern assigned to it. This might be due to the fact that a single corrupted training pattern (where the exponent of its floating point representation is hit) can lead to a corrupted centroid in the next iteration. Afterwards, this centroid always will have this single point assigned to it.

possible, the experiments are conducted only on the largest dataset available, namely `Quinghai Large`. Also, we fix the parameters as follows. We set $\delta = 50$ and, to ensure that the number of corruptions is smaller than δ, the rate at which memory corruptions occur is set to 10^{-8}.

Tradeoff parameter b: As previously discussed, the parameter b imposes a tradeoff. The greater b the more points a leaf structure contains leading to fewer nodes in the k-d tree. This means smaller space consumed but higher running times if filtering reaches the leaves, as in this situation we scan all the points in a leaf. We conduct experiments to demonstrate the influence of b on the running time and memory usage. Results in Figure 4(a) confirm our expectations and clearly state that as b increases, the running time increases and the memory usage decreases. For the remaining experiments we use a (balanced) value $b = 10$.

Faster running times: By profiling our code we observed that a significant amount of time was spent retrieving reliable values. Under the practical assumption that corruptions occur uniformly at random, it becomes highly unlikely that two data items become corrupted to the same value. Since retrieving a reliable value is done based on seeing at least $\delta + 1$ identical values, i.e. when $\mathit{confidence} = \delta + 1$ in Algorithm 2, this means that halting its execution at a value $2 \leq \mathit{confidence} < \delta + 1$ is extremely likely to provide the actual value stored. This way, even if writing reliable values still requires writing $2\delta + 1$ memory cells, reading them can be significantly sped up. Since we are not willing



(a) Running time (left) and memory usage (right) for varying b values. (b) Running time (left) and ARI (right) for varying *confidence* values.

Figure 4. Dependencies of running time, memory usage, and clustering performance on (a) the parameter b and (b) the *confidence* value.

to trade accuracy for running time, we study experimentally how this parameter affects both the running time and the ARI. The results in Figure 4(b) show that retrieving reliable values at low values for *confidence* does not seem to affect the accuracy measured by ARI, but they significantly speed up the execution of k-d k-means.

Significantly more corruptions than δ: We now investigate a scenario where the number of corruptions α that occur is significantly larger than the δ value provided to the algorithm. The results in Figure 5 show that the ARI is not affected when doing more corruptions than δ, even when α exceeds δ by very large margins. First, the likelihood of corrupting a resilient value is extremely small as discussed above, and thus sensitive data such as the centers and the recursion stack are still reliable. Secondly, we make sure that corruptions in the input patterns do not disturb the output too much by moving any pattern outside the bounding box of its leaf structure into the middle of this box.

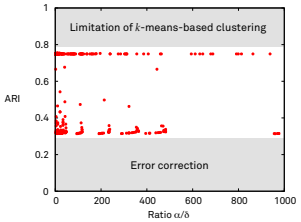


Figure 5. Performing more than δ corruptions. The x-axis shows the corruption ratio α/δ. The best k-means-based clustering gives an ARI of ≈ 0.78 (see Figure 2), the lower bound is due to our error correction.

References

[1] R. Castano, D. Mazzoni, N. Tang, T. Doggett, S. Chien, R. Greeley, B. Cichy, and A. Davies, "Learning classifiers for science event detection in remote sensing imagery," in *Symp. on Artificial Intelligence, Robotics and Automation in Space*, 2005.

[2] K. L. Wagstaff and B. Bornstein, "K-means in space: A radiation sensitivity evaluation," in *Intl. Conf. on Machine Learning*, 2009, pp. 1097–1104.

[3] ——, "How much memory radiation protection do onboard machine learning algorithms require?" in *Workshop on Artificial Intelligence in Space*, 2009.

[4] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Trans. Electron Devices*, vol. ED-26, no. 1, pp. 2–9, Jan. 1979.

[5] I. Finocchi, F. Grandoni, and G. F. Italiano, "Designing reliable algorithms in unreliable memories," *Computer Science Review*, vol. 1, no. 2, pp. 77–87, 2007.

[6] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Berkeley Symp. on Mathematical Statistics and Probability*, 1967, pp. 281–297.

[7] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 24, no. 7, pp. 881–892, Jul. 2002.

[8] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Comm. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[9] M. Dickerson, C. A. Duncan, and M. T. Goodrich, "k-d trees are better when cut on the longest side," in *European Symp. on Algorithms*, ser. LNCS, vol. 1879, 2000, pp. 179–190.

[10] G. S. Brodal, R. Fagerberg, I. Finocchi, F. Grandoni, G. Italiano, A. G. Jørgensen, G. Moruz, and T. Mølhave, "Optimal resilient dynamic dictionaries," in *European Symp. on Algorithms*, ser. LNCS, vol. 4698, 2007, pp. 347–358.

[11] I. Finocchi, F. Grandoni, and G. F. Italiano, "Optimal resilient sorting and searching in the presence of dynamic memory faults," *Theoretical Computer Science*, vol. 410, no. 44, pp. 4457–4470, 2009.

[12] A. G. Jørgensen, G. Moruz, and T. Mølhave, "Priority queues resilient to memory faults," in *Workshop on Algorithms and Data Structures*, ser. LNCS, vol. 4619, 2007, pp. 127–138.

[13] R. S. Boyer and J. S. Moore, "MJRTY: A fast majority vote algorithm," in *Automated Reasoning: Essays in Honor of Woody Bledsoe*, 1991, pp. 105–118.

[14] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold, "Space-efficient geometric divide-and-conquer algorithms," *Computational Geometry: Theory & Applications.*, vol. 37, no. 3, pp. 209–227, Aug. 2007.

[15] L. Hubert and P. Arabie, "Comparing partitions," *Journal of Classification*, vol. 2, no. 1, pp. 193–218, 1985.