Front. Comput. Sci., 2012, 6(2): 166–178
DOI 10.1007/s11704-012-2870-8

RESEARCH ARTICLE

Resilient *k*-d trees: *k*-means in space revisited

Fabian GIESEKE (✉)[1], Gabriel MORUZ[2], Jan VAHRENHOLD[3]

1 Department für Informatik, Carl von Ossietzky Universität Oldenburg, Oldenburg 26111, Germany
2 Department of Computer Science, Goethe Universität Frankfurt am Main, Frankfurt am Main 60325, Germany
3 Fakultät für Informatik, Technische Universität Dortmund, Dortmund 44227, Germany

Abstract We propose a *k*-d tree variant that is resilient to a pre-described number of memory corruptions while still using only linear space. While the data structure is of independent interest, we demonstrate its use in the context of high-radiation environments. Our experimental evaluation demonstrates that the resulting approach leads to a significantly higher resiliency rate compared to previous results. This is especially the case for large-scale multi-spectral satellite data, which renders the proposed approach well-suited to operate aboard today's satellites.

Keywords data mining, clustering, resilient algorithms and data structures

1 Introduction

Data analysis aboard spacecraft 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 delays like Jupiter and Saturn missions [2,3]. Hence, machine learning approaches like clustering or classification seem to have the potential to dramatically reduce unnecessary idle periods by prioritizing data and/or by creating bandwidth-saving compressed "previews" for transmission to earth.

Received June 14, 2011; accepted November 6, 2011

E-mail: f.gieseke@uni-oldenburg.de

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][1)]. Both conditions can severely affect the computations aboard these spacecraft by causing memory corruptions (*bit flips*) in the various layers of the system memory hierarchy. For modern (spacecraft) systems, the influence of memory corruptions in the CPU (i.e., registers and caches) is addressed by using fully radiation-hardened hardware components [2]. For the remaining parts of the memory hierarchy, software (e.g., bit-encoding schemes) and/or hardware-based concepts are used to reduce the effect of corruption. However, both approaches lead to higher costs, an increase in mass, and/or a reduction in capability and speed in most cases [2].

As an alternative approach, we propose to resort to *resilient* algorithms which can guarantee reasonable outputs in the presence of memory corruption and exhibit only a small overhead in space and runtime. These algorithms are usually analyzed in the *faulty-memory RAM model* [6] which extends the classical RAM model in the sense that memory cells may get corrupted at any place and at any time during the execution of an algorithm, and that corrupted and uncorrupted cells cannot be distinguished.

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

[1)] The influence can even be observed on earth (sea-level) but is already two orders of magnitude larger at 10 000 meters [5].

in a radiation-free environment, exhibits inferior clustering performance compared to the standard approach when the data cells are exposed to memory corruptions. In this work, we propose a resilient version of the 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 on real-world data compared to the standard k-means approaches even in the presence of massive memory corruption.

In Section 2, we provide the algorithmic background for our approach including descriptions for the standard k-means approach and its k-d tree-based acceleration as well as a description of the faulty-memory RAM model [6]. Our resilient k-means variant is based on a resilient version of the k-d tree data structure, which is of independent interest. Both the resilient k-d tree data structure as well as the induced resilient k-means approach are detailed in Section 3. Our experimental analysis on real-world data is provided in Section 4 followed by the conclusions given in Section 5.

2 Background

2.1 k-means clustering

The standard k-means clustering approach is a simple local search strategy which aims to find an assignment of all input points to a predefined number of clusters k [7]. For completeness, we provide a short description of it, see Algorithm 1. The main idea of the algorithm is to maintain a set of k candidate centers as the mean of the input points closest to them and to iteratively adjust the centers until the algorithm converges. More precisely, after initializing the set of candidate centers by k points, sampled uniformly at random from the set of input points, the algorithm proceeds in multiple iterations. In each iteration, every point is assigned to its nearest candidate center (lines 6–9), and at the end of the iteration, each candidate center is updated to the mean of all points assigned to it (lines 10–11). The algorithm terminates as soon as some stopping criterion is met (e.g., an iteration did not result in any change of assignments or that a user-defined number of iterations is reached). Even though an exponential worst-case lower bound on the number of iterations needed to converge is known [8], k-means and its variants are known to be efficient in practice, and it has been shown, using the concept of *smoothed analysis*, that k-means exhibits polynomial run time in practice [9]. In the past decade, several variants of the original k-means algorithm have been proposed [10–15]. The one we work with in the following is the k-d k-means algorithm proposed by Kanungo et al. [14].

2.2 k-d-based filtering

One of the bottlenecks of a straightforward k-means implementation is the recurrent computation of nearest neighbors. We build upon Kanungo et al.'s [14] k-means variant that is based on k-d trees [16]. Their algorithm first builds a k-d tree for the input points and uses this data structure to speed up the nearest neighbors computations. We briefly review the concept of k-d trees and subsequently sketch Kanungo et al.'s heuristic.

2.2.1 k-d trees

A k-d tree [16] is a binary search tree for a set of multidimensional points[2]. Each internal node v in a k-d tree for a set of

Algorithm 1 K-means algorithm [7]

```
1:  function KMEANS(Points $P$, int $k$)
2:      Randomly choose clusters $\{c_1, \ldots, c_k\}$ from $P$.
3:      for $i := 1$ to $|P|$ do
4:          Assign point $p_i$ to cluster 1, i.e., set $a_i := 1$.        ▹ Initialize set of point/cluster assignments
5:      repeat
6:          for $i := 1$ to $|P|$ do
7:              for $j := 1$ to $k$ do
8:                  if $p_i$ is closer to $c_j$ than to $c_{a_i}$ then
9:                      $a_i := j$;                                  ▹ Update point/cluster assignment
10:         for $i := 1$ to $k$ do
11:             Compute $c_j$ as the mean of $\{p_i \mid a_i = j\}$.     ▹ Update cluster centers
12:     until no assignment was modified in this iteration
13:     return $\{c_1, \ldots, c_k\}$
```

[2] For historical reasons, the parameter k in "k-d tree" denotes the dimensionality of the input and is not to be confused with the number of cluster centers k in k-means.

d-dimensional points corresponds to a d-dimensional box that contains all the 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 from the top ($i = 0, 1, \ldots$) we use the median of the points' coordinates in the $(i \bmod d) + 1$-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 linear-time median-finding in $O(n \log n)$ time. Classically, a k-d tree is used for answering orthogonal range queries; all t points in a d-dimensional query range can be reported in $O(n^{1-1/d} + t)$ time [17].

2.2.2 Filtering approach

Kanungo et al. [14] presented a k-d tree-based variant of the k-means algorithm. Their algorithm first builds a k-d tree on top of the points to be clustered and then replaces lines 6–11 of Algorithm 1 by a call to Filter (see Algorithm 2) 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 for each point its closest candidate center, each iteration of the k-d k-means algorithm invokes Filter to try to first reduce the set of candidate centers that might be of relevance for certain subsets of the input points. More precisely, the algorithm takes advantage of the hierarchical subdivision induced by the k-d tree and the implication that candidate centers "far away" from the box of an internal node v thus are guaranteed to be "far away" from any input point stored in the subtree rooted at v. The algorithm recursively traverses the tree and maintains a set of candidate centers for each visited node. At each node v, all candidate centers that are strictly farther away from v's box than the candidate center closest to the *center* of v's box are excluded from further consideration since no point stored in the subtree rooted at v can possibly be assigned to them (lines 7–10). If only one candidate center is left at this point, all points stored in the subtree rooted at v are assigned to this center and the center is updated (lines 12–13), otherwise the algorithm recurses (lines 15–16). If the recursion terminates at a leaf w, the point stored at w is checked against all remaining candidate centers, and the center closest to w's point is updated accordingly (lines 3–5). We note for the sake of completeness that, to facilitate the analysis, the k-d k-means algorithm is actually based upon a variant of the k-d tree, the so-called *longest-side k-d tree* [18]. In this variant, the splits are not performed in the dimension-by-dimension round-robin manner described above but the dimension inducing the splitting hyperplane for each node is chosen to be orthogonal to the longest side of the node's box. Clearly, neither the space consumption nor the depth of the tree are affected by this.

2.3 Model of computation

As stated in the introduction, we resort to a variant of the classical RAM model of computation for the theoretical analysis of our approach. We now provide some details related to this model.

2.3.1 Faulty-memory RAM model

The faulty-memory RAM model [6] is an extension of the classical RAM model where memory-cell corruptions may occur at any time during the execution of an algorithm with

Algorithm 2 Filter (remove) candidate centers not relevant for the points stored in the (sub-)tree rooted at v [14]

```
1:  procedure FILTER(Node $v$, Points $candidateCenters$)
2:      if $v$ is a leaf node then                 ▷ Base of the recursion: check all remaining candidate centers
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 the centerpoint of $v$'s cell.
8:          for each $z \in candidateCenters$ do
9:              if $z^*$ strictly closer to the boundary of $v$'s cell than $z$ then
10:                 Remove $z$ from $candidateCenters$.     ▷ Center $z$ cannot be relevant for subtree rooted at $v$
11:         if $|candidateCenters| = 1$ then
12:             Update the (weighted) center $z^*$ by the weighted center stored in $v$.
13:             Increment weight of $z^*$ by the size of the subtree rooted at $v$.
14:         else
15:             FILTER($v.leftChild$, $candidateCenters$);      ▷ Recurse on the left subtree
16:             FILTER($v.rightChild$, $candidateCenters$);     ▷ Recurse on the right subtree
```

no restrictions on when and where these corruptions may take place. The only exception is a (small) constant number of "safe" memory cells (CPU registers) that are available to store algorithmic building blocks, like loop indices, program counters, or a constant number of data items. Furthermore, corrupted cells cannot be distinguished from uncorrupted ones. To ensure a worst-case scenario while at the same time allowing for a meaningful analysis, the corruptions are assumed to be performed by an adversary with full knowledge of the algorithm and with the goal to inflict maximum damage. However, the algorithms are provided with an upper bound δ on the number 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. Finally, the space complexity should not increase asymptotically.

Several problems have been addressed in this model, mostly fundamental algorithms and data structures such as sorting, searching, and priority queues [6,19–22]. Recently, local dependency dynamic programming [23], resilient counters [24], and cache-efficient resilient algorithms [25] have been proposed. We note that, since the model allows only $O(1)$ safe memory cells, certain techniques may not be used as in the classical RAM model. For instance, recursion is largely prohibited, since it requires maintaining a recursion stack, usually of non-constant size. Also, pointers may not be used in data structures unless they are reliably stored, since pointer corruption leads to the loss of data stored in the referenced part of the data structure.

2.3.2 Reliable values

In the remainder of the paper we denote a value stored safely in unsafe memory as a *reliable value*. 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 in consecutive memory cells. Since there can be at most δ corruptions, the majority of the copies are uncorrupted. A reliable value is retrieved using the majority vote algorithm [26] (Algorithm 3) that scans all the copies while maintaining a candidate and a confidence value. A trivial way of ensuring resilience is to apply this replication to *all* data items. This, however, would increase both space and time by a factor of $\Theta(\delta)$ and thus is infeasible.

3 *k*-d *k*-means made resilient

In this section, we introduce a version of the *k*-d *k*-means algorithm which is guaranteed to be resilient to a predefined number of memory corruptions. The key difference to the classical *k*-d *k*-means is that we develop a resilient *k*-d tree, and for this data structure we show how to resiliently filter candidate centers without assuming critical data to be protected from corruptions. In accordance with the model, we use only a constant number of corruption-free cells throughout the algorithm. We note in passing that, since all comparisons between data points are done on a coordinate-by-coordinate basis, we do not need to assume the dimensionality to be constant or bounded by δ.

3.1 Resilient *k*-d tree

We first describe the structure of the resilient *k*-d tree and then discuss how to construct and query the tree.

3.1.1 Structure

The resilient *k*-d tree consists of a *top tree* and a number of *leaf structures*, see Fig. 1. The top tree is 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 3), and each leaf structure stores a number of

Algorithm 3 Retrieving a value reliably stored in an array *A* (based upon [26])

```
1:  function GetValue(Array $A[0\dots 2\delta]$)
2:      $confidence := 0$;                              ▹ Start with zero confidence
3:      for $i := 0$ to $2\delta$ do                    ▹ Check all $2\delta+1$ 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:              $confidence := confidence + 1$;         ▹ Increase confidence
9:          else
10:             $confidence := confidence - 1$;         ▹ Decrease confidence
11:     return $candidate$;
```

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 also stored in an array, 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 the 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.

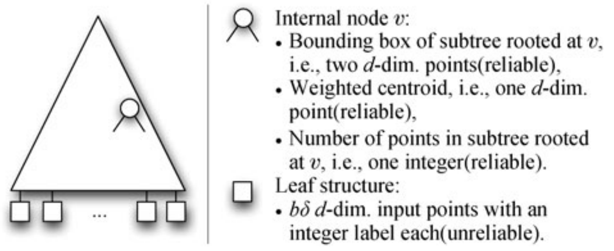
Internal node v:
• Bounding box of subtree rooted at v, i.e., two d-dim. points(reliable),
• Weighted centroid, i.e., one d-dim. point(reliable),
• Number of points in subtree rooted at v, i.e., one integer(reliable).
Leaf structure:
• $b\delta$ d-dim. input points with an integer label each(unreliable).
...

Fig. 1 Resilient k-d tree

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 $[2^{l_{\max}} - 1, \ldots, 2^{l_{\max}+1} - 2]$ in the top tree. Thus, the leaf structure corresponding to one of these nodes stored at index $i \in [2^{l_{\max}} - 1, \ldots, 2^{l_{\max}+1} - 2]$ is stored at index $i - (2^{l_{\max}} - 1)$ in the array of leaf structures.

Lemma 1 Given $b \geqslant 1$, 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.

Proof Since in the top-down construction phase, a point set at any node is split into two sets of roughly equal sizes and since recursion stops as soon as the point set is no larger than $b\delta$, each leaf stores between $1 + b\delta/2$ and $b\delta$ points. Assume the best case, i.e., that each leaf stores exactly $b\delta$ points and that n is an exact multiple of $b\delta$. We then have $n/(b\delta)$ leaf structures and, since each leaf structure corresponds to exactly one leaf node of the top tree, in total $n/(b\delta)+(n/(b\delta)-1) < 2n/(b\delta)$ nodes in the top tree. For each leaf structure, we allocate memory for $b\delta$ input points with an integer label each, see Fig. 1. Since we have $n/(b\delta)$ leaf structures in the best base, the leaf structures occupy space for n points and n integers in the best case. For each node in the top-tree, we allocate memory for $2(2\delta + 1)$ points to reliably store the box (more precisely, the "lower left" and the "upper right" corner of the box), $2\delta + 1$ points to reliably store the weighted centroid, and $2\delta + 1$ integers for the number of points in the subtree. Thus, the nodes in the top tree occupy memory for no more than $3(2\delta+1)\cdot 2n/(b\delta) = 6n(2\delta+1)/(b\delta)$ points and $(2\delta + 1) \cdot 2n/(b\delta) = 2n(2\delta + 1)/(b\delta)$ integers.

Summing up, we obtain that our data structure stores no more than $n + 6n(2\delta + 1)/(b\delta)$ points and $n + 2n(2\delta + 1)/(b\delta)$ integers in the best case. In the worst case, the number of leaf structures and thus the number of nodes in the top tree becomes at most twice as much. Since all leaf structures have to be of the same size to allow for the indexing scheme to work, the maximum number of points that are stored in any of the leaf structures determines the size of all structures. Since, in the worst case, there is exactly one structure that is assigned $b\delta$ points, while all other structures are assigned $1 + b\delta/2$ points, all structures have to be of size $b\delta$. Thus, we have twice as many leaf structures as in the best case, but in either case a leaf structure requires space for $b\delta$ points. This implies that the memory requirements for the top tree and leaf structures double in the worst case.

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 2 Assuming that $\delta \geqslant 3$, the space requirement for the resilient k-d tree is at most four times the space requirement for the classical k-d tree as long as $b \geqslant 14$.

Proof To compare the space requirement of a worst-case resilient k-d tree with the space requirement of a best-case k-d tree, we first note that the classical k-d tree is realized as a pointer-based data structure. To avoid pointer corruption, we implement this k-d tree using the breadth-first-search layout discussed above (this actually saves space for $2n$ pointers). In contrast to the resilient k-d tree, the k-d tree does not store bounding boxes, so, in order to be able to infer the dimension relevant for splitting, each node in a k-d tree stores not only one input point with a label but also an integer denoting the dimension. If $\delta \geqslant 3$ and $b \geqslant 14$, we have $4n(2\delta+1)/(b\delta) < 12n(2\delta+1)/(b\delta) \leqslant 2n$ and, thus, the worst-case space requirement for a resilient k-d tree is less than $4n$ input points and $4n$ integers. In the best case, a k-d tree is a complete binary tree, and thus we can embed it into a node array of size n exactly without wasting any space. In the worst case, the k-d tree is a complete binary tree up to the level directly above the leaves, and every node on this level has exactly one child; to be able to embed such a tree into a node array, we have to waste one entry for every entry storing a leaf. Thus, the space requirement for the original k-d k-means algorithm is no more than n d-dimensional points and $2n$ integers (again, assuming the best case). Without any loss of generality, we assume that a point occupies more space than an integer: the proof follows.

Setting $b \geqslant 28$ we can further reduce the space requirement to no more that three times the space requirement for the classical k-d tree. On the other hand, increasing δ to 10 and setting $b = 5$ results in a space requirement of roughly five times the space required for the original k-d tree; this is the setting used as the default for the experimental evaluation reported upon in Section 3.

3.1.2 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 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.

Proof For the construction of the tree, at each recursion level the run time is bounded by the sorting complexity which is $O(n\log n + \delta^2)$ [20]. Since the errors *over all levels of the tree* add up to δ, and since every point participates in exactly one sorting process on each level, the global complexity of constructing the tree is $O(n\log^2 n + \delta^2)$ as claimed.

Even though we do not use the resilient k-d tree for orthogonal range searching, it supports this operation as well. It works just like in a classical k-d tree, except that when we reach the leaf structures we scan all the points and report only those points inside the query range. The top tree is a k-d tree containing $O(n/\delta)$ nodes; the algorithm requires $\Theta(\delta)$ time to retrieve the reliable data for each of the at most $O(\sqrt{n/\delta})$ nodes visited without reporting any element. This amounts to $O(\sqrt{n\delta})$ time plus the time for reporting the output. Since the time spent scanning a leaf structure is paid for by the corresponding internal node, resilient orthogonal range queries take $O(\sqrt{n\delta} + t)$ time for reporting t points.

3.1.3 Remark: Resilient median selection with relaxed accuracy

For improving the run time of the construction algorithm, which is of less importance for the focus of the work presented here than the filtering algorithm, one could also hope to resort to an efficient resilient median selection algorithm instead of invoking the sorting procedure on each level. As mentioned above, no *deterministic* median selection algorithm has been developed so far.

In the remainder of this subsection, we show that it is possible to adapt the classical (randomized) median selection algorithm the resilient setting if one is willing to relax the accuracy of the result. More precisely, the algorithm takes $O(n+\delta)$ expected time and outputs a (possibly corrupted) value which is guaranteed to be at most $O(\delta)$ positions away from the actual median.

In the setting we are considering (construction of an almost completely balanced tree), using such an algorithm would result in an unnecessarily complicated construction since one would have to adjust the size of the leaf structures on the fly so as to guarantee that the last level of the top tree is complete. Also, as mentioned above, obtaining the most efficient construction of the tree is of less importance for the focus of this paper than the filtering algorithm. Nevertheless, we consider the median selection algorithm to be of enough independent interest to be presented here.

We recall that the classical algorithm first selects a random element, denoted pivot, in the sequence and then partitions the input into two subsequences containing all elements smaller and greater than the pivot. It then recurses on one of the two sequences depending on the rank of the pivot with respect to the median. To adapt this algorithm, we first notice that it is easy to make it iterative (and in-place) because it always recurses on one subsequence, and no work is done after the recursive call, and thus no information needs to be remembered on the recursive stack, see [27]. Naturally, the parameters to be used in the recursive call are stored in the safe memory. When the input size drops to between 3δ and 6δ we stop recursing, and instead we switch to a second phase where we pick a random element x and count the number of elements s and l which are smaller and greater than x, respectively. If both s and l are greater than δ, then x is returned; otherwise, we pick another element x and repeat the second phase. Since the returned element x is at the same time greater and smaller than other δ elements of the at most 6δ of the input of the second phase, it follows that the returned value is at most $O(\delta)$ positions away from its correct location. For the run time, similar to the classical median selection, the first phase takes $O(n)$ time. For the second phase, each iteration scans the at least 3δ elements and the expected number of iterations is at most 3 because each element has probability of at least $1/3$ to be returned, i.e., to be picked such there are at least δ elements smaller and δ elements larger than it.

Lemma 4 Resilient median selection takes $O(n + \delta)$ expected time to find, for a set of n elements and a given integer $k \leqslant n$, an element with rank $k \pm O(\delta)$.

3.2 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. The first two adaptations are needed to cope with the fact that the recursion stack of the original algorithm is of non-constant size and thus cannot be stored in "safe" memory cells, and the third adaptation improves the resiliency of the (unprotected) input points by exploiting the hierarchical structure of the resilient k-d tree.

3.2.1 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 (non-constant) depth of $O(\log n/(b\delta))$, it cannot be stored in the safe memory. However, the breadth-first-search layout of the tree allows for pointerless navigation; when going down one level in the tree, the children of node i are stored at index $2i + 1$ and $2i + 2$. In addition to this, we know that 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, similar to the simulation of divide-and-conquer algorithms in the in-place model [27], we can deduce whether the child returned from is a left or a right child.

3.2.2 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 the 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 [27]. 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.

3.2.3 Checking consistency during the filtering phase

To filter candidate centers, we distinguish between dealing with internal nodes and leaves. In an internal node, we simply use Algorithm 2, where every piece of information required is read and written reliably; the recursive calls are handled as discussed above. To avoid corruptions in the leaf structures severely affecting the candidate centers, e.g., if a corruption flips a significant bit in the exponent of some coordinate, we perform a consistency check on each point before assigning it to a center. More precisely, we check whether p lies inside the (reliably stored) box associated with 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 $i \in \{1, 2, \ldots, d\}$. Due to the hierarchical structure of the resilient k-d tree, we know that when 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 reduce the effect of memory corruptions on the output.

4 Experiments

For the experimental evaluation, we implemented the standard k-means algorithm (k`-means`) and its variant based on k-d trees (k`-d` k`-means`) along with our resilient k-d tree-based approach (`resilient` k`-d` k`-means`). 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 several real-world data sets to investigate the behavior of all approaches given certain setups for the involved parameters.

4.1 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`.

4.1.1 Data sets

Following Wagstaff and Bornstein [2], we use the `Iris` data set from the UCI repository [28] which consists of $n = 150$ patterns each having $d = 4$ features. Furthermore, 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 performed 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 by Castano et al. [1]. We consider two particular data sets which are based on observations of Qinghai Province in China made on October 3, 2002[3]. The first data set (`Qinghai Large`) contains $n = 179\,200$ patterns (i.e., pixels) each having $d = 11$ features (bands), see Fig. 2. The second data set (`Qinghai Small`) is a subset of the first data set and contains $n = 1\,600$ patterns with $d = 11$ features.

4.1.2 Clustering evaluation

To evaluate the clustering performance, we resort to "grounded truth" labels. For both the `Qinghai Large` and the `Qinghai Small` data set, these labels were manually obtained for all pixels (clouds, ground, water), see Figs. 2 and 3. The quality of a computed partition with respect to the true labels is then measured with the *Adjusted Rand Index* (ARI) [29]. This index is a measure for the agreement of two given partitions of a set. A value of 1.0 indicates perfect agreement of both partitions and a value of 0.0 is obtained in expectation by random partitions for the set. Negative values stem from negatively correlated partitions.

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

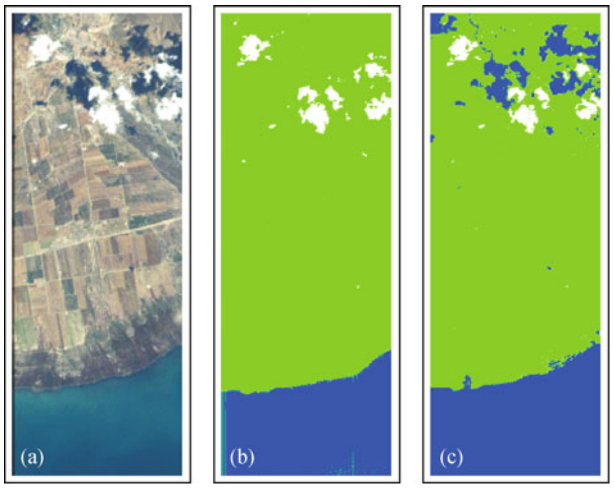

Fig. 2 `Qinghai Large` data set: (a) RGB image based on the 11 bands, (b) manually obtained labels (clouds, ground, water), and (c) standard k-means clustering result (ARI=0.749)

Figures 2 and 3 reveal that the clustering accuracy of the (unmodified) k-means clustering algorithm is much lower for the `Qinghai Large` data set (ARI=0.749) than for the `Qinghai Small` data set (ARI=0.940). This can be explained by observing that the `Qinghai Large` data set contains pixels that correspond to the clouds' shadows (and thus should be classified as "ground") but are misclassified as "water" based on the 11 bands used for the SWIL classifier. These different "upper bounds" for the "grounded truth" classification should be kept in mind when interpreting the classification performance of the other clustering algorithms.

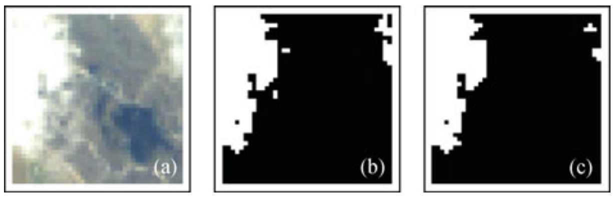

Fig. 3 `Qinghai Small` data set used in [2]: (a) RGB image based on the 11 bands, (b) manually obtained labels (clouds, ground), and (c) standard k-means clustering result (ARI=0.940)

4.1.3 Parameters

Various parameters need to be set for all approaches. We set the number of designated clusters k to the true number of classes for each data set (i.e., $k = 3$ for `Iris` and `Qinghai Large` and $k = 2$ for `Qinghai Small`). Since all convergence guarantees of the k-means clustering approach are invalid due to the (possible) corruption of memory information, we fix the number of iterations to 30 for all experiments. The k-means approach in general is susceptible to the problem of local optima. In addition, memory corruption can lead to unstable results. We therefore average the achieved clustering results over 30 runs. For our resilient k-d k-means variant, two additional parameters δ and b have to be set. If not noted otherwise, we fix $\delta = 10$ and $b = 5$. Memory corruption can lead to unpredictable behavior of all implementations and can, hence, lead to unlimited practical runtimes. We therefore restrict all algorithms to finish within a user-defined time interval which we set to 500 seconds.

4.1.4 Memory manager

The testbed for performing memory corruptions is mainly based on a *memory manager*. The memory manager can be used to allocate corruptible cells, i.e., bytes of memory which can be affected by bit flips. The $O(1)$ corruption-free cells can be obtained using the standard `C`-routines for memory administration. In addition to the allocation of sufficient bytes of memory, the memory manager also takes care of an auxiliary data structure which can be used to perform memory corruption at arbitrary positions of the space allocated[4]. A memory corruption consists of a single bit flip. Since all algorithms make use of corruptible indices (e.g., the current assignments of the patterns), access to arbitrary memory positions might occur. In line with Wagstaff and Bornstein [2], we do not corrupt the program code itself.

4.1.5 Implementation issues

To assure a corruption-free execution of the code, we endow all non-resilient algorithms with a `getIndex`-routine which prevents out-of-bounds access to memory locations. The access of a memory position via a possibly corrupted index is safeguarded by this routine in the following way: 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 point out that this routine is an added benefit for the k-means and non-resilient k-d k-means algorithm 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 resiliency of our competitor by implementing the non-resilient k-d k-means using a pointerless k-d tree, i.e., we also embed this tree in breadth-first-search order in a node array.

4.1.6 Memory corruption

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 depends on both the runtime and the (current) space usage of an algorithm, i.e., corruptions occur proportional to the radiation rate, execution time, and allocated space (as recorded by the memory manager). Most notably, this implies that one might be wrong in estimating the δ parameter in a practical scenario. More precisely, even if our algorithm is guaranteed by design to be resilient against a certain number of memory corruptions, it may face a much larger number of corruptions depending on its runtime and the radiation rate. Experiment 4 is devoted to investigating these issues. In a real-world scenario, memory corruption can happen at any time. To simplify the implementation, we inject a batch of memory corruptions (based on the iteration's runtime and global space usage) after each iteration of each of the k-means approaches such that all of them affect the next iteration. Finally, we do not perform memory corruptions during the building phases of the k-d tree-based approaches. This decision is justified first and foremost by the insignificant relative run time of the building phase: profiling our code shows that we spend consistently and almost always significantly less than 6% of the run time on building the (resilient) k-d trees. This implies that an insignificant number of corruptions is induced in the building time. Furthermore, most if not all 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 classical k-d tree, not injecting errors in the building phase actually helps our algorithm's competitor.

4.2 Results

In the remainder of this section we evaluate the clustering performances of all approaches under the influence of memory

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

corruption given certain setups for their parameters.

4.2.1 Clustering accuracy

A natural question is how different radiation rates affect the clustering performances of the approaches. Figure 4 shows the behavior of all implementations on the three data sets given varying rates of radiation. We report averaged results with standard deviation; note that the *x*-axis is different for the largest data set. On the `Iris` data set, all approaches exhibit a quite similar clustering accuracy. On the other two data sets, the resilient version shows a superior clustering performance. This may be due to the fact that both *k*-d tree variants are well-suited for data sets with a large number of patterns (i.e., large *n*) in a low-dimensional feature space (i.e., small *d*). Here, the practical 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, memory corruptions should have less influence on the final clustering performance.

A comparison of the clustering results for our resilient approach on `Qinghai Small` and `Qinghai Large` also indicates that the influence of the radiation rate increases with increasing size of the data set. An explanation for this behavior might be that even a single bit flip can have a significant influence on the final clustering result: for instance, if the exponent of a floating point representation is corrupted. This issue is illustrated in Fig. 5: here, for both `k-means` and `k-d k-means`, one of the two final centroids has only a single training pattern assigned to it. The latter outcomes supposedly stem from 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 will (only) have this single point assigned to it for the remaining iterations. For `resilient k-d k-means`, the clustering result is not affected by the memory corruptions.

Since the probability for memory corruptions (given a fixed radiation rate) increases with increasing size of the data set (both with respect to *n* and *d*), we expect worse results for clustering larger data sets with non-resilient algorithms.

In the remaining experiments, we attempt to give more insight into the resilient *k*-d *k*-means implementation. To ensure that the experimental results are as insightful as possible, the experiments are conducted only on the largest dataset available, namely `Qinghai Large`. Also, we fix the parameters on which the resilient *k*-d *k*-means runs as follows. We

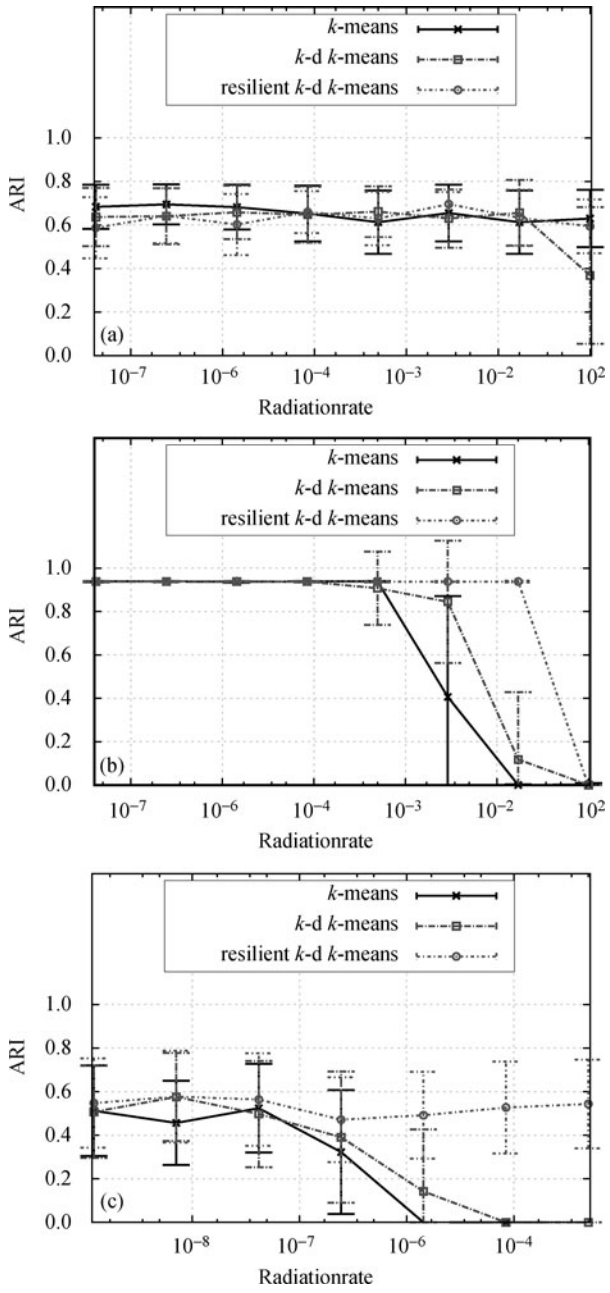

Fig. 4 Clustering performance (adjusted rand index) of all approaches under the disturbance of memory corruptions (determined by the radiation rate): (a) `Iris`, (b) `Qinghai Small`, and (c) `Qinghai Large`

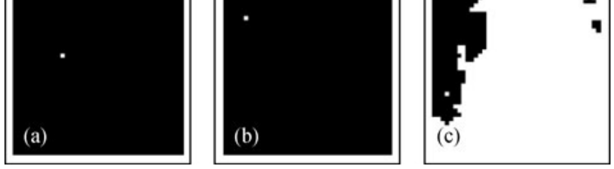

Fig. 5 Final clustering result on the `Qinghai Small` data set for (a) `k-means`, (b) `k-d k-means`, and (c) `resilient k-d k-means` given a fixed radiation rate of 10^{-2}

set $\delta = 100$ and, to ensure that the number of corruptions is smaller than δ, the rate at which memory corruptions occur is set to 10^{-8}.

4.2.2 Tradeoff parameter b

As previously discussed, the parameter b imposes a tradeoff. The larger b is the more points a leaf structure contains, and this in turn leads to fewer nodes in the k-d tree. This implies a smaller space consumption but also higher run 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 run time and memory usage. The results summarized in Fig. 6(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 $b = 10$ which results in a moderate memory requirement and fast run times.

4.2.3 Faster execution 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 *confidence* $= \delta + 1$ in Algorithm 3, this means that halting its execution at a value $2 \leqslant$ *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 sped up significantly. Since we are not willing to trade accuracy for run time, we study experimentally how this parameter affects both the run time and the ARI. The results in Fig. 6(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.

We now analyze the algorithm to retrieve a reliable value, i.e., Algorithm 3 returning when the confidence value reaches a pre-defined threshold t. We show in Lemma 5 that even for small values of t it is extremely unlikely that the returned value is corrupted.

Lemma 5 Assuming corruptions occur uniformly at random, consider that we retrieve a reliable value by returning a candidate when confidence *confidence* $= t$, for some predefined t. The probability that the retrieved value is corrupted is at most $\frac{(2\delta - t + 3)^2}{2} \cdot (\delta/S)^t$, where S is the total amount of space (in cells) used.

Proof Assuming corruptions occur uniformly at random, since there can be at most δ corruptions the probability that a particular memory cell is corrupted is δ/S.

We analyze the cases when the value returned is corrupted. If the corrupted copies are consecutive, then t of them suffice to return a corrupted value. There are $2\delta + 1 - t + 1$ such possibilities, each of them occuring with probability $(\delta/S)^t$.

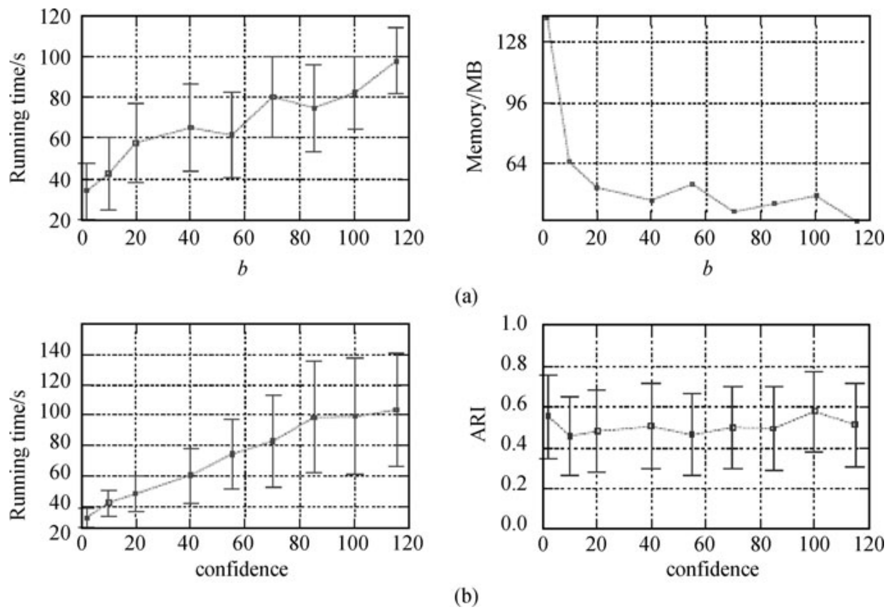

Fig. 6 Dependencies of running time, memory usage, and clustering performance on (a) the parameter b and (b) the confidence value

We turn to the case when the corrupted copies are not consecutive. Let g be the number of gaps between the corrupted values. In this case we need $t + g$ values to be corrupted (to the same value) for the output to be wrong. The sub-sequence returning the wrong value thus has length $2g + t$, there are $\binom{2g+t}{g}$ possibilities to distribute the gaps inside, and this sub-sequence can be placed at $2\delta + 1 - (2g + t) + 1$ locations. Since $t + g$ values need to be corrupted, the probability in each case is $(\delta/S)^{g+t}$. We conclude that when there are g gaps the probability that a corrupted value is returned is at most $\binom{2g+t}{g} \cdot (2\delta + 1 - 2g - t + 1) \cdot (\delta/S)^{g+t}$. Using $2g + t \leqslant 2\delta + 1$, we obtain $g \leqslant \frac{2\delta+1-t}{2}$. We obtain that the probability that the returned value is wrong is bounded by

$$\sum_{g=0}^{\frac{2\delta-t+1}{2}} \binom{2g+t}{g} \cdot (2\delta + 1 - 2g - t + 1) \cdot (\delta/S)^{g+t}.$$

Consider the function $f(g) = \binom{2g+t}{g} \cdot (2\delta + 1 - 2g - t + 1) \cdot (\delta/S)^{g+t}$. For $\delta/S \leqslant 1/(4t)$, we have that

$$\begin{aligned}\frac{f(g+1)}{f(g)} &< \frac{(2g+t+1)(2g+t+2)}{(g+1)(g+t+1)} \cdot \frac{\delta}{S}\\ &\leqslant \frac{(2g+t+1)(2g+t+2)}{4t(g+1)(g+t+1)}\\ &\leqslant 1.\end{aligned}$$

Therefore, for $\delta/S \leqslant 1/(4t)$ the function f is decreasing, which yields that the probability that a false value is returned is at most $(1 + \frac{2\delta-t+1}{2}) \cdot \binom{t}{0} \cdot (2\delta + 1 - t + 1) \cdot (\delta/S)^t \leqslant \frac{(2\delta-t+3)^2}{2} \cdot (\delta/S)^t$.

Note that in the proof of the previous lemma we do not require that the corrupted values have the same value. Even so, for usual values of δ, S, and t the probability that a wrong value is returned is extremely small. For instance, using $S = 32\,\text{MB} = 2^{28}$ (see Fig. 6(a)), $\delta = 128$, and $t = 3$, the probability is at most 2^{-48}.

4.2.4 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 Fig. 7 show that the ARI is not affected when more corruptions than δ occur, 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. Second, 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.

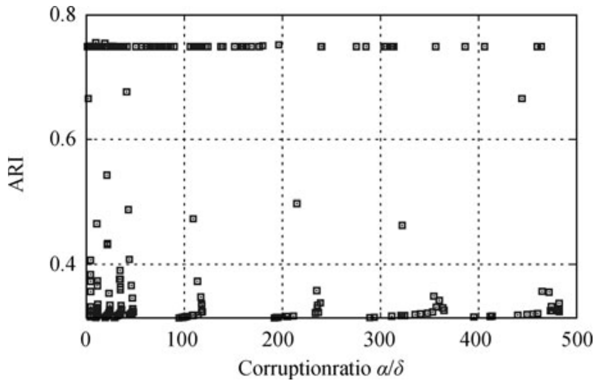

Fig. 7 Performing more than δ corruptions: The x-axis shows the corruption ratio α/δ. The best k-means-based clustering gives an ARI of ≈ 0.749 (due to the clouds' shadows', see Fig. 2), the lower bound is due to our error correction

5 Conclusions

We have proposed a resilient version of the k-d tree data structure and shown how to incorporate it into the k-d tree enhanced k-means clustering approach. The experiments indicate that the resulting (resilient) clustering approach exhibits a superior clustering accuracy under the influence of memory corruptions, even though competing methods have been hand-tuned to avoid standard corruption failures such as segmentation faults. This seems to be especially the case for large data sets in low-dimensional feature spaces (like the `Qinghai Large` data set). In these scenarios, the negative effect of memory corruptions seems to be even more severe since single-bit flips can jeopardize the overall outcome of a (clustering) algorithm.

Acknowledgements Fabian Gieseke is supported in part by the DFG grant KR 3695. Gabriel Moruz is partially supported by the DFG grants ME 3250/1-3 and MO 2057/1-1, and by MADALGO. Jan Vahrenhold is supported in part by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis" (http://sfb876.tu-dortmund.de), project A2.

References

1. Castano R, Mazzoni D, Tang N, Doggett T, Chien S, Greeley R, Cichy B, Davies A. Learning classifiers for science event detection in remote sensing imagery. In: Proceedings of the 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space. 2005
2. Wagstaff K L, Bornstein B. K-means in space: a radiation sensitivity evaluation. In: Proceedings of the 26th International Conference on Machine Learning. 2009, 1097–1104
3. Wagstaff K L, Bornstein B. How much memory radiation protection do onboard machine learning algorithms require? In: Proceedings of the IJCAI-09/SMC-IT-09/IWPSS-09 Workshop on Artificial Intelligence in Space. 2009
4. May T C, Woods M H. Alpha-particle-induced soft errors in dynamic memories. IEEE Transactions on Electron Devices, 1979, 26(1): 2–9

5. Kopetz H. Mitigation of transient faults at the system level — the TTA approach. In: Proceedings of the 2nd Workshop on System Effects of Logic Soft Errors, 2006
6. Finocchi I, Grandoni F, Italiano G F. Designing reliable algorithms in unreliable memories. Computer Science Review, 2007, 1(2): 77–87
7. MacQueen J B. Some methods for classification and analysis of multivariate observations. In: Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability. 1967, 281–297
8. Vattani A. *k*-means requires exponentially many iterations even in the plane. In: Proceedings of the 25th Annual Symposium on Computational Geometry. 2009, 324–332
9. Arthur D, Manthey B, Röglin H. *k*-means has polynomial smoothed complexity. In: Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science. 2009, 405–414
10. Ding C, He X. *k*-means clustering via principal component analysis. In: Proceedings of the 21st International Conference on Machine Learning. 2004, 225–232
11. Elkan C. Using the triangle inequality to accelerate *k*-means. In: Proceedings of the 20th International Conference on Machine Learning. 2003, 147–153
12. Frahling G, Sohler C. A fast *k*-means implementation using coresets. International Journal of Computational Geometry and Applications, 2008, 18(6): 605–625
13. Jin R, Goswami A, Agrawal G. Fast and exact out-of-core and distributed *k*-means clustering. Knowledge and Information Systems, 2006, 10(1): 17–40
14. Kanungo T, Mount D M, Netanyahu N S, Piatko C D, Silverman R, Wu A Y. An efficient *k*-means clustering algorithm: analysis and implementation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2002, 24(7): 881–892
15. Sakuma J, Kobayashi S. Large-scale *k*-means clustering with user-centric privacy-preservation. Knowledge and Information Systems, 2010, 25(2): 253–279
16. Bentley J L. Multidimensional binary search trees used for associative searching. Communications of the ACM, 1975, 18(9): 509–517
17. de Berg M, Cheong O, van Kreveld M, Overmars M. Computational Geometry: Algorithms and Applications. 3rd ed. Santa Clara: Springer-Verlag, 2008
18. Dickerson M, Duncan C A, Goodrich M T. *k*-d trees are better when cut on the longest side. In: Proceedings of the 8th Annual European Symposium. 2000, 179–190
19. Brodal G S, Fagerberg R, Finocchi I, Grandoni F, Italiano G, Jørgensen A G, Moruz G, Mølhave T. Optimal resilient dynamic dictionaries. In: Proceedings of the 15th Annual European Symposium on Algorithms. 2007, 347–358
20. Finocchi I, Grandoni F, Italiano G F. Optimal resilient sorting and searching in the presence of dynamic memory faults. Theoretical Computer Science, 2009, 410(44): 4457–4470
21. Jørgensen A G, Moruz G, Mølhave T. Priority queues resilient to memory faults. In: Proceedings of the 10th International Workshop on Algorithms and Data Structure. 2007, 127–138
22. Petrillo U F, Finocchi I, Italiano G F. The price of resiliency: a case study on sorting with memory faults. In: Proceedings of the 14th Annual European Symposium on Algorithms. 2006, 768–779
23. Caminiti S, Finocchi I, Fusco E G. Local dependency dynamic programming in the presence of memory faults. In: Proceedings of the 28th International Symposium on Theoretical Aspects of Computer Science. 2011, 45–56
24. Brodal G S, Jørgensen A G, Moruz G, Mølhave T. Counting in the presence of memory faults. In: Proceedings of the 20th Annual International Symposium on Algorithms and Computation. 2009, 842–851
25. Brodal G S, Jørgensen A G, Mølhave T. Fault tolerant external memory algorithms. In: Proceedings of the 11th International Symposium on Algorithms and Data Structures. 2009, 411–422
26. Boyer R S, Moore J S. MJRTY: a fast majority vote algorithm. In: Automated Reasoning: Essays in Honor of Woody Bledsoe. 1991, 105–118
27. Bose P, Maheshwari A, Morin P, Morrison J, Smid M, Vahrenhold J. Space-efficient geometric divide-and-conquer algorithms. Computational Geometry: Theory and Applications, 2007, 37(3): 209–227
28. Frank A, Asuncion A. UCI machine learning repository. 2010, http://archive.ics.uci.edu/ml
29. Hubert L, Arabie P. Comparing partitions. Journal of Classification, 1985, 2(1): 193–218

Fabian Gieseke received his Diplomas in Mathematics and Computer Science in 2006 from the University of Münster, Germany. He is currently working towards his PhD in the field of machine learning. His research interests include various related topics including support vector machines and its extensions to semi- and unsupervised learning settings as well as applications of machine learning techniques to problems in the field of astronomy.

Gabriel Moruz received his PhD in Computer Science from the University of Aarhus in 2007. Since 2007 he has been a post doctoral researcher in the group of Prof. Ulrich Meyer (Chair for Algorithm Engineering) at the Goethe University, Frankfurt am Main. His primary research interest concerns the design of efficient algorithms and data structures in practice. This involves designing and implementing algorithms that are aware of various hardware issues that have a great impact in practice. These include cacheaware and cache-oblivious algorithms, streaming algorithms, algorithms performing few branch mispredictions, as well as resilient algorithms, i.e., algorithms aware of memory corruption.

Jan Vahrenhold received his Diploma in Mathematics and his PhD and Habilitation in Computer Science from the University of Münster, Germany, in 1996, 1999, and 2004, respectively. Since 2006, he has been an Associate Professor with the Faculty of Computer Science, Technische Universität Dortmund, Germany. His current research interests include I/O-and cache-efficient algorithms and data structures, computational geometry, and computer science education.