Alcom-FT Technical Report Series
ALCOMFT-TR-01-132

Soft Kinetic Data Structures

Artur Czumaj* and Christian Sohler†

Abstract

We introduce the framework of *soft kinetic data structures (SKDS)*. A soft kinetic data structure is an approximate data structure that can be used to answer queries on a set of moving objects with unpredictable motion. We analyze the quality of a soft kinetic data structure by giving a competitive analysis with respect to the dynamics of the system.

We illustrate our approach by presenting soft kinetic data structures for maintaining classical data structures: sorted arrays, balanced search trees, heaps, and range trees. We also describe soft kinetic data structures for maintaining the Euclidean minimum spanning trees.

1 Introduction.

The need of storing and processing continuously moving data arises in a broad variety of applications, including weather forecast, geographic information systems, air-traffic control, and telecommunications applications. The classical dynamic data structures, which assume that the data changes only at some explicitly given time steps, are not suitable for processing continuously moving objects, because in order to be of some value they must be continuously updated. This is clearly an inefficient and infeasible solution considering the prohibitively large update overhead. An alternative approach, called *kinetic data structures* [6], has been recently proposed in the context of computational geometry (see also [21]). In kinetic data structures one assumes that the motions of the objects are parameterizable by (pseudo-)algebraic functions (typically linear, or low-degree polynomial) of time, so that the positions of the objects change without any explicit modification in the input data. (It is allowed however, that the motion function can be modified, in which case an "explicit" modification in the database is reported.) A typical example for a kinetic data structure is to maintain the closest pair of balls in a billiard simulation [5]. In such an application the closest pair of balls may change at certain discrete points of time which are called *(external) events*. Possible future events are stored in an event queue and a kinetic data structure always processes the next event in the event queue. It may be necessary to have additional events that are needed to keep control of the system. These events are called *internal*.

In recent years kinetic data structures have been applied to many problems (see, e.g., [2, 5, 6, 21]). The previous research has focused on the case when the objects motion is described by some "simple" functions which are known to the system. In many applications, however, the motion of the objects is either completely *unpredictable* in time or is *unknown* to the system (think, for example, on the car motion in a traffic control systems). In such a case one has to deal with the motion in the *on-line* fashion. Further, because of the massive input data, it is typically infeasible to process all data in

*Department of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ 07102-1982, USA. Email: `czumaj@cis.njit.edu`. WWW: `www.cis.njit.edu/~czumaj`. Research supported in part by an SBR grant.

†Heinz Nixdorf Institute and Department of Mathematics & Computer Science, University of Paderborn, D-33095 Paderborn, Germany. Email: `csohler@uni-paderborn.de`. Research supported in part by DFG Grant Me872/7-1 and by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

the system (as it is, e.g., in real-time systems) and therefore the only possible solution is to provide approximate answers to the queries about the system.

Of course, since we do not make any assumptions about the motion changes, it is possible that the changes in the system are so large that it is difficult to provide in a reasonable time even an approximate information about the system. Therefore we want to measure the quality of the algorithms depending on the *dynamics of the system*: if the system is very dynamic and very many changes has been performed, the solutions will be slower; but if there are very few modifications, then we aim at very quick solutions. To measure this quantity we shall use the *competitive* analysis of the algorithms: we shall measure how good the algorithms are if we compare them to the algorithms which work in best possible ways.

Unlike classical data structures, soft kinetic data structures provide only approximate answers with accuracy guarantees. In systems with low dynamics this allows us, however, order of magnitude faster responses over classical data structures. Additionally, by providing answers with accuracy guarantees as fast as one wants them, one can continue computations within an allotted time frame for increasingly accurate answers.

1.1 Definition of Soft Kinetic Data Structures.

In this section we describe the framework of soft kinetic data structures. A soft kinetic data structure is an approximate data structure that answers queries on a set of objects that move in a totally unknown way. In our data structure each object is referenced by an *object identifier*. Using the object identifier we can query the current position of the corresponding object. Queries to our data structures are typically access or search queries. For example, when we use soft kinetic sorted arrays we may query for the ith largest object in the array, and a soft kinetic Euclidean minimum spanning tree (EMST) supports the usual graph operations on the EMST.

Each time before a query to the data structure is processed we run a procedure called the *data structure reorganizer*. The data structure reorganizer checks whether the current status of the data structure is *almost correct*. If it is not, then the reorganizer will repair errors in the data structure until it is almost correct. Clearly, the amount of work that has to be done by the reorganizer depends on the dynamics of the system. To be able to analyze the quality of our data structures we compare the time we need to process a sequence of queries with the dynamics of the system.

1.2 Capturing the Objects' Dynamics.

In this section we explain how we measure the dynamics of the objects. We are given a system of moving objects. The dynamic behavior of the system can be described as an unknown function of time Φ that maps a point of time to a configuration of objects. Let $\langle Q \rangle = Q_1, ..., Q_k$ be a sequence of chronological queries to our data structure. We assume that a query Q_i is answered at a certain *point of time* which we denote by $T(Q_i)$. Our sequence of queries $\langle Q \rangle$ induces a sequence of configurations of objects $\langle C \rangle = C_1, \dots, C_k$, where $C_i = \Phi(T(Q_i))$ denotes the status of the system when query Q_i is processed. One can think of a system as being static, if most pairs of consecutive configurations are "close" to each other and it is dynamic, if most pairs are "far" from each other. In the following we explain how we measure the dynamics in a formal way. We assume the objects are moving continuously from C_i to C_{i+1} using the "cheapest" possible motion. To be able to say whether a motion is "cheap" we define the cost of a motion in the spirit of kinetic data structures in the following way:

Definition 1.1 Let C_i and C_{i+1} be two configurations of objects and let D be a correct data structure for configuration C_i. We assume that the objects are moving continuously from configuration C_i to

C_{i+1}. At certain discrete points of time we have to change the data structure, because some invariant does not hold any longer. Then the cost $c(C_i, C_{i+1})$ of a motion of objects from configuration C_i to C_{i+1} is the minimum number (over all continuous motions) of these topological changes in D, if the objects are moving continuously from configuration C_i to C_{i+1} and D is always correct.

Note that, although we assume for the analysis that objects are moving continuously, we do not necessarily have to deal with continuously changing data.

The cost of a sequence of configurations is the cheapest motion from the first configuration to the second, from the second to the third and so on. Thus it is the sum of the cost of the cheapest motion between each pair of consecutive configurations.

Definition 1.2 The cost $c(\langle C \rangle)$ of a sequence of configurations $\langle C \rangle = C_1, \ldots, C_k$ is $\sum_i c(C_i, C_{i+1})$.

Finally, the cost of a sequence of queries is the cost of the induced sequence of configurations. Note that, in some sense, *the cost of a sequence of queries is a lower bound on the work that has to be done to reorganize the data structure before each query*. We will therefore compare the time needed to answer a sequence of queries with the cost of the sequence as it has just been defined. This will be done in form of a competitive ratio.

1.3 Notion of Approximation.

As already mentioned, it is impossible to use exact data structures in our model unless we always query the position of all objects before each query is processed. Similarly, it seems to be meaningless to use a classical notion of approximation when we consider problems like the Euclidean Minimum Spanning Tree. The reason is that in such problems already a very small change in the structure (e.g., inserting or deleting a single edge) can change the total cost very significantly, and therefore without looking at almost all edges we cannot guarantee any reasonable good approximation. We therefore consider a kind of combinatorial approximation as it was used before in the context of property testing and spot checking (see, e.g., the survey work [19]). We define a function that measures the error of a data structure and says that it the structure is *almost correct* (ϵ-close) if the error of the structure is less than a given threshold. Both, the function and the threshold depend on a parameter ϵ. Typically, the error function counts the edit-distance or Hamming distance to a correct data structure and the threshold is ϵn. If the error of the data structure is larger than the threshold, then we say it is ϵ-far from correct.

The goal of soft kinetic data structures is to ensure w.h.p., that each time before a query is processed the data structure is almost correct.

For a given sequence of queries $\langle Q \rangle = Q_1, \ldots, Q_k$ we will aim at the running time of soft kinetic data structure to be sublinear in $k n$ and close to $c(\langle Q \rangle)$. In the theoretical model we are allowed to act each time, before a query is passed to the structure.

Definition 1.3 A soft kinetic data structure D is d-competitive, if for any sequence of queries $\langle Q \rangle = Q_1, \ldots, Q_k$ the expected time (the sum over the whole sequence) needed to keep D close to correct is $\mathcal{O}(d \cdot (c(\langle Q \rangle) + k))$.

1.4 A Generic Reorganizer.

In this section we describe how we can guarantee that with high probability the data structure is almost correct when we process a query. First we need a procedure that tells us whether the data structure is almost correct or not. This procedure is called a *spot checker* or a *property tester* (see [10, 11, 12, 15, 20]). A spot checker samples a small set of objects from our data structure w.r.t. a (possibly non-uniform) probability distribution and checks whether the *current* values of these objects do not violate some invariant of the data structure. If they do, we have found an error in the data structure which has to be fixed. The choice of the sample must be in such that, if the structure is ϵ-far from correct, the data structure is rejected with probability at least $\frac{2}{3}$. The following algorithm achieves with high probability that the structure is close to the correct one after the algorithm has finished its correction. Assume the structure is ϵ-far from correct. Then the property tester rejects and finds an error in the structure with probability at least $\frac{2}{3}$. Using standard amplification techniques we achieve a confidence probability of $\frac{9}{10}$. We prove below that the following algorithm guarantees w.h.p. that a data structure is ϵ-close to correct after the algorithm was called:

REORGANIZER(D)
  if is_not_correct(D) **then**
    Repair the error that has been found
    REORGANIZER(D)
    REORGANIZER(D)

Theorem 1 *Algorithms* REORGANIZER() *guarantees with probability* $\frac{2}{3}$ *that the data structure is* ϵ*-close to correct, if the property tester finds errors with probability* $\frac{9}{10}$ *when the data structure is* ϵ*-far from correct.*

Proof : Assume the data structure is ϵ-far from correct. We want to analyze the probability that REORGANIZER() stops when the data structure is ϵ-far from correct. This process can be viewed as a binary tree. Each node corresponds to a call of REORGANIZER() that rejects the current data structure and it has a left (right) child if the first (second) recursive call rejects the data structure. For example, if the first call to REORGANIZER() accepts the input, then the corresponding tree is the empty tree and it appears with probability $\frac{1}{10}$. It is well known that the number of binary trees with k nodes is less than 4^k. Let T be a tree with k nodes. Then the probability that REORGANIZER() behaves like the tree is $(\frac{9}{10})^k \cdot (\frac{1}{10})^{k+1}$ because each binary tree with k nodes has $k+1$ empty leafs. The overall probability that the process stops before the structure is ϵ-close to correct is bounded from the above by $\sum_{0\le i\le\infty} 4^i(\frac{9}{10})^i(\frac{1}{10})^{i+1} \le \frac{1}{6}$. □

Remark 1 *Note that in a real time system the reorganizer may run as a background process. The approximation parameter* ϵ *may be automatically adjusted to the dynamics of the system and the available resources.*

1.5 An Example: Sorted Arrays.

As an illustrative example we consider soft kinetic sorted arrays. Let $\mathcal{O}$ be a set of n moving objects in $\mathbb{R}$. The object identifiers of $\mathcal{O}$ are stored in an array $A[1\cdots n]$. When we consider continuous motion the topological structure of an exact array changes when the values v_1, v_2 of two objects change from $v_1 < v_2$ to $v_1 > v_2$. Under the assumption of continuous motion the two corresponding object identifiers must be adjacent in the array and will be swapped. Below we design a soft kinetic version of this structure.

We have a sequence of queries $Q = Q_1, \ldots, Q_k$ to our array and before proceeding each query we want to ensure that our array is nearly sorted. To check whether the array is close to a sorted array, we run the spot-checker from [10]. The array is rejected if it is ϵ-far from sorted (the distance measure used is edit distance). If the array is rejected the spot checker from [10] returns a pair of indices (k, l) with $A[k] > A[l]$ and $k < l$.

Observation 1.1 *The cost of two consecutive queries Q_1, Q_2 to a soft kinetic sorted array is the number of swaps needed to transform a correctly sorted array A for configuration $\Phi(T(Q_1))$ to a correctly sorted array for configuration $\Phi(T(Q_2))$. This is equal to the number of inversion in the array A at time $T(Q_2)$.*

From the pair of indices (k, l) returned by the property tester we can compute a pair $(k', k'+1)$ with $A[k'] > A[k'+1]$ by a binary search like procedure. As long as $k \neq l+1$ we compute $m = \lceil \frac{k+l}{2} \rceil$ and proceed either with (k, m) or with (m, l) depending on the value of $A[m]$. Finally, we swap $A[k']$ and $A[k'+1]$. Below we state the algorithm for the soft kinetic sorted array that is called before a query is answered:

```
SKArray(A)
    if SortedTest()= FALSE then
        Let (k, l) be the pair returned by SortedTest()
        k'=FindInversion(k, l)
        swap(A[k'], A[k' + 1])
        SKArray(A)
        SKArray(A)
```

Theorem 2 *There is a soft kinetic sorted array that is $\mathcal{O}(\frac{\log n}{\epsilon})$-competitive.*

Proof :

We have to prove that the distance from a correct structure in the 'property testing distance' is monotone decreasing and that the distance in terms of events is decreasing by 1 each time an error is fixed.

It is obvious that the edit distance to a correctly sorted array cannot increase when we correct an inversion. Thus we will now consider the distance in terms of events and the running time. A single call to the reorganizer takes time $\mathcal{O}(\frac{\log n}{\epsilon} \cdot k)$ where k is the number of corrected errors. Any error correction reduces the distance to a correctly sorted array by 1. By the triangle inequality it follows that the overall time to process a sequence $\langle Q \rangle$ of queries is at most $c(\langle Q \rangle) \cdot \frac{\log n}{\epsilon}$. □

2 Basic Data Structures.

In this section we present soft kinetic versions of some classical data structures.

2.1 Balanced Search Trees.

We begin our discussion with soft kinetic search trees.

A (balanced) binary search tree is a rooted tree in which each node z stores a key $\text{KEY}(z)$ (wlog. we assume this is a real number) and which satisfies the following property: For any node x in the tree and any node y in the left (right, respectively) subtree of x, it holds that $\text{KEY}(x) \geq \text{KEY}(y)$ ($\text{KEY}(x) \leq \text{KEY}(y)$, respectively). In the case of soft kinetic search trees each node z stores an object identifier which itself contains the key. This key may change in an unpredictable way as a function of time. We first suppose that only the standard *access* (or search/membership) operation is to be implemented.

We observe that a binary search tree is correct, if the keys are in increasing order from the leftmost node to the rightmost one in the in-order tree traversal. Therefore our goal is similar to that in sorted arrays: we want to maintain a sorted sequence of keys. We consider also the same events as in sorting, namely, an even occur when two keys change their relative order. We observe also that it does not seem to be a good idea to measure the quality of the search tree by its edit distance to a correct search tree. This is because the most important tree operation is the *access* operation. There are search trees whose edit distance to a correct tree is very small but almost no *access* operation will work correctly. Consider for example a balanced search tree and exchange the root with the smallest element of the tree. then roughly half of the *access* operations will fail, but the edit distance to a correct tree is only constant. We therefore shall use another, more sensitive definition.

Definition 2.1 A search tree is ϵ-far from correct, if at least $\epsilon\, n$ access operations fail.

The next thing we need is a property tester for the invariant of the search tree. But this is simple for balanced search trees with the distance measure from above. We assume that the nodes of the search tree are stored in an array and we may sample (in constant time) nodes from this array. The following is a spot checker for the above distance measure: *sample a set of $\mathcal{O}(1/\epsilon)$ nodes and check whether they are accessible by following pointers to their parent towards the root.* At each node we verify the correctness of the search path. Let $\prec_T$ denote the linear order induced by the tree. If the search path is incorrect, then we find two nodes v_1, v_2 such that $v_1 \prec_T v_2$ and $v_1 > v_2$.

Then we proceed similar to the sorted arrays. We use a binary search on the tree nodes to find 2 adjacent nodes x, y with $x \succ_T y$ and $x < y$. In order to perform this operation in a binary tree efficiently we maintain the size of each subtree. Then we swap x and y. Unfortunately, swapping two adjacent elements may increase the distance to a correct data structure in terms of property testing. Thus the analysis of the generic reorganizer does not hold any longer. Therefore, we use standard amplification techniques to put the confidence that the tester from above accepts an input that is far from correct to $\Theta(1/n^2)$. A single call to the tester will then require $\mathcal{O}(\log^2 n/\epsilon)$ time. We need $\mathcal{O}(\log n)$ time to perform the access operations and we have to repeat the test $\mathcal{O}(\log n)$ times to ensure the high confidence probability we need. The time for the binary search is $\mathcal{O}(\log^2 n)$. This leads to the following theorem.

Theorem 3 *There is a soft kinetic balanced search tree that is $\mathcal{O}(\frac{\log^2 n}{\epsilon})$-competitive.*

Now we consider two dynamic operations on trees: *insert* and *delete*. We focus here on binary search trees whose updates are based on rotations (this includes, for example, AVL-trees). Thus in order to implement *insert* and *delete* operations we have to implement rotations.

We call two elements x and y *corrupted* if $x \prec_T y$ and $x > y$, or $y \prec_T x$ and $y > x$. We prove that rotations do not increase the overall error, if the two top elements of the rotation are not corrupted (the two top elements are x and y in Figure 1).

Lemma 2.1 *Rotations do not change the number of errors in the structure unless the order of the two top elements is corrupted.*

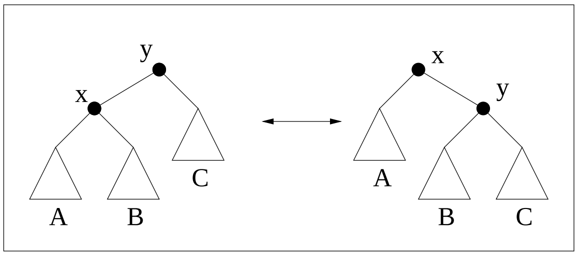

Figure 1: Illustration to the proof of Lemma 2.1.

Proof : We consider a rotation from the left tree to the right one (cf. Figure 1). The other rotation is similar. Let i be an item that can be accessed. If i is in subtree A, then $i < x$ and $i < y$ holds because i is accessible. Thus, it can also be accessed after the rotation. The same holds for items in subtree B. If an accessible item i is in subtree C we know that $i > y$. Since the two top elements (x and y) are not corrupted, it holds that $x < y$; hence $i < x < y$ and the item can be accessed after the rotation. □

Now let us consider the situation when the top elements are corrupted. We keep a reference to these two elements and continue with rotations. After we finished the *insert/delete* operation, we repair the corrupted elements in a similar way as we did above and then we call the reorganizer. We can amortize the time needed for this call to the reorganizer to the errors we just corrected. This leads to the following theorem:

Theorem 4 *There is a soft kinetic search tree that is $\mathcal{O}(\frac{\log^2 n}{\epsilon})$-competitive and that supports insert and delete operations in $\mathcal{O}(\log n)$ time.*

Remark 2 *Let us notice in this place that an insertion/deletion may increase the distance (in events) to a correct structure by $\mathcal{O}(n)$.*

Remark 3 *Notice that the structure of the search tree does not change during the reorganization process, i.e., a balanced search tree will remain balanced.*

2.2 Binary Heaps.

Below we develop a soft kinetic binary heap. Let us recall that a *binary heap* is a data structure which stores n elements (wlog. we assume these elements have assigned real-valued keys) in an array and which satisfies the *heap property*: for every i, $1 \leq i \leq n$, it holds that $\text{KEY}(\lfloor i/2 \rfloor) \leq \text{KEY}(i)$. We shall also use an implicit tree representation of the heap and for each ith element in the array the element $\lfloor i/2 \rfloor$ will be called $\mathit{parent}[i]$.

2.2.1 Property Tester.

In this subsection we develop a property tester for binary heaps. When shall use the following distance measure (notice here a similarity of this definition with the definition of soft heap due to Chazelle [7] - our notion is however significantly weaker):

Definition 2.2 A binary heap H is ϵ-far from correct, if at least $\epsilon\, n$ keys must be changed to satisfy the heap property in H.

Our property tester works in the following way: We first sample at random a set S of elements in the heap. Then we look at the first $\log(1/\epsilon)$ ancestors of each element chosen. By the heap property, so defined $|S|$ chains must be non-increasing. If one of the chains is not non-increasing, we reject the input heap.

TESTHEAP(H, ϵ)
    Sample set S of $\mathcal{O}(1/\epsilon)$ heap elements at random
    for each element $i \in S$ **do**
        for $i = 1$ **to** $\mathcal{O}(\log(1/\epsilon))$ **do**
            if KEY$(i) <$ KEY$(parent[i])$ **then reject**
            $i = parent[i]$
    accept

To prove the correctness of our algorithm, we will show that there are $\mathcal{O}(\epsilon n)$ distinct pairs of heap elements that

- violate the heap property,
- one element of the pair is ancestor of the other one, and
- the distance between the two elements in the heap is at most $\mathcal{O}(\log\frac{1}{\epsilon})$.

We call such a pair a *violating pair*. For the analysis we introduce a new value $blank$ for the keys. If an element in the heap is assigned the value $blank$ it may be set to any value we like, thus for any key x $blank \le x$ and $blank > x$ are both true. It follows immediately from the definition that if a heap is ϵ-far from correct, we have to set at least ϵn elements to the value $blank$ to fullfill the heap property.

We say that a vertex is *low* in H, if its depth is at least $\log(\epsilon n) - 4$. Let us now assume that heap H is ϵ-far from correct. We show that there are $\mathcal{O}(\epsilon n)$ violating pairs. In a first step we set all $\epsilon n/16$ keys of elements that are not low to $blank$. If no pair (x, y) of elements remains s.t. x is ancestor of y and (x, y) violates the heap property, we are close to a correct heap which is a contradiction. Thus there exists such a pair. Also note that both x and y must be low in the heap. Thus the distance between them is at most $\mathcal{O}(\log\frac{1}{\epsilon})$ and we conclude that (x, y) is a violating pair. Then we set x and y to $blank$ and apply our argument again. This way we can construct $\epsilon n\frac{15}{32} = \mathcal{O}(\epsilon n)$ violating pairs. when we sample the lower element of the violating pair, then our algorithm will find a conflict in the heap. Thus it is sufficient to sample $\mathcal{O}(\frac{1}{\epsilon})$ elements. We have just proven:

Theorem 5 *Algorithm* TESTHEAP() *is a property tester for the heap property of binary heaps. Its running time is* $\mathcal{O}(1/\epsilon \cdot \log(1/\epsilon))$.

2.2.2 Analyzing the Competitive Ratio.

Unlike in the case of total orders there are many different partial orders that satisfy the heap property. This leads to the question how we compare our data structure to a correct sequence of data structures. One way to deal with this problem would be to require to make the heap competitive to any correct sequence of heaps. But this seems to much to ask for. We therefore only require that the amount of work we do must be comparable to the minimum number of events to transform the current heap into a correct one.

Similarly to the sorted array, we analyze the competitiveness of binary heaps with respect to swaps of two adjacent elements. In this case we may swap an item a with its parent in the heap. Unfortunately, in binary heaps it is not clear whether swapping two conflicting adjacent items does reduce the overall error of the structure.

We concentrate on reducing the error count only among the low vertices in the heap. If we find a pair of low vertices that violates the heap property we recompute the largest sub-heap that contains only low vertices and the conflicting pair. This sub-heap has size $\mathcal{O}(1/\epsilon)$. If we find another conflicting pair, we do nothing. This way we need $\mathcal{O}(1/\epsilon)$ time to reduce the minimal distance to a correct heap by 1.

Theorem 6 *There is a soft kinetic binary heap that is $\mathcal{O}(1/\epsilon \cdot \log(1/\epsilon))$-competitive.*

3 Geometric Data Structures.

In the previous section we have focused our attention on standard elementary data structures. Now we shall consider some soft kinetic data structures from computational geometry.

3.1 1D-Range Trees.

We can use soft kinetic data structures for binary search trees to obtain the following result (see [1] for a formal definition of range trees):

Theorem 7 *There is a soft kinetic version of 1D-range trees such that range queries are supported in $\mathcal{O}(\log n + k)$ time, where k is the number of reported points. The 1D-range tree is $\mathcal{O}(\frac{\log^2 n}{\epsilon})$-competitive. Further, if k^* denotes the number of points in the query range then it holds whp. that $k \geq k^* - \epsilon n$.*

Proof : We have to show that $k \geq k^* - \epsilon n$ holds with probability $2/3$ and we have to prove the running time for the range queries. The other results follow from the binary search trees. First of all, we can assume that the data structure is ϵ-close to correct when the query is processed. This is because before the query the reorganizer is called. A range query for a one dimensional range tree is answered in the following standard way: we search for the right and the left end of the query interval and output all nodes that are between the search paths. Before we report a point we check if it is really inside the query interval. If it is, we output it. If it is not we have found an inversion with some node on the search paths. We can correct this inversion and call the reorganizer. The time for this call as well as the time to process the point is amortized to the corrected error.

Now assume that less than $k^* - \epsilon n$ points are reported. Then there are ϵn nodes that cannot be accessed, because any access operation to one of the missing nodes with end either on the search paths or between them. But these are the reported nodes. Thus we have that the tree is ϵ-far which is a contradiction. $\square$

In a similar way we can prove:

Corollary 3.1 *There exists a soft kinetic 1D-range tree for interval counting queries. The data structure is $\mathcal{O}(\frac{\log n}{\epsilon} + \log^2 n)$-competitive and answers interval counting queries in time $\mathcal{O}(\log n)$ with absolute error ϵ with high probability.*

3.2 2D-Range Trees.

In this section we describe and analyze 2-dimensional soft kinetic range trees (see, e.g., [1] for a formal definition).

We consider a standard implementation of 2D-range trees which can be regarded as a double-level 1D-range tree: there is one 1D-range tree for the first level structure and n 1D-range trees (one for each node in the first level structure) for the second level structure (see, e.g., [1] for more details).

Such a range tree is correct, if the nodes of the first level tree are sorted from left to right in increasing order w.r.t. the x-coordinate of the points and if the nodes of the second level trees are sorted from left to right in increasing order w.r.t. y-coordinate of the points. Thus here we have to maintain two sorted sequences.

Similar to the binary search trees we want to have a data structure that supports range queries in a reasonable way. That is, we want to make sure that, if a structure is ϵ-close to correct, it cannot happen that almost every range query fails.

We say that a point is accessible in a 2D range tree, if it can be accessed in the first level tree using its x-coordinate as key and it can be accessed in *all* second level trees on the search path using its y-coordinate. A 2D range tree is ϵ-far from correct, if there are at least ϵn points that are not accessible. Similar to the binary search trees we can design a simple property tester. We sample a set of $\mathcal{O}(1/\epsilon)$ nodes and check whether they are accessible by following the parents pointers in the trees. We also use two arrays as auxiliary data structures. One of them is used to define the order of objects in the first level of the range tree and the other one defines the order in the second level. When the property tester rejects, it provides a proof that one of these arrays is not in the correct order and we can find an adjacent pair that is in the wrong order. Then we swap these two objects in the corresponding array and we adjust the range tree to the changes we made. To be able to find the corresponding nodes in the range tree we maintain pointers to all occurrences of a point in the second level trees. Then we can adjust the range tree in $\mathcal{O}(\log n)$ time, if there is an error in the y-order of the points. If we swap two elements in the first level tree, we delete all occurrences in the second level tree. Then we swap the two nodes in the first level tree and reinsert the points into the second level trees. This can be done by using the position in the maintained array as a key. This procedure can be done in $\mathcal{O}(\log^2 n)$ time. Checking whether a point is accessible takes $O(\log^2 n)$ time. We again need that the test accepts a far input with probability $\Theta(\frac{1}{n^2})$ thus we need $O(\log^3 n/\epsilon)$ time for the test.

We conclude:

Theorem 8 *There is a soft kinetic version of 2D-range trees such that range queries are supported in $\mathcal{O}(\log^2 n + k)$ time, where k is the number of reported points. The 2D-range tree is $\mathcal{O}(\frac{\log^3 n}{\epsilon})$-competitive. Furthermore, if k^* denotes the number of points in the query range then w.h.p.* $k \geq k^* - \epsilon n$.

Proof : The analysis of the running time of the queries is similar to the 1D case. And again, if there is a query that returns less than $k^* - \epsilon n$ points, then the structure must be ϵ-far which is a contradiction. □

3.3 Euclidean Minimum Spanning Tree.

In this section we design a soft kinetic data structure for *Euclidean Minimum Spanning Tree* (EMST). Let us first define a distance measure for the EMST (cf. also [8]):

Definition 3.1 An input graph is ϵ-far from the EMST if the Hamming distance to the EMST is at least $\epsilon\, n$.

In [8], an $\mathcal{O}(\sqrt{n/\epsilon} \cdot \log^2(1/\epsilon) \cdot \log n)$-time property tester for the EMST has been presented. (That is, the property tester takes as an input a point set P and an Euclidean graph G on P, and then it verifies whether G is not ϵ-far from the EMST of P.) We briefly recall here this algorithm. A spot checker for the EMST first checks with two property testers whether the straight-line embedding of the input graph is crossing-free and whether its edges cross edges of the correct EMST for the given point set which is stored at the vertices of the input graph. If the input graph passes both tests, then the graph obtained by the edges of the input graph together with all edges of the correct EMST is almost planar. We can then explore (almost) planarity by showing that if the graph is ϵ-far from the EMST then we will find a short cycle in the complete Euclidean graph whose longest edge belongs to the input graph.

If we have a connected input graph that is ϵ-far from the EMST at least one of the three tests described above will fail. Thus one can design the tester such that in any case it returns an edge of the input graph which does not belong to the EMST of the point set. We show below that this is sufficient to show the existence of an efficient soft kinetic EMST:

Theorem 9 *There is a soft kinetic Euclidean minimum spanning tree that is* $\mathcal{O}(\sqrt{n/\epsilon} \cdot \log^2(1/\epsilon) \cdot \log n)$*-competitive.*

Sketch of the proof : We maintain a counter for the edges in our graph. Clearly, if an edge does not belong to the EMST, then it must be deleted and it is replaced by an edge that belongs to the EMST. Since the property tester for the EMST does only provide witnesses that an edge is *not* in the EMST we delete such an edge from the graph we maintain and we increase a counter. If the number of deleted edges is greater than $\epsilon\, n/2$ we recompute the EMST of the whole set - note that it is not possible to recompute the EMST without getting a bad approximation ratio, because some errors are detected with rather high probability. Then we run the property tester with parameter $\epsilon' = \epsilon/2$. We amortize the time needed to recompute the EMST and we get the desired competitive ratio. □

References

[1] P. K. Agarwal. Range searching. In *Handbook of Discrete and Computational Geometry*, J. E. Goodman and J. O'Rourke, eds., CRC Press, Boca Raton, FL, 1997.

[2] P. K. Agarwal, D. Eppstein, L. J. Guibas, and M. R. Henzinger. Parametric and kinetic minimum spanning trees. In *Proc. 39th FOCS*, pp. 596–605, 1998.

[3] N. Alon, E. Fischer, M. Krivelevich, and M. Szegedy. Efficient testing of large graphs. In *Proc. 40th FOCS*, pp. 656–666, 1999.

[4] N. Alon, S. Dar, M. Parnas, and D. Ron. Testing of clustering. In *Proc. 41st FOCS*, 2000.

version appeared in *Proc. 22nd STOC*, pp. 73–83, 1990.

[5] J. Basch, L. J. Guibas, and Li Zhang. Proximity problems on moving points. In *Proc. 13th SoCG*, pp. 344–351, 1997.

[6] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *J. Algorithms*, 31(1):1–28, 1999. A preliminary version appeared in *Proc. 8th SODA*, pp. 747–756, 1997.

complexity. version appeared in *Proc. 38th FOCS*, pp. 22–31, 1997.

[7] B. Chazelle. The Soft Heap: An approximate priority queue with optimal error rate. *J. Assoc. Comput. Mach.*, to appear, 2000. A preliminary version appeared in *Proc. 6th ESA*, pp. 35–42, 1998.

[8] A. Czumaj, C. Sohler, and M. Ziegler. Property testing in computation geometry. In *Proc. 8th ESA*, pp. 155–166, 2000.

[9] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay. Clustering in large graphs and matrices. In *Proc. 10th SODA*, pp. 291–299, 1999.

[10] F. Ergün, S. Kannan, S. Ravi Kumar, R. Rubinfeld, and M. Viswanathan. Spot-checkers. *J. Comput. System Sci.*, 60:717–751, 2000. A preliminary version appeared in *Proc. 30th STOC*, pp. 259–268, 1998.

[11] A. Frieze and R. Kannan. Quick approximation to matrices and applications. *Combinatorica*, 19:175–220, 1999.

[12] A. Frieze, R. Kannan, and S. Vempala. Fast Monte-Carlo algorithms for finding low-rank approximations. In *Proc. 39th FOCS*, pp. 370–378, 1998.

[13] F. Ergün, S. Ravi Kumar, and R. Rubinfeld. Approximate checking of polynomials and functional equations. In *Proc. 37th FOCS*, pp. 592–601, 1996.

[14] O. Goldreich, S. Goldwasser, E. Lehman, and D. Ron. Testing monotonicity. In *Proc. 39th FOCS*, pp. 426–435, 1998.

[15] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *J. Assoc. Comput. Mach.*, 45(4):653–750, 1998. A preliminary version appeared in *Proc. 37th FOCS*, pp. 339–348, 1996.

[16] O. Goldreich and D. Ron. Property testing in bounded degree graphs. In *Proc. 29th STOC*, pp. 406–415, 1997.

[17] O. Goldreich and D. Ron. A sublinear bipartiteness tester for bounded degree graphs. *Combinatorica*, 19(3):335–373, 1999. A preliminary version appeared in *Proc. 30th STOC*, pp. 289–298, 1998.

[18] M. Parnas and D. Ron. Testing the diameter of graphs. In *Proc. RANDOM-APPROX'99*, pp. 85–96, LNCS 1671, 1999.

[19] D. Ron. Property testing. In *Handobook of Randomized Algorithms*. Kluwer Academic Publishers, 2000. To appear.

[20] R. Rubinfeld and M. Sudan. Robust characterization of polynomials with applications to program testing. *SIAM J. Comput.*, 25(2):252–271, 1996.

[21] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158-221, 1999.