Semi-Dynamic Closest-Pair Algorithms
(Extended Abstract)

Yossi Matias
AT&T Bell Laboratories
Murray Hill NJ 07974
matias@research.att.com

Abstract

Given a set of points S in $\Re^d$, where $d > 0$ is an arbitrary constant, the *closest-pair* problem is to compute the closest pair of points in S in the Euclidean metric. In the *on-line closest-pair* problem, the points of S are given for insertion one at the time by an adversary, and the closest pair must be computed after each insertion. In a generalized problem, the *β-on-line closest-pair* problem, the points are inserted in batches of size β, and the closest pair must be computed after each insertion. In this paper we give a simple and efficient algorithm for the β-on-line closest-pair problem. Also presented are: an output-sensitive algorithm for the on-line problem, an on-line algorithm for points taken from a bounded universe, a $\{\beta_j\}$-on-line algorithm, where batches are given in different sizes $\beta_1, \beta_2, \ldots$, and a new incremental algorithm for the off-line problem. The *ϵ-closest bichromatic pair* problem is to compute a bichromatic pair in S whose distance is within a $(1+\epsilon)$ factor from the closest bichromatic pair. We present simple and efficient algorithms for the semi-dynamic versions of the ϵ-closest bichromatic pair problem. Also considered is the *on-line-deletions closest-pair* problem, where a set S is initially given and its points are deleted one at the time by an adversary, and the closest pair of the remaining set is computed after each deletion. We show that the on-line-deletions closest pair problem is at least as hard as sorting.

1 Introduction

Given a set S of n points in $\Re^d$, the *closest-pair* problem is to find the closest pair of points in S, in the Euclidean metric. This simply stated problem can be found in almost any algorithms text-book as a basic problem in computation geometry (see, e.g., [6, 19, 22]). The problem has several dynamic variants, in which the set of points S is changing dynamically by inserting new points into S and deleting existing points from S; the problem then is to compute and maintain a data structure that enables efficient closest-pair queries. In this paper we consider several semi-dynamic variants and provide simple and efficient algorithms. We also consider the related problem of *ϵ-closest bichromatic pair*, and provide efficient algorithms for semi-dynamic versions of this problem.

1.1 Related work

The off-line problem For the *off-line closest pair* problem, in which all the points of S are given initially and only the closest pair of S needs to be computed, several algorithms were given. Deterministic algorithms that run in $O(n \lg n)$ time are due to [4, 5, 16, 28]. These algorithms are optimal in the algebraic decision-tree model of computation, where a matching lower bound of $\Omega(n \lg n)$, even for the 1-dimensional closest-pair problem, is implied by a lower bound for element distinctness [2, 33]. The lower bound only holds when the floor function is not allowed, as was shown by Fortune and Hopcroft [10], who obtained an $O(n \lg \lg n)$ deterministic algorithm by making use of the floor function. A linear time randomized algorithm which uses the floor function was given by Rabin in his seminal paper [23]; his algorithm uses a *random sampling* technique. Another linear time randomized algorithm, using a new *sieving* technique, was recently given in [17].

The on-line problem Assume that a set S of n points in $\Re^d$ is given by inserting one point at a time, with the points given dynamically by some arbitrary adversary (possibly adaptive). The *on-line closest pair* problem is to find, after each insertion, the closest pair in S (in the Euclidean metric). There has been a sequence of papers studying the on-line version of the problem. Smid [29] gave an algorithm that computes the closest pairs on-line in $O(n(\lg n)^{d-1})$ total time; the algorithm uses only algebraic functions and is therefore optimal for the planar case in the algebraic model. He also gave an algorithm that using the floor function takes $O(n(\lg n)^2/\lg \lg n)$ time (for any fixed d). Subsequently, Schwarz and Smid [26] gave an $O(n \lg n \lg \lg n)$ time algorithm (for any fixed d), also using the floor function. Finally, very recently Schwarz *et al.* [27] gave an algorithm that takes $O(\lg n)$ amortized time per insertion. Their algorithm uses only algebraic functions and is hence optimal in the amortized algebraic model.

The on-line-deletions problem In the *on-line-deletions closest-pair* problem, the set S is given initially,

and its points are deleted one at the time. After each deletion the closest pair of the new set needs to be computed. Supowit [32] gave an algorithm with $O(\lg^d n)$ amortized update time and $O(n \lg^{d-1} n)$ space.

The fully dynamic problem A more general problem is the fully dynamic closest-pair problem, in which points are inserted into as well as deleted from the set. Smid [31] gave an algorithm which uses $O(n \lg^d n)$ space and takes $O(\lg^d n \lg\lg n)$ amortized time per update. A linear space algorithm, with $O(\sqrt{n} \lg n)$ time per update is a result of the works of Smid [30], Salowe [25], and Dickerson and Drysdale [7]. Very recently, Golin *et al.* [14] presented an algorithm which supports insertions into and deletions from the set in expected $O(\lg n)$ time and requires $O(n)$ expected space. Their algorithm uses the floor function and assumes the updates are chosen by an adversary who does not know the random choices made by the algorithm; they also show how to get an algorithm for the algebraic tree model, with $O(\lg^2 n)$ expected time per insertion.

Approximate closest bichromatic pair A generalization of the closest-pair problem is the following *closest bichromatic pair* problem: Given a set of n colored points in $\Re^d$, for some constant $d > 0$, find the closest pair of points that are colored differently ("bichromatic pair"); i.e., find a point p and a point q that have different colors, such that the distance between p and q is minimum among all the bichromatic pairs. An instance of the problem where each point is colored by one of *two* colors was considered by Agarwal *et al.* [1]. For an input consisting of m red points and n blue points from $\Re^3$ (i.e., for $d = 3$), they give a randomized algorithm running in expected time $O((nm \lg n \lg m)^{2/3} + m \lg^2 n + n \lg^2 m)$. (This has applications in solving the Euclidean minimum spanning tree problem as was shown by [1].) An approximation problem, the *ϵ-closest bichromatic pair* problem, was considered by [17]. In this problem, a bichromatic pair is found whose distance is within a factor of $(1+\epsilon)$ from the distance of the actual closest bichromatic pair. For any fixed $\epsilon > 0$ the algorithm given in [17] takes $O(n)$ expected time and $O(n)$ space. We are not aware of any previous algorithm for the dynamic or semi-dynamic versions of this problem.

1.2 Results

In this paper we give simple and efficient algorithms for various semi-dynamic closest-pair problems.

Batch-on-line algorithms We consider the *β-on-line closest pair* problem, in which points are inserted in batches of size β each. An algorithm is presented that uses $O(n)$ space; inserting the next batch into a set of size n takes $O(\beta \lg(n/\beta))$ time with high probability. (For instance, $\lg n$ batches of size $n/\lg n$ each are inserted on-line in $O(n \lg\lg n)$ total time.)

To our knowledge, this is the first algorithm that deals with this rather natural extension.

An efficient parallel algorithm is given as well in which, using $\beta/\lg^* \beta$ processors on a CRCW-PRAM, a batch is inserted in $O(\lg(n/\beta) \lg^* \beta)$ time with high probability. (For instance, $\lg n$ batches of size $n/\lg n$ each are inserted on-line in $O(\lg\lg n \lg^* n)$ total parallel time, using $n/(\lg n \lg^* n)$ processors.) Note that for $\beta = 1$, we get a parallel on-line algorithm where each insertion takes $O(\lg^* n)$ time and $O(\lg n)$ operations with high probability, using linear space.

A varying-size batch-on-line algorithm Also presented is an algorithm for a generalized problem, the *$\{\beta_j\}$-on-line closest pair* problem, where the batches of inserted points are of possibly different sizes $\beta_1, \beta_2, \ldots, \beta_{n'}$. A point of the j'th batch is inserted in time $O((\lg n - \lg \beta_{j-1}) + (\lg n - \lg \beta_j))$ or in time $O(\lg n')$ with high probability, using linear space.

An output-sensitive algorithm An *output-sensitive* algorithm for the on-line closest-pair problem is presented, in which the insertion time depends on the frequency in which the closest-pair distance actually changes. If we let the set of points inserted between two changes of the closest-pair distance be called a *batch*, then the complexity of the output-sensitive algorithm is the same as the complexity described above for the *$\{\beta_j\}$-on-line closest pair* algorithm. We are not aware of a comparable result in previous papers.

An on-line algorithm with bounded domain We consider the on-line closest pair problem for points taken from a *bounded universe* $\{1, \ldots, u\}^d$. Using the output-sensitive algorithm, we get an algorithm with $O(\lg\lg u)$ time per insertion in the worst case. The algorithm can be implemented in linear space and $O(\lg\lg u)$ time per insertion, with high probability.

An incremental off-line algorithm Using the output-sensitive algorithm, we present a new *incremental* algorithm for the off-line closest pair problem that takes linear expected time. This algorithm complements the previous algorithms, that are based on a random sampling technique and on a sieving technique, with a third different and well known technique for this basic problem. A different incremental algorithm was recently developed in an independent work by Golin et al [15].

Semi-dynamic ϵ-closest bichromatic pair algorithms The algorithms for the closest-pair problems are extended to ϵ-closest bichromatic pair algorithms: we give an on-line algorithm, a β-on-line algorithm, a $\{\beta_j\}$-on-line algorithm, an output-sensitive algorithm, an on-line algorithm in bounded universe, and an incremental off-line algorithm, all with similar performances to those of the closet-pair algorithms. To our knowledge, these are the first algorithms of their kind.

Hardness of on-line deletions algorithms We show that the on-line-deletions closest-pair problem is at least as hard as sorting. In particular, we give a simple reduc-

tion of sorting to the on-line-deletions closest-pair problem in $\Re^1$.

Our algorithms are based on a basic on-line closest-pair algorithm presented by Schwarz and Smid [26]. This algorithm uses a dynamic data structure that is an application of Bentley's logarithmic method for decomposable searching problems [3]. We describe how to modify the on-line algorithm to obtain worst case time bounds, rather than amortized (there are other ways to obtain this), and give an efficient parallel implementation.

The rest of the paper is organized as follows. The basic on-line closest pair algorithm is given in Section 2. In Section 3 the extensions to the β-on-line algorithms (sequential and parallel) are presented. The output-sensitive algorithms with the applications for the on-line algorithm over a bounded universe and to the $\{\beta_j\}$-on-line algorithm are given in Section 4. The incremental algorithm for the off-line closest pair is given in Section 5. The algorithms for the respective ϵ-closest bichromatic pair problems are given in Section 6. The reduction of sorting to on-line-deletions closest-pair is given in Section 7. Conclusions and several open questions are given in Section 8.

2 The on-line algorithm

Our algorithm maintains a dynamically changing data structure, related to those described in [21]. A set S of n points (which were already inserted) is partitioned into $\eta+1$ subsets $S_0, S_1, \ldots S_\eta$, for $\eta = \lceil \lg n \rceil$ (some of them perhaps empty). For $i_1 > i_2$, points in S_{i_2} will be more recent than points in S_{i_1}; a new point will first be inserted into S_0 and will be "upgraded" later to S_1, then to S_2, etc. For $i = 0, 1, \ldots, \eta$ let δ_i be the closest pair distance in $S_\eta \cup S_{\eta-1} \cup \cdots \cup S_i$. Thus, δ_i will be the closest-pair distance over the first $\sum_{j=i}^{\eta} |S_j|$ points. For each non-empty subset S_i the data structure holds a *virtual mesh* of size δ_i, in $\Re^d$. For each non-empty cell in the virtual mesh of S_i we keep a list of the points in S_i which are in that cell.

The partition into subsets is defined as follows. Let $\vec{n} = (n_\eta, \ldots, n_1, n_0)$ be the binary representation of n. Then, for each $i = 0, 1, \ldots, \eta$, $|S_i| = 2^{n_i \cdot i}$. It is easy to see that indeed $|S| = \sum_{i=1}^{\eta} |S_i|$.

Inserting a new point x into S is done as follows:

0. $S \leftarrow S \cup \{x\}$.

1. Update $\delta(S)$.

2. Let i be the least significant zero in $\vec{n}$ (i.e., $S_i = \Phi$ and $S_0, S_1 \ldots, S_{i-1}$ are not empty).

3. $S_i \leftarrow S_{i-1} \cup S_{i-2} \cup \cdots \cup S_0 \cup \{x\}$.

4. $\delta_i \leftarrow \delta(S)$.

5. Compute the virtual mesh for S_i.

6. For $j = i-1, i-2, \ldots, 0$ let $S_j \leftarrow \Phi$.

7. $n \leftarrow n+1$; $\eta \leftarrow \lceil \lg n \rceil$.

The only steps in which the computation is not straightforward are steps 1 and 5. In a virtual mesh, let the neighborhood of point x be the cell containing x and the $3^d - 1$ neighboring cells (see Figure 1 for the case $d = 2$). For a mesh of size b, the following facts can be easily verified:

(1) All points whose distance from x is at most b are in the neighborhood of x.

(2) If S' is a set of points and $b \leq \delta(S')$ then the neighborhood of a point x contains at most a constant number of points from S'.

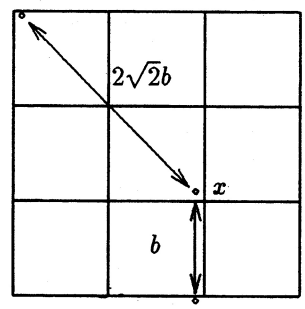

Figure 1: Neighborhood of a point x

Implementation of Step 1 The closest distance $\delta(S)$ needs to be updated only if the closest pair in $S \cup \{x\}$ is (x, y) where $y \in S_i$ for some $i = 0 \ldots, \eta$. In this case, note that y must be in the neighborhood of x in the virtual mesh of S_i (by Fact (1) above). Therefore, for each $i = 0, 1, \ldots, \eta$ we compute the distance $dist(x, y)$ for all y in the neighborhood of x in S_i, and compare this distance to $\delta(S)$. If any of these distances satisfies $dist(x, y) < \delta(S)$ then we update $\delta(S)$ to be $dist(x, S) = \min\{dist(x, y) : y \in S\}$. Since each neighborhood contains at most a constant number of points (by Fact (2) above) this step takes $O(\eta)$ time.

Implementation of Step 5 Computing the virtual mesh for S_i takes $O(|S_i|)$ steps: for each point in S_i we compute its cell and append the point to the list of its cell. We also need to enable constant access time to the list of each cell. This can be done by using a lookup table, which can be implemented using indirect addressing and unbounded memory. A linear-space hash table with constant lookup time can be computed in linear time, with high probability [11, 8]. For our algorithm we will need to use dynamic-hashing, in which keys are dynamically inserted into and deleted from the hash table. We will use a "real-time" dictionary algorithm which, using linear space at all time, supports each insertion, deletion, or lookup in constant time, with high probability [9, 8]. In

the rest of the paper we will not explicitly mention the use of the dictionary algorithm.

Complexity Computing a virtual mesh in Step 5 can take $O(n)$ time in the worst case. However, it only takes $O(\lg n)$ amortized time per insertion: Note that when m points are moved into a new subset it takes $O(m)$ time to compute a virtual mesh for the new subset; i.e., it takes $O(1)$ amortized time per point. For a set of size n, a point could have only moved into $\eta + 1$ new subsets and therefore the amortized insertion time per point is $O(\eta) = O(\lg n)$.

From amortized to worst case time Instead of stalling the algorithm until the virtual mesh for the updated subset S_i is computed, continue to insert points while computing the virtual mesh in the "background." Specifically, we start a background process in which points from $\{x\} \cup S_0 \cup S_1 \cup \cdots \cup S_{i-1}$ are inserted into the set S_i. First x is inserted, then the points from S_0, S_1, etc. The meaning of "background process" is that for each operation of the regular insertion process, there are a constant number of operations for the background process. Note that in the regular insertion process, we may start more background processes: after another $2^{i'}$ points are inserted, they all need to be moved from sets $S_0, S_1, \ldots, S_{i'-1}$ into a new set $S_{i'}$. However, by an appropriate slowdown of the insertion process in favor of the background process we guarantee that by the time $2^{i'} - 1$ new points are inserted into $S_0, S_1, \ldots, S_{i'-1}$, all the previous points of $S_{i'}$ are already moved into S_i; this holds for $i' = 0, 1, \ldots, i-1$. Therefore, we never have interference between background computations, and the number of sets in the data structure is always $\eta + 1$. Moreover, since in the insertion process each query takes $O(\lg n)$ time, we can afford running up to $\lg n$ background processes, which is in any case an upper bound for the number of such processes in the data structure.

Parallel implementation Step 1 can be implemented easily in parallel: Using $\lg n$ processors, the $O(\lg n)$ distances can be computed in constant time in parallel, and their minimum can be computed in constant time with high probability [24]. In Step 5, we need to append points of S_i into their lists in parallel. We know that each list will contain at most a constant number of points, and therefore this can be implemented using $|S_i|$ processors in constant time, using indirect addressing into unbounded memory. Linear-space implementation can be obtained by using parallel hashing; a linear-space hash table for the lists with constant lookup time can be computed in $O(\lg^* n)$ time and $O(|S_i|)$ operations with high probability [20, 13].

We have

Theorem 1 *In the on-line closest pair algorithm, each point can be inserted into a set of size n in $O(\lg n)$ time in the worst case; it can also be implemented using $O(n)$ space and $O(\lg n)$ time per insertion with high probability. Using $\lg n / \lg^* n$ processors on a CRCW PRAM, each insertion can be processed in $O(\lg^* n)$ time with high probability (optimal speedup).*

3 The β-on-line algorithm

If points are inserted in n/β batches of size β each then we modify the on-line algorithm as follows. Let $n' = n/\beta$ be the number of batches already inserted into the set S and X be a new batch to be inserted. Let $\vec{n}' = (n'_{\eta'}, \ldots, n'_1, n'_0)$ be the binary representation of n', where $\eta' = \lceil \lg n' \rceil$. The set S is partitioned into subsets $S_0, S_1, \ldots, S_{\eta'}$ such that, for all $i = 0, 1, \ldots, \eta'$, $|S_i| = \beta \cdot 2^{n'_i \cdot i}$. It is easy to see that indeed $|S| = \sum_{i=1}^{\eta'} |S_i|$.

Inserting a new batch X of β points into S is similar to the algorithm in Section 2:

0. $S \leftarrow S \cup X$.

1. Update $\delta(S)$.

2. Let i be the least significant zero in $\vec{n}'$ (i.e., $S_i = \Phi$ and $S_0, S_1 \ldots, S_{i-1}$ are not empty).

3. $S_i \leftarrow S_{i-1} \cup S_{i-2} \cup \cdots \cup S_0 \cup X$.

4. $\delta_i \leftarrow \delta(S)$.

5. Compute the virtual mesh for S_i.

6. For $j = i-1, i-2, \ldots, 0$ let $S_j \leftarrow \Phi$.

7. $n' \leftarrow n' + 1$; $\eta' \leftarrow \lceil \lg n' \rceil$.

The implementation is the same as in the on-line algorithm, except for Step 1, which is computed as follows:

$$\delta(S) \leftarrow \min\{\delta(S), \delta(X), dist(X, S)\} \ ,$$

where $dist(X, S) = \min\{dist(x, y) : x \in X, y \in S\}$. The distance $dist(X, S)$ can be computed in $O(\beta)$ time, by computing for each point $x \in X$ its distance to S, $dist(x, S) = \min\{dist(x, y) : y \in S\}$, as in the on-line algorithm. The distance $\delta(X)$ can be computed in $O(\beta)$ time with high probability, using an off-line closest pair algorithm [23, 17].

The β-on-line problem is also natural in the parallel context, in which at each parallel step, β new points are inserted into the set. Indeed, all steps in the β-on-line algorithm above can be done in parallel in constant time in a straightforward manner, except for the computation of $\delta(X)$. This computation can be done in constant time with high probability, using β processors and unbounded space [18]. Parallel computation in $O(n)$ space can be done by using a parallel hashing algorithm which takes $O(\lg^* \beta)$ time and $O(\beta)$ operations with high probability [20, 13]. We will also need a parallel dictionary algorithm which supports parallel insertions into and deletions from the hash table. Insertions and deletions of β

keys can be implemented in $O(\lg^* \beta)$ time and $O(\beta)$ operations, with high probability [13]. The parallel implementation of queries and insertions are as in the on-line algorithm.

We therefore have,

Theorem 2 *In the β-on-line closest pair algorithm, each batch of β points is inserted into a set of size n in $O(\beta \lg(n/\beta))$ time with high probability, using $O(n)$ space; it can also be inserted in $O(\lg^* n)$ time with high probability, using $\beta \lg(n/\beta)/\lg^* n$ processors and $O(n)$ space.*

4 An output-sensitive algorithm

The β-on-line problem generalizes both the off-line closest pair problem (with $\beta = n$) and the on-line closest pair problem (with $\beta = 1$). The β-on-line problem demonstrates that performance of an algorithm may depend primarily on the number of times $\delta(S)$ actually changes. We show that in fact it may depend on the exact locations (in the sequence of input points) where these changes occur. This fact is shown below by modifying our algorithm to be output-sensitive.

4.1 A basic algorithm

Denote the set of points between the $(j-1)^{\text{st}}$ time and the jth time that $\delta(S)$ changes as the jth *batch*. Let β_j be the number of points in the jth batch. Each subset S_i will now be a set of batches rather than a set of points; a non-empty set S_i contains 2^i batches, but perhaps many more points. Let n' be the number of batches. Then, the number of sets S_i is $\eta' + 1 = \lg n' + 1$. In addition to the subsets $S_0, S_1, \ldots, S_{\eta'}$, we keep a set X that contains all the last points x that did not change $\delta(S)$, and a virtual mesh of size $\delta(S)$ for X. Thus, S is partitioned into $S = X \cup S_0 \cup S_1 \cup \ldots \cup S_{\eta'}$. Given a new point x, we compute $dist(x, S)$ as before; if $dist(x, S) \geq \delta(S)$ then x is inserted into X and appended to the list of x's cell in the virtual mesh of X. If $dist(x, S) < \delta(S)$ then we update $\delta(S)$ to be $dist(x, S)$ and move x and the points of X to be in $S_0 \cup S_1 \cup \ldots \cup S_{\eta'}$. The insertion is the same as in the on-line algorithm, except that n is replaced by n' and x is replaced by X. We get

Theorem 3 *In the output-sensitive on-line algorithm each point is inserted in $O(\lg n')$ time in the worst case; it can also be implemented using $O(n)$ space and $O(\lg n')$ time per insertion with high probability.*

4.2 A more sensitive algorithm

The algorithm given above is sensitive to the number of times $\delta(S)$ changes. However, it is not sensitive to exactly when these changes occur. In particular, large batches and small batches are treated equally. To see the inefficiency of such a treatment, consider an input sequence consisting of $\sqrt{n}$ batches of size 1, followed by a large batch of size $n - 2\sqrt{n}$, followed by $\sqrt{n}$ batches of size 1. The algorithm above will take $O(\lg n)$ time per insertion, while $O(1)$ amortized time can be easily achieved by keeping the large batch in a separate data structure. The output-sensitive algorithm below takes into account the sizes of batches. The insertion of a new point x to S is done as follows:

0. $S \leftarrow S \cup \{x\}$.

1. Update $\delta(S)$.

2. Let $i_x = \lceil \lg |X| \rceil$ and
$i = \min\{i' : S'_i = \Phi \text{ and } i' \geq i_x\}$.

3. $S_i \leftarrow S_{i-1} \cup S_{i-2} \cup \cdots \cup S_{i_x} \cup X$.

4. $\delta_i \leftarrow \delta(S)$.

5. Compute the virtual mesh for S_i.

6. For all j, $i > j \geq i_x$, let $S_j \leftarrow \Phi$.

7. $n' \leftarrow n' + 1$; $\eta' \leftarrow \lceil \lg n' \rceil$.

By the amortization analysis in Section 2, the amortized insertion cost for each point in X is $O(\lg n - \lg |X|)$; intuitively, each point has "jumped" directly into the i_x'th level, thus saving i_x "upgrades." To improve on the time, however, we also need to save in the query time of Step 1; namely, we need to reduce the number of sets in the partition of S. To accomplish that, along with inserting points into a set X, we process a "background" computation in which points from the sets $S_0, S_1, \ldots$ are inserted into a combined set Y. After X is inserted, as above, set Y is inserted as well in a similar way. By having a constant slowdown in the insertion process in favor of the background computation of Y, we can get $|Y| \geq |X|$, meaning that the number of non-empty sets in the partition becomes at most $\lg n - \lg |X|$, as required.

The query time and amortized insertion time for a point in the j'th batch are therefore $O(\lg n - \lg \beta_{j-1})$ and $O(\lg n - \lg \beta_j)$, respectively. (Note that the first bound might be quite pessimistic.) Therefore, the query time and amortized insertion time per point are $O\left(\frac{1}{n}\sum_{j=1}^{n'} \beta_j (\lg n - \lg \beta_{j-1})\right)$ and $O\left(\frac{1}{n}\sum_{j=1}^{n'} \beta_j (\lg n - \lg \beta_j)\right)$, respectively. Using similar techniques to those described in Section 2, the insertion time can be made worst case, rather than amortized. Let $\mathcal{F}(\{\beta_j\}) = \frac{1}{n}\sum_{j=1}^{n'} \beta_j \left((\lg n - \lg \beta_{j-1}) + (\lg n - \lg \beta_j)\right)$. We get

Theorem 4 *For batches of sizes $\beta_1, \beta_2, \ldots, \beta_{n'}$ defined as above, the output-sensitive on-line algorithm takes $O(\mathcal{F}(\{\beta_j\}))$ time per insertion in the worst case; it can also be implemented using $O(n)$ space and the same time, with high probability.*

4.3 Further improvements

There are two more improvements that can be obtained for certain sequences, both regarding the updating criteria:

Relaxation: The definition of "a change in $\delta(S)$" is relaxed to "a change in $\delta(S)$ by a factor of c," where $c > 1$. This results in a slowdown of computing $dist(x, S)$ by at most a factor of c^d, but may result with more favorable sequences of $\beta_1, \beta_2, \ldots, \beta_{n'}$. In fact, this relaxation is used below to obtain better performance in the worst case for input taken from a bounded universe.

Adaptation: When $\delta(S)$ changes, we essentially want to update the virtual mesh X because the number of points in a cell may become large, implying a large query time per point. However, the actual performance may be quite different; it may well be the case that even though $\delta(S)$ changes, the query time of a new point into the virtual mesh of X remains small. We adapt the updating strategy to the actual performance in run time:

Let $\bar{x}$ be the future size of X when X will be inserted into the data structure; let k be the number of sets in the data structure; let q be the number of queries of points within X. Our objective is to delay the insertion of X into the data structure as much as possible, i.e., to get as large $\bar{x}$ as possible. We have to make sure however that q does not become too large. The idea is to amortize q against the other necessary costs. We know that for points in X the query time in other sets is $O(k)$ per point and the future amortized insertion time is $O(\lg n - \lg \bar{x})$ per point. Accordingly, we allow $q = O(\bar{x}(k + \lg n - \lg \bar{x}))$.

4.4 A $\{\beta_j\}$-on-line algorithm

We note that for $\beta_1 = \beta_2 = \cdots = \beta_{n'} = \beta$ we get the same complexity as for the β-on-line problem: $O(\lg n - \lg \beta)$ per insertion. Also, if the points are given in batches of sizes $\beta_1, \beta_2, \ldots$ then we can get a similar complexity to that of Theorem 4 by using techniques from both the above output-sensitive algorithm and from the β-on-line algorithm of Section 3. Note that if the points of a batch are inserted one at the time, then $\delta(S)$ may change several times, unlike the case in the output-sensitive algorithm, where this cannot happen by definition. We thus extend the β-on-line algorithm to the more general case.

Theorem 5 *The on-line problem in which points are given in batches of sizes $\beta_1, \beta_2, \ldots, \beta_{n'}$ can be solved in $O(\min\{\lg n', \mathcal{F}(\{\beta_j\})\})$ time per insertion in the worst case with high probability, using $O(n)$ space.*

4.5 An on-line algorithm in a bounded universe

Assume that the points are given from a bounded universe U^d, where $U = [1, 2, \ldots, u]$ (as is often the case in practice). We use the *relaxation* rule from Section 4.3: $\delta(S)$ must change by a constant factor before any update action is taken. Since for distinct points we have $1 \leq \delta(S) \leq \sqrt{d}u$, the number of updates can be at most $n' = O(\lg u + \lg d)$. Therefore, by Theorem 3 we have

Theorem 6 *The on-line closest-pair problem with points taken from U^d can be solved such that each insertion takes $O(\lg\lg(u + d))$ time; it can also be implemented using $O(n)$ space and $O(\lg\lg(u + d))$ time per insertion with high probability.*

5 An incremental off-line algorithm

Given a set S of n points, we compute the closest pair for S by the following incremental algorithm:

1. Randomly permute the elements in S.
2. Insert the points according to the random permutation and apply the output-sensitive algorithm of Theorem 4.

A random permutation can be computed easily in linear expected time. We show below that the performance of the output-sensitive on-line algorithm for a random permutation of any given set S is linear expected time as well.

Lemma 7 *Given a subset S' of k points selected at random from the set S and its closest distance $\delta(S')$, the expected number of points that will be inserted into S' before $\delta(S')$ is changed is $\Omega(k)$.*

Proof. Omitted. ∎

As a result, we get $\mathbf{E}(\beta_j) = \Omega(\beta_1 + \beta_2 + \cdots + \beta_{j-1})$ and as can be easily shown by induction $\mathbf{E}(\beta_j) \geq c2^j$, for some constant $c > 0$. By convexity arguments we get $\sum_j \beta_j(2\lg n - \lg \beta_{j-1} - \lg \beta_j) \leq \sum_j c2^j(2\lg n - 2\lg c - j + 1 - j) \leq \sum_j c'n2^{j-\lg n}(\lg n - j)$, for some constant $c' > 0$, and by changing j to $(\lg n - j)$ we get $\sum_j \beta_j(2\lg n - \lg \beta_{j-1} - \lg \beta_j) \leq \sum_j c'n2^{-j}(j) = O(n)$.

We therefore have

Theorem 8 *The incremental algorithm solves the off-line closest pair in $O(n)$ expected time and $O(n)$ space.*

6 Algorithms for ϵ-closest bichromatic pair problems

The algorithms are similar to the semi-dynamic closest pair algorithms, and have similar performances. They

incorporate modifications that are based on the off-line ϵ-closest bichromatic pair algorithm of [17].

Due to space limitations, we only describe the facts that are the basis for the modifications. Let $\delta(S)$ be the distance of the bichromatic closest pair in a set S. In a mesh of size $\delta(S)$ each cell contains points of only a constant number of different colors. However, a cell may contain many points of the same color. As in [17], consider a *refined virtual mesh* of size $\epsilon\delta(S)/9$, and for all points of the same color in a non-empty cell select one (arbitrarily) as a *representative*. It follows that the closest bichromatic pair over the representatives is of distance which is larger than $\delta(S)$ by at most a factor of $(1+\epsilon)$ (see [17] for details). For a constant $\epsilon > 0$, the number of representatives in each neighborhood (in the mesh of size $\delta(S)$) is constant (specifically, it is $O(1/\epsilon^d)$).

7 On-line deletions is as hard as sorting

We give a simple reduction from *sorting* to on-line-deletions closest-pair computations in $\Re^1$. (The reduction is a "Karp-reduction" in the sense that the on-line-deletions closest-pair algorithm is used as a subroutine in the sorting algorithm.) Let S be a set of input points to be sorted. An algorithm for the on-line-deletions closest-pair problem will be used to sort S. Our sorting algorithm will consist of $n-1$ iterations; at each iteration a closest pair computation will be done, a constant number of data processing steps will follow, and one point will be deleted from the set.

At each iteration, the closest-pair computation will enable to compute the successor of one of the input points. The main idea is that these two points can be represented by only one of them for further processing. More generally, at each iteration each point represents a subset of consecutive points from the output (sorted) set that were merged into the subset in previous iterations. The ordering between points in each subset is already known. Each closest-pair computation results with merging the two sets represented by the points of the closest pair.

More formally, let $S_0 = S$ and for $i \geq 1$ let S_i be the set of points after the ith deletion. As an invariant the algorithm will keep the following properties:

P1. Each point $x \in S_i$ represents a subset S^x of points in S, and each point of S is represented by exactly one point in S_i. Also, $x \in S^x$.

P2. Let $x' = \min\{S^x\}$ and $x'' = \max\{S^x\}$. Then, $S^x = \{y \in S : y \in [x', x'']\}$; i.e., the partition of S into $\cup_{x \in S_i} S^x$ is an ordered partition.

P3. For each point in $S^x \setminus \{x''\}$ the successor in S is already known; similarly, for each point in $S^x \setminus \{x'\}$ the predecessor in S is already known.

It is clear from the above that if the invariant is kept then after the $(n-1)$'st deletion we know for each element in S its successor and its predecessor and we have a sorted list. It remains to show how the algorithm with this invariant proceeds. The i'th iteration of the algorithm is as follows.

Iteration i:

1. Compute the closest pair (p, q) in S_{i-1} (assume, without loss of generality, that $p < q$).
2. Complete successor/predecessor information for p'' and for q': let $succ[p''] \leftarrow q'$; $pred[q'] \leftarrow p''$.
4. Merge S^p and S^q into S^p: let $S^p \leftarrow S^p \cup S^q$.
5. Delete q: let $S_i \leftarrow S_{i-1} \setminus \{q\}$.
3. Update p'': let $p'' \leftarrow q''$.

It is straightforward to show by induction that the invariant properties hold. We have

Theorem 9 *The problem of sorting n numbers can be reduced in linear time to the on-line-deletions closest-pair problem for a set of size n from $\Re^1$.*

8 Conclusions

We presented a simple and efficient algorithm for the batch-on-line closest pair problem whose complexity depends on the sizes of batches; in the extreme cases of on-line and off-line problems (batches of size one in the former and a batch of size n in the latter) the algorithm's performance matches the best known algorithms for these problems. We also presented an output-sensitive algorithm; motivated by the fact that as long as the closest pair distance does not change, the situation should be no worse than in the off-line algorithm, the output-sensitive algorithm adapts itself to be more efficient on "good" input sequences. Its usefulness is demonstrated by two applications: an efficient on-line algorithm for points in a bounded universe: $O(\lg\lg u)$ time per insertion for a universe of size u, and a linear expected time incremental algorithm for the off-line problem. Can this approach be useful for getting a more efficient on-line algorithm in the worst case?

We have considered semi-dynamic algorithms for the ϵ-closest bichromatic pair problem and gave efficient algorithms, similar to the algorithms given for the closest pair problems. The problem of finding efficient algorithms for the exact (off-line or on-line) bichromatic closest pair problems remains open.

We showed that the on-line-deletions closest-pair problem is at least as hard as sorting, even for points taken from $\Re^1$. While this is not new for the algebraic

model, this has implications for the non-algebraic models. Note that for the non-algebraic model, sorting can be computed in $O(n\sqrt{\lg n})$ expected time using the Fusion Tree data structure of Fredman and Willard [12]. An open question on the positive side is: Can Fusion Trees help to get a faster on-line algorithm with deletions?

Acknowledgments

I would like to thank Samir Khuller and Dave Mount for helpful discussions. Their comments helped to motivate the application of output-sensitive algorithm for an off-line closest pair algorithm in Section 5, and the ϵ-closest bichromatic pair algorithms in Section 6. I would like to thank Samir also for his valuable comments regarding an earlier version of this paper.

References

[1] P. K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, and E. Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. In *ACM-SCG'90*, pages 203–210, 1990.

[2] M. Ben-Or. Lower bounds for algebraic computation trees. In *STOC '83*, pages 80–86, 1983.

[3] J. L. Bentley. Decomposable searching problems. *Inf. Process. Lett.*, 8:244–251, 1979.

[4] J. L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23:214–229, 1980.

[5] J. L. Bentley and M. I. Shamos. Divide and conquer in multidimensional space. In *STOC '76*, pages 220–230, 1976.

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw Hill and The MIT Press, 1990.

[7] M. T. Dickerson and R. S. Drysdale. Enumerating k distances for n points in the plane. In *ACM-SCG'91*, pages 234–238, 1991.

[8] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *ICALP '92*, pages 235–246, July 1992.

[9] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *ICALP '90*, pages 6–19, 1990.

[10] S. Fortune and J. E. Hopcroft. A note on rabin's nearest-neighbor algorithm. *Inf. Process. Lett.*, 8(1):20–23, 1979.

[11] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, July 1984.

[12] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *STOC '90*, pages 1–7, 1990.

[13] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *FOCS '91*, pages 698–710, Oct. 1991.

[14] M. Golin, R. Raman, C. Schwarz, and M. Smid. Randomized data structures for the dynamic closest-pair problem. In *SODA '93*, Jan. 1993.

[15] M. Golin, R. Raman, C. Schwarz, and M. Smid. Simple randomized algorithms for closest pair problems. In *CCCG '93*, Aug. 1993.

[16] K. Hinrichs, J. Nievergelt, and P. Schorn. Plane-sweep solves the closest pair problem elegantly. *Inf. Process. Lett.*, 26:255–261, 1988.

[17] S. Khuller and Y. Matias. A simple randomized sieve algorithm for the closest-pair problem. In *CCCG '91*, pages 130–134, 1991. To appear in *information and computation*.

[18] P. D. MacKenzie and Q. F. Stout. Ultra-fast expected time parallel algorithms. In *SODA '91*, pages 414–423, 1991.

[19] U. Manber. *Introduction To Algorithms, A Creative Approach*. Addison-Wesley, 1989.

[20] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *STOC '91*, pages 307–316, May 1991.

[21] K. Mehlhorn. *Multi-dimensional Searching and Computational Geometry*. Springer-Verlag, Berlin Heidelberg, 1984.

[22] F. P. Preparata and M. I. Shamos. *Computational Geometry; An Introduction*. Springer-Verlag, 1985.

[23] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, 1976.

[24] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM J. Comput.*, 14(2):396–409, May 1985.

[25] J. S. Salowe. Shallow interdistance selection and interdistance enumeration. *International Journal of Computational Geometry & Applications*, 2:49–59, 1992.

[26] C. Schwarz and M. Smid. An $O(n \lg n \lg \lg n)$ algorithm for the on-line closest pair problem. In *SODA '92*, pages 280–285, 1992.

[27] C. Schwarz, M. Smid, and J. Snoeyink. An optimal algorithm for the on-line closest-pair problem. In *ACM-SCG'92*, pages 330–336, 1992.

[28] M. I. Shamos and D. Hoey. Closest-point problems. In *FOCS '75*, pages 151–162, 1975.

[29] M. Smid. Dynamic rectangular point location, with an application to the closest pair problem. In *Proc. 2nd Annual International Symp. on Algorithms*, 1991.

[30] M. Smid. Maintaining the minimal distance of a point set in less than linear time. *Algorithms Review*, 2:33–44, 1991.

[31] M. Smid. Maintaining the minimal distance of a point set in polylogarithmic time. *Discrete Comput. Geom.*, pages 415–431, 1992.

[32] K. J. Supowit. New techniques for some dynamic closest-point and farthest-point problems. In *SODA '90*, pages 84–90, 1990.

[33] A. C. Yao. Lower bounds for algebraic computation trees with integer inputs. *SIAM J. Comput.*, 20(4):655–668, 1991.