Dynamic data structures for approximate Hausdorff distance in the word RAM

Timothy M. Chan* Dimitrios Skrepetos†

Abstract

We give a fully dynamic data structure for maintaining an approximation of the Hausdorff distance between two point sets in a constant dimension d, a standard problem in computational geometry. Our solution has an approximation factor of $1+\varepsilon$ for any constant $\varepsilon > 0$ and expected update time $O(\frac{\log U}{\log\log n})$, where U is the universe size, and n is the number of the points. The result of the paper greatly improves over the previous exact method, which required $O(n^{5/6}\text{polylog}\, n)$ time and worked only in a semi-online setting. The model of computation is the word RAM model.

1 Introduction

The problem of computing the Hausdorff distance between a red point set R and a blue point set B is to find $\max_{b\in B}\min_{r\in R} d(b,r)$ where $d(\cdot,\cdot)$ is a distance function. The dynamic version of the Hausdorff distance problem is to solve the problem under a series of insertions and deletions of points both from the red point set and from the blue point set. The approximate dynamic version of the Hausdorff distance is to solve the dynamic version of the problem allowing a factor of approximation.

Chan [4] provided a solution for the semi-online maintenance of Hausdorff distance in 2-d that had an update time of $O(n^{5/6}\text{polylog}\, n)$ in the worst case. Here *semi-online* means that when we insert an element we are given its deletion time in advance. Chan also gave a fully dynamic solution for a decision version of the problem in 2-d that had an amortized update time of $O(n^{1/2}\text{polylog}\, n)$.

In this paper we show that the update time can be greatly improved if we allow approximation. We give a dynamic data structure that maintains the Hausdorff distance within an approximation factor of $1+\varepsilon$ in *sublogarithmic* update time—or more concretely, $O(\frac{\log U}{\log\log n})$ time.

Note that our update time of $O(\frac{\log U}{\log\log n})$ greatly improves over the previous bound of $O(n^{5/6}\text{polylog}\, n)$ for the semi-online maintenance of the exact Hausdorff distance, as it is purely polylogarithmic and fully dynamic. Furthermore, our solution works for any constant dimension d, while the previous method works only in 2-d.

We originally started this research for a closely related problem, the dynamic bichromatic closest pair problem, studied by Eppstein [3], which involves finding $\min_{b\in B}\min_{r\in R} d(b,r)$. However, this seems easier to approximate than the Hausdorff distance, as one could obtain a constant-factor approximation with update time of $O(\log\log U)$ by extending the technique of Chan [1].

1.1 Model of computation

We assume the word RAM model of computation; that is, the point coordinates are drawn from the set $[U] = \{0,1,\ldots,U-1\}$, the words are composed of $\Theta(\log U)$ bits, and each standard word operation (such as shifting, arithmetic and logical operations) takes constant time. The distance function is assumed to be the Euclidean distance function. The approximation factor ε and the dimension d are considered to be constant in both the running time and space of our algorithms.

2 Approximate dynamic Hausdorff distance

In section 2.1 we give a dynamic data structure that solves the approximate decision Hausdorff distance problem. That is, given a distance r and an approximation factor ε fixed at preprocessing time, we want to answer decision queries of the form: "Is the Hausdorff distance greater than r?". The data structure should return "yes" if the Hausdorff distance is greater than $r(1+\varepsilon)$ and "no" if it is less than r. In section 2.2 a stronger version of the decision problem is considered: instead of a fixed r, the data structure can answer decision queries for all $r \in \{2^i \mid 0 \le i \le \log U\}$ at the same time. Finally, in section 2.3 we solve the original version of the approximate Hausdorff distance problem; that is, we compute a $(1+\varepsilon)$-approximation of the Hausdorff distance.

*Cheriton School of Computer Science, University of Waterloo, tmchan@uwaterloo.ca

†Cheriton School of Computer Science, University of Waterloo, dskrepet@uwaterloo.ca

2.1 Decision queries for a fixed distance r

The first step is to compute a grid G_s of the d-dimensional space composed of cells that are hypercubes with $s = r\varepsilon$ side length. For a cell c, another cell c' is a neighbor of c if the minimum distance between the centers of c and c' is less than or equal to r. We imagine that all the points that fall inside a cell are *rounded* to the center of the cell. Let $neighbors_c$ denote the set of the neighbors of the cell c that is produced by this definition. It is clear that the number of neighbors of a cell is dependent only on ε and d, so it will be treated as a constant in the rest of the paper. Because of rounding the approximation factor is $1 + O(\varepsilon)$.

Each cell c of the grid has a unique ID; thus, for each point $p = (x_1, x_2, \ldots, x_d) \in c$ the ID of the cell is $id(c) = (\lfloor \frac{x_1}{s} \rfloor, \lfloor \frac{x_2}{s} \rfloor, \ldots, \lfloor \frac{x_d}{s} \rfloor)$. Each cell c also has three counters, one for the number of blue points inside the cell, $blueCount_c$, one for the number of red points inside the cell, $redCount_c$, and one for the number of red points in any cell c' that belongs to the set $neighbors_c$, $redNeighborsCount_c$. Furthermore, each cell c maintains a boolean flag, $flag_c$, with the following semantic: $flag_c$ is set if and only if $blueCount_c > 0$ and there is no neighbor cell c' such that $redCount_{c'} > 0$, which is the same with checking whether $redNeighborsCount_c > 0$. If there is at least one such flag set, then the Hausdorff distance is greater than r (after rounding). Otherwise, it is at most r. We keep a global variable *globalCount* that stores the number of cells with flags that have been set. With this variable, we can answer decision queries in constant time.

In order to keep the total space linear in the number of points in the red and blue point sets, we use a hash table to store only the $O(n)$ cells that are nonempty (where n is the number of points of both the point sets), since in the worst case each point occupies a different cell. We use the ID of the cell as key. The hash table supports insertions, deletions, and searching in expected constant time.

To insert a point p, we compute the ID of the cell c that contains the point (that is, $id(c) = (\lfloor \frac{x_1}{s} \rfloor, \lfloor \frac{x_2}{s} \rfloor, \ldots, \lfloor \frac{x_d}{s} \rfloor)$). If the cell does not exist in the hash table, a new entry for the cell is created and inserted to the hash table with $redCount_c$ and $blueCount_c$ equal to zero, $redNeighborsCount_c$ equal to the accumulation of all the $redCount_{c'}$ values for every cell c' in the set $neighbors_c$, and $flag_c$ equal to zero. Assume that the inserted point is red. Then, the $redCount_c$ value is incremented. Afterwards, for each cell c' in the set $neighbors_c$ we increment the value $redNeighborsCount_{c'}$, and if $blueCount_{c'} > 0$ and $redNeighborsCount_{c'}$ was zero prior to the insertion, then the $flag_{c'}$ is reset, and the *globalCount* is decremented. Now, assume that the inserted point is blue. Then the $blueCount_c$ value is incremented. Afterwards, if the $blueCount_c$ value was zero prior to the insertion and $redNeighborsCount_c > 0$, the $flag_c$ is set, and *globalCount* is incremented. The pseudocode of the algorithm for inserting points is given in Algorithm 1.

```
1  Find the cell c that contains the point p
2  if c does not exist in the hash table then
3  |  create it and insert to the hash table
4  |  set blueCount_c, redCount_c,
   |  redNeighborsCount_c, flag_c to 0
5  |  for all c' ∈ neighbors_c do
6  |  |  redNeighborsCount_c += redCount_c'
7  if p is red then
8  |  increment redCount_c
9  |  for all c' ∈ neighbors_c do
10 |  |  increment redNeighborsCount_c'
11 |  |  if blueCount_c' == 1 then
12 |  |  |  set flag_c'
13 |  |  |  increment globalCount
14 else
15 |  increment blueCount_c
16 |  if blueCount_c == 1 then
17 |  |  if redNeighborsCount_c > 0 then
18 |  |  |  set flag_c increment globalCount
```

Algorithm 1: Insert-Point

To delete a point p, the ID of the cell c that contains the point is computed $(id(c) = (\lfloor \frac{x_1}{s} \rfloor, \lfloor \frac{x_2}{s} \rfloor, \ldots, \lfloor \frac{x_d}{s} \rfloor))$, and $redCount_c$ or $blueCount_c$ is accordingly decremented. If the point is red, for every c' in the set $neighbors_c$ we decrement the $redNeighborsCount_{c'}$ value, and if this value becomes zero, the $flag_{c'}$ is reset and the *globalCount* is decremented. If the point is blue and the cell c does not have any other blue points after the deletion, the $flag_c$ is reset and the *globalCount* is decremented. Finally, if the cell does contain any points (either red or blue), it is deleted from the hash table. The pseudocode of the algorithm for deleting points is given in Algorithm 2.

In order to answer a distance query, we check the value *globalCount*, and we return "yes" if it is zero and "no" otherwise.

The aforementioned data structure can handle insertions and deletions of points in constant time. The queries can be answered in constant time as well. Consequently, this data structure with a linear space and preprocessing time can handle point insertions and deletions and can answer queries for a fixed distance r and a fixed ε in constant time.

2.2 Decision queries for all distances $r \in \{2^i \mid 0 \leq i \leq \log U\}$

The naive way to perform the query for all distances $r \in 1, 2, 4, ..., U$ would be to create $O(\log U)$ instances

```
1  Find the cell c that contains the point p
2  if p is red then
3  |  decrement redCount_c
4  |  for all c' ∈ neighbors_c do
5  |  |  decrement redNeighborsCount_c'
6  |  |  if redNeighborsCount_c' == 0 then
7  |  |  |  reset flag_c'
8  |  |  |  decrement globalCount
9  else
10 |  decrement blueCount_c
11 |  if blueCount_c == 0 then
12 |  |  reset flag_c
13 |  |  decrement globalCount
14 if (redCount_c == blueCount_c == 0) then
15 |  delete cell c from the hash table
```

Algorithm 2: Delete-Point

of the above data structure, but this would increase the required space to $O(n \log U)$ words and the required update time to $O(\log U)$. In order to reduce these bounds, we use standard word RAM techniques, more specifically word packing tricks.

2.2.1 Interpreting the grids as a quadtree

In order to handle multiple distance values of r at the same time, we imagine the space decomposition that is implicitly imposed by the grids of the multiple side lengths as a d-dimensional complete quadtree with all the leaves at the same depth (although such a tree is not explicitly maintained) for the hypercube of side length U and with height $O(\log U)$. Such a quadtree creates hypercubes of side length r with $r \in \{2^i \mid 0 \leq i \leq \log U\}$, but we need hypercubes of side length $r\varepsilon$ as explained in section 2.1. Therefore, in order to adjust the standard quadtree decomposition of the space, we have to create $\frac{1}{\varepsilon^d}$ children to each of the leaves of the tree. Notice that all the nodes of the tree at depth i correspond to the grid G_s for $s = \frac{U}{2^i}\varepsilon$, and each node of the tree at depth i corresponds to a cell of that grid.

The $flag_c$ value of a cell c of a grid G_s has the same meaning as in section 2.1, but it is not explicitly maintained. The single register *globalCount* is replaced by the array *globalCountArray* that has $O(\log U)$ size, and each value at index i corresponds to the number of the cells c with nonzero $flag_c$ for the grid G_s with $s = \frac{U}{2^i}\varepsilon$. The goal of the update process is to find the array *changeArray* that stores the changes that need to be done to *globalCountArray*.

Two flags are related to each node of the tree, $emptyRed_c$ and $emptyBlue_c$. The first flag is set to 1 if there are no red points in the cell c of the grid G_s with $s = \frac{U}{2^i}\varepsilon$ that corresponds to that node, and 0 otherwise. The second flag is similarly defined for the blue points.

2.2.2 Applying word RAM techniques

The key idea in achieving sublogarithmic time is to perform the operations for multiple grids in constant time by taking advantage of the tools that word RAM provides us with; that is, we have $O(\log U)$ grids, but we need to compress them. Starting at the root of the quadtree, we store all the $emptyRed_c$ flags of the nodes of the quadtree that are in the subtree s of the root of depth b, where b is a value to be determined later, in a RAM word ($emptyRed_s$), and we store all the $emptyBlue_c$ flags of that subtree in another word ($emptyBlue_s$). The same is done for each subtree of depth b of every node at depth $ib+1$ for $0 \leq i \leq \frac{\log U}{b} - 1$. We note that only the words of the subtrees that are not entirely composed of zeros need to be stored, and we store them in a hash table using the ID of the cell that corresponds to the root of the subtree and the depth of the subtree as key. The $redCount_c$ and the $blueCount_c$ of the nonempty cells c of the most detailed grid G_s with $s = \varepsilon$ are stored in a separate word in another hash table using their ids as keys.

2.2.3 Maintaining the flags

When a point is inserted (deleted), we find the leaf cell c that corresponds to the grid G_s with $s = \varepsilon$, and we increment (decrement) $redCount_c$ or $blueCount_c$ accordingly. We create or delete subtrees from the hash table when needed. If the number of the points of the same color with the inserted point is nonzero (zero for deleted point), there is nothing that needs to be done. Otherwise, we start processing the subtrees that lie in the path from c to the root of the quadtree in a bottom-up manner as follows.

If the inserted (deleted) point is red, for each subtree s on the path we update the $emptyRed_s$ word. To find the part of the *changeArray* that corresponds to s, we need to count the changes of the $flag_{c'}$ values of the cells $c' \in neighbors_c$ for each cell c in the path from the leaf of s in which the insertion (deletion) took place to the root of s, as only these cells are affected from the insertion (deletion). First we fetch the $emptyBlue_{s'}$ words of the subtrees s' that contain neighbors $c'_{root} \in neighbors_{c_{root}}$ of the root cell c_{root} of s (it is easy to see that these words also contain the neighbors for all cells c described above), and then we fetch the $emptyRed_{s''}$ words of subtrees s'' that contain the neighbors $c''_{root} \in neighbors_{c'_{root}}$ for all the root cells c'_{root} of the subtrees s' (similarly, these words also contain the neighbors for all cells in the subtree s'). With these words we can find the cells c' of each subtree s' whose $flag_{c'}$ needs to be set (reset) since we have access to all the neighbors of these cells. That can be done by checking for each cell c' if

there is at least one neighbor cell other than the cells c of p with nonzero number of red points. If there is at least one such cell, and we have an insertion of a red point, then the $flag_{c'}$ does not change. Otherwise, it is reset. On the contrary, if there is at least one such cell, and we have a deletion of a red point, then the $flag_{c'}$ does not change. Otherwise, it is set.

If the inserted (deleted) point is blue, for each subtree s on the path we update the *emptyBlue* word. To find the part of the *changeArray* that corresponds to s, we need to count the changes of the $flag_c$ values for each cell c in the path from the leaf of s in which the insertion (deletion) took place to the root of s, as only these cells are affected from the insertion (deletion). We fetch the constant number of *emptyRed* words of the subtrees of the same depth that contain neighbors $c'_{root} \in neighbors_{c_{root}}$ of the root cell c_{root} of s (these words contain also the neighbors for all cells c in the subtree s). With these words, if we have an insertion, we can determine if $flag_c$ for each cell c needs to be set by checking if there is at least one neighbor cell c' with nonzero number of red points. On the contrary, if we have a deletion, we only need to reset $flag_c$.

2.2.4 Choosing the value of b

We notice that the number of subtrees that need to be fetched while processing a subtree s is constant because a cell in a grid has only a constant number of neighbors. Supposing that the word operations take constant time, the update requires $O(\frac{\log U}{b})$ RAM words to be accessed and processed because there are that many subtrees on the path from the root to the leaf in which the update was done. The value of b needs to be maximized in order to minimize the update time, but there is an upper bound on b that is imposed by the need to fit all the *empty* flags of a subtree of depth b in a single word. More concretely, there are $O(2^{db})$ nodes in a tree of branching factor b and depth d, and each node needs only a bit for its flag. Therefore, we choose b to be $\delta \log\log n$ for a sufficiently small constant $\delta > 0$, so that $O(2^{db}) = o(\log n)$. Since the space usage is $O(n)$ words of nonempty subtrees with the same root depth and there are $O(\frac{\log U}{\log\log n})$ different depths of subtrees' roots, our data structure requires $O(n\frac{\log U}{\log\log n})$ words.

The word operations described in the previous section require only constant time, as it is easy to create a look-up table of $o(n)$ words with preprocessing time of $o(n)$. This is because the number of bits in a word in the above word operations is $O(2^{db}) = o(\log n)$, so the number of possible words is sublinear.

2.2.5 Maintaining the *globalCountArray* in $O(\frac{\log U}{\log\log n})$ worst-case time

Since each update adds or subtracts at each index of *globalCountArray* a is bounded by $O(1)$, which is dependent on the number of set $flag_{c'}$ for $c' \in neighbors_c$, the changes at every position of that array are of constant size per update, so we can encode $O(\log\log n)$ consecutive indexes of *changeArray* in a single word; thus we maintain an array called *tempCountArray* of size $O(\log U/\log\log n)$ words. In a period of $O(\log U/\log\log n)$ updates, we store the changes in the *tempCountArray* array in $O(\frac{\log U}{\log\log n})$ time per update. After a period of $O(\log U/\log\log n)$ updates, we update the *globalCountArray* in $O(\log U)$ steps. Therefore, the overall update of the *globalCountArray* can be done in $O(\frac{\log U}{\log\log n} + \log\log n) = O(\frac{\log U}{\log\log n})$ amortized time. A decision query to an index of *globalCountArray* is accomplished by adding the value at that position with the corresponding value from the *tempCountArray* in constant worst-case time.

The aforementioned data structure can be deamortized by following ideas by Dietz [2] for the partial sums problem. In a sequence of $O(\frac{\log U}{\log\log n})$ updates, we do all the steps that were described in the previous paragraph, but we also update *globalCountArray*[i mod $\frac{\log U}{\log\log n}$] in the i^{th} update, by adding to it the value in *tempCountArray*[i] in $O(\log\log U)$ time per update. It is easy to see that the words in the *tempCountArray* do not exceed the word size. Therefore, we obtain $O(\frac{\log U}{\log\log n})$ update time of *globalCountArray* in worst case.

We conclude that, with the above data structure we can answer the decision query for $r \in \{2^i \mid 0 \leq i \leq \log U\}$ using $O(n\frac{\log U}{\log\log n})$ words, $O(n\frac{\log U}{\log\log n} + n) = O(n\frac{\log U}{\log\log n})$ preprocessing time, $O(\frac{\log U}{\log\log n})$ update time, and constant query time.

2.3 Original version of the problem

With the current scheme we can maintain fully dynamically the Hausdorff distance with an approximation factor of 2. We now show how to make the data structure work with approximation factor of $1 + \varepsilon$. Let k be a constant parameter to be determined later. First create k instances of the data structure described in section 2.2 for distances $r_i = 2^{\frac{ki+j}{k}}$ with each data structure having a subscript $j = 0, \ldots, k-1$. The only twist that needs to be done to the j^{th} data structure is to rescale the coordinates of points by a factor of $\frac{1}{2^{j/k}}$.

To handle an update, we update the information to each one of the k data structures, and since k is constant (because it is dependent only on ε), and an update to one data structure requires $O(\frac{\log U}{\log\log n})$ time, one update takes $O(\frac{\log U}{\log\log n})$ time.

To perform a distance query to compute an approximation of the Hausdorff distance, we collect from each one of the k data structure the biggest distance that has an answer "yes" to a decision query. Afterwards, we return the the biggest of these distances. Both the decision and the distance query can take place in constant time.

It is clear that with the aforementioned scheme and with $k = \lfloor \frac{1}{\varepsilon} \rfloor$ we can obtain an approximation factor of $2^{\frac{1}{k}}(1+O(\varepsilon)) = 2^{O(\varepsilon)}(1+O(\varepsilon)) = (1+O(\varepsilon))(1+O(\varepsilon)) = 1 + O(\varepsilon)$, which can be made $1 + \varepsilon$ if we readjust ε. Therefore, the original version of the problem can be solved in $O(\frac{\log U}{\log \log n})$ expected update time.

3 Conclusion

Finding solutions with still better update time (e.g. $O(\log \log U)$) or with linear space remains interesting open problems.

References

[1] T. M. Chan. Closest-point problems simplified on the RAM. *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete algorithms*, 472–473, 2002.

[2] P. F. Dietz. Optimal algorithms for List Indexing and Subset Rank. *Proceedings of the First Annual Workshop on Algorithms and Data Structures*, 39–46, 1989.

[3] David Eppstein. Dynamic Euclidean minimum spanning trees and extrema of binary functions. *Discrete & Computational Geometry*, 13(1):111–122, 1995.

[4] T. M. Chan. Semi-Online Maintenance of Geometric Optima and Measures. *SIAM Journal on Computing*, 32(3):700–716, 2003.