Dynamic Generalized Closest Pair: Revisiting Eppstein's Technique

Timothy M. Chan*

October 28, 2019

Abstract

Eppstein (1995) gave a technique to transform any data structure for dynamic nearest neighbor queries into a data structure for dynamic closest pair, for any distance function; the transformation increases the time bound by two logarithmic factors. We present a similar, simple transformation that is just as good, and can avoid the extra logarithmic factors when the query and update time of the given structure exceed n^{ε} for some constant $\varepsilon > 0$.

Consequently, in the case of an arbitrary distance function, we obtain an optimal $O(n)$-space data structure to maintain the dynamic closest pair of n points in $O(n)$ amortized time plus $O(n)$ distance evaluations per update.

1 Introduction

In 1995, David Eppstein [9] published a paper (earlier version of which appeared in [2]) describing a useful technique to obtain dynamic data structures for closest-pair-like problems. The technique has since found numerous applications in computational geometry [1, 5, 7, 11, 12, 13], for example, to dynamic geometric minimum spanning trees, dynamic collision detection, semi-dynamic planar width, and static algorithms for straight skeletons, geometric shortest paths, geometric matchings, etc. In this note, we present a new simple alternative that yields small improvements in certain cases.

The abstract setting. Let $d : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ be a distance function. The problem is to design a dynamic data structure for two sets $P \subset \mathcal{X}$ and $Q \subset \mathcal{Y}$ of at most n elements (which we will refer to as "points"), which can

- maintain the *closest pair* $(p, q) \in P \times Q$ (i.e., the pair (p, q) minimizing $d(p, q)$), and
- support insertions and deletions of points in both P and Q.

We assume that there is available an initial dynamic data structure for any set $P \subset \mathcal{X}$ of at most n points, with $P_0(n)$ preprocessing time and $S_0(n)$ space, which can

- find the *nearest neighbor* in P to any given point $q \in \mathcal{Y}$ (i.e., the point $p \in P$ minimizing $d(p, q)$) in $Q_0(n)$ time, and

*Department of Computer Science, University of Illinois at Urbana-Champaign (tmc@illinois.edu). Work supported in part by NSF Grant CCF-1814026.

- support insertion of a point in P in $I_0(n)$ (amortized) time and deletion of a point in P in $D_0(n)$ (amortized) time.

We also assume that there is a symmetric dynamic data structure with the same bounds, for finding the nearest neighbor of any point $p \in \mathcal{X}$ to a dynamic set $Q \subset \mathcal{Y}$. Note that a solution to the dynamic closest pair problem necessarily requires a solution to the above dynamic nearest neighbor query problems, since the latter can obviously be reduced to the former.

Given such dynamic nearest neighbor structures, Eppstein designed a data structure for dynamic closest pair with $O(S_0(n))$ space and

- $O((Q_0(n) + I_0(n) + D_0(n)) \log n)$ amortized insertion time and
- $O((Q_0(n) + I_0(n) + D_0(n)) \log^2 n)$ amortized deletion time.

(An earlier result by Dobkin and Suri [8] addressed only the special case of "semi-online updates".)

Note that the abstract problem as formulated above is entirely non-geometric, in that no additional properties about the distance function $d(\cdot, \cdot)$ are required. When combined with known geometric data structures for dynamic nearest neighbors under specific distance functions, Eppstein's technique led to the best known results for dynamic monochromatic/bichromatic closest/farthest pair in a variety of geometric settings (although very recently a direct, slightly faster method was given in the 2-d Euclidean case by the author [6]).

The most general case. A later paper by Eppstein [10] (see also [4]) redescribed the algorithm in the most general case where we assume nothing about the distance function except that it can be evaluated in unit time. Here, we have $Q_0(n), I_0(n), D_0(n) = O(n)$ trivially, and the resulting dynamic closest pair data structure requires $O(n)$ space, $O(n \log n)$ amortized insertion time, and $O(n \log^2 n)$ amortized deletion time ($\Omega(n)$ is a lower bound on the update time in this case). Several applications of this general result, for example, to hierarchical clustering or the implementation of certain TSP heuristics, were described in the paper [10], which also contained an alternative solution with $O(n)$ amortized insertion and deletion time. However, in this alternative solution (based on a simple quadtree with leaves corresponding to all n^2 entries of the distance matrix), the space usage increases to $O(n^2)$. Eppstein wrote: "It remains open whether quadratic space is required to achieve linear time per update."

New result. We present a new data structure for dynamic closest pair achieving $O(S_0(n))$ space and

- $O(Q_0(n) \log n + (P_0(n)/n) \log n)$ amortized insertion time and
- $O(Q_0(n) \log^2 n + (P_0(n)/n) \log^2 n + D_0(n) \log n)$ amortized deletion time,

assuming that $Q_0(n)$ exceeds $\log n$. These bounds are at least as good as Eppstein's, since $P_0(n)/n$ is trivially upper-bounded by $I_0(n)$. In fact, there is one fewer logarithmic factor in the $O(D_0(n) \log n)$ term, which matters if deletions are sufficiently more expensive than insertions and queries for dynamic nearest neighbors (as is the case in some applications). Also note that there is no appearance of $D_0(n)$ in the insertion time bound, and there is no appearance of $I_0(n)$ at all, i.e., we do not need insertions in the given dynamic nearest neighbor structure. (One could apply the standard "logarithmic method" to automatically transform a deletion-only data structure for nearest neighbor

queries into one that supports insertions [3], since nearest neighbor queries are "decomposable", but the transformation would increase the query and insertion time by one more logarithmic factor.)

Our data structure also has smaller preprocessing time: $O(nQ_0(n) + P_0(n))$ instead of $O(nQ_0(n) + P_0(n) + nD_0(n))$.

There is one more important advantage: if $Q_0(n)/n^\varepsilon$, $P_0(n)/n^{1+\varepsilon}$, and $D_0(n)/n^\varepsilon$ are monotonically increasing for an arbitrarily small constant $\varepsilon > 0$, then the amortized insertion and deletion time can be further improved to $O(Q_0(n)+P_0(n)/n)$ and $O(Q_0(n)+P_0(n)/n+D_0(n))$, eliminating all the extra logarithmic factors.

Consequently, in the most general case, with the trivial bounds $Q_0(n) = O(n)$, $P_0(n) = O(n^2)$ and $D_0(n) = O(n)$, we obtain an (optimal) data structure with $O(n)$ space and $O(n)$ amortized insertion and deletion time, thus resolving Eppstein's aforementioned open question.

Overview. Eppstein's method is simple but clever. Roughly, we maintain logarithmically many layers, where the i-th layer stores subsets $P_i \subset P$ and $Q_i \subset Q$ whose sizes form a geometric progression (like in the standard "logarithmic method" [3]). In each layer, we build an *ordered nearest neighbor path* over $P_i \cup Q$ of length $2|P_i|$ (aptly called a "conga line" in Eppstein's later presentation [10]), where each point $p \in P_i$ is linked to its nearest neighbor among all points in Q that appear after p along the path, and each point $q \in Q$ is linked to its nearest neighbor among all points in P_i that appear after q along the path. A similar path is built over $P \cup Q_i$. The key feature of a path is that the maximum in-degree is 1; a deletion in a layer thus requires "fixing" only $O(1)$ points, which is done by reinserting these points (initially placed at the bottommost layer, but later elevated to higher layers due to periodic merging, as in the logarithmic method). The description of the entire data structure is concise, though the details are subtle.

In the standard logarithmic method, the time bound does not increase when it grows faster than n^ε. However, the same cannot be said about Eppstein's data structure, since the ordered nearest neighbor path is built between P_i and the *global* set Q, thus requiring a global dynamic nearest neighbor structure at every level.

In our new data structure, we build a more flexible graph, where the in-degree is $O(1)$ rather than 1 (Eppstein did suggest this as a possible variant at the end of his paper [9], but did not pursue the idea in detail). At each layer, our approach does not need to refer to the global point sets. This enables the improvements in the case when the time bound exceeds n^ε; it also allows the entire method to be described recursively, and the proof of correctness becomes more self-evident. We hope that some may find the new solution more intuitive.

There is some similarity with the author's recent $O(\log^4 n)$-time dynamic closest pair data structure in the 2-d Euclidean case [6], which was a direct modification of a dynamic 2-d nearest neighbor data structure with the same update time bound, but the solution there was specific to the 2-d case (using a geometric construction known as "shallow cuttings").

2 The Solution

Our data structure uses two parameters b and Δ, with $\Delta \geq 2b$. For simplicity, we assume that all distances are distinct (by perturbation, or by straightforward modifications to the algorithm).

The data structure for (P, Q). We maintain a bipartite graph $G_{P,Q}$ with vertex sets P and Q (where we think of the edges as directed from P to Q, as shown in Figure 1), along with a partition of Q into two subsets Q_{good} and Q_{bad}, satisfying the following properties:

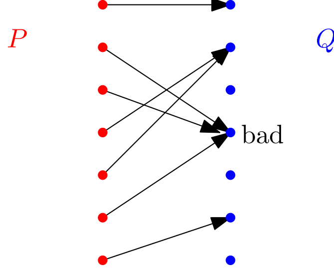

Figure 1: The bipartite graph $G_{P,Q}$.

1. each point $p \in P$ has out-degree 1;
2. each point $q \in Q$ has in-degree at most 2Δ;
3. if (p, q) is an edge in $G_{P,Q}$, then p is closer to q than to any other point in $Q_{\rm good}$.

We similarly maintain a bipartite graph $G_{Q,P}$, along with a partition of P into two subsets $P_{\rm good}$ and $P_{\rm bad}$, with similar properties except with P and Q swapped. The edges of both bipartite graphs are stored in a heap, ordered by distances. We store $P_{\rm good}$ and $Q_{\rm good}$ in the given dynamic nearest neighbor data structures. We recursively maintain a data structure for $(P_{\rm bad}, Q_{\rm bad})$. The smaller of the heap's minimum and the closest pair in $(P_{\rm bad}, Q_{\rm bad})$ gives us the overall closest pair. The recursion terminates when $n \le b$ (here, we just maintain the closest pair by brute force in $O(b)$ time).

Correctness follows by an easy case analysis: Let (p^*, q^*) be the actual closest pair in $P \times Q$. If $q^* \in Q_{\rm good}$, then (p^*, q^*) must be in $G_{P,Q}$ by property 3, and so is in the heap. If $p^* \in P_{\rm good}$, then (p^*, q^*) is in the heap by a symmetric argument. Finally, if $p^* \in P_{\rm bad}$ and $q^* \in Q_{\rm bad}$, then (p^*, q^*) is found by recursion.

Preprocessing (P, Q)**.** We construct the bipartite graph $G_{P,Q}$, along with $Q_{\rm good}$ and $Q_{\rm bad}$, satisfying the above properties, in the most intuitive way: we repeatedly link points of P to its nearest neighbors in Q, but as soon as a point $q \in Q$ has in-degree exceeding Δ, we mark q as bad and remove it (though the edges linked to q are kept in the graph). More precisely:

> $Q_{\rm good} = Q$, $Q_{\rm bad} = \emptyset$, $G_{P,Q} = \emptyset$
> for each $p \in P$ do:
>   find p's nearest neighbor q in $Q_{\rm good}$ in $O(Q_0(n))$ time
>   add (p, q) to $G_{P,Q}$ (and to the heap)
>   if q's in-degree reaches Δ then
>   delete q from $Q_{\rm good}$ in $O(D_0(n))$ time and insert q to $Q_{\rm bad}$

We similarly construct the bipartite graph $G_{Q,P}$, along with $P_{\rm good}$ and $P_{\rm bad}$. We then recursively preprocess $(P_{\rm bad}, Q_{\rm bad})$.

If $|P|, |Q| \le n$, then $|P_{\rm bad}|, |Q_{\rm bad}| \le n/\Delta$. The total preprocessing time obeys the recurrence

$$P(n) \;\le\; P(n/\Delta) + O(nQ_0(n) + P_0(n) + (n/\Delta)D_0(n)).$$

Inserting a point p **to** P**.** (Note: inserting a point to Q can be handled symmetrically.)

We find p's nearest neighbor q in $Q_{\rm good}$ in $O(Q_0(n))$ time, add (p,q) to $G_{P,Q}$ (and to the heap), and if q's in-degree reaches 2Δ, delete q from $Q_{\rm good}$ in $O(D_0(n))$ time, and insert q to $Q_{\rm bad}$ recursively in the data structure for $(P_{\rm bad}, Q_{\rm bad})$. It takes $\Omega(\Delta)$ edge changes in $G_{P,Q}$ before q's in-degree reaches 2Δ, and each insertion to P causes just one edge change. Thus, each insertion to P triggers an amortized $O(1/\Delta)$ number of deletions from $Q_{\rm good}$ and insertions to $Q_{\rm bad}$.

In addition, we do not change $G_{Q,P}$ but just recursively insert p to $P_{\rm bad}$.

When the size of $P_{\rm bad}$ or $Q_{\rm bad}$ reaches n/b, we rebuild the entire data structure for (P,Q) in $O(P(n))$ time. It takes $\Omega(n/b)$ updates before a rebuild can occur, and thus the amortized cost for rebuilding is $O(P(n)\cdot b/n)$.

The overall amortized insertion time satisfies the recurrence

$$I(n) \;\le\; (1+O(1/\Delta))I(n/b) \;+\; O(Q_0(n)+\log n+D_0(n)/\Delta+bP(n)/n).$$

Deleting a point q from Q. (Note: deleting a point from P can be handled symmetrically.)

If $q\in Q_{\rm good}$, we delete q from $Q_{\rm good}$ in $O(D_0(n))$ time, else we recursively delete q from $Q_{\rm bad}$. In either case, we find the at most 2Δ points p that are linked to q in $G_{P,Q}$. For each such p, we find p's new nearest neighbor q' in $Q_{\rm good}$ in $O(Q_0(n))$ time, replace (p,q) with (p,q') in $G_{P,Q}$ (and in the heap), and if the in-degree of q' reaches 2Δ, delete q' from $Q_{\rm good}$ in $O(D_0(n))$ time, and insert q' to $Q_{\rm bad}$ in $I(n/b)$ amortized time. It takes $\Omega(\Delta)$ edge changes in $G_{P,Q}$ before a vertex's in-degree reaches 2Δ, and each deletion from Q causes $O(\Delta)$ edge changes. Thus, each deletion from Q triggers an amortized $O(1)$ number of deletions from $Q_{\rm good}$ and insertions to $Q_{\rm bad}$.

In addition, we remove the single edge incident to q from $G_{Q,P}$ (and from the heap).

Again, when the size of $P_{\rm bad}$ or $Q_{\rm bad}$ reaches n/b, we rebuild the entire data structure for (P,Q) in $O(P(n))$ time. It takes $\Omega(n/b)$ updates before a rebuild can occur, and thus the amortized cost for rebuilding is $O(P(n)\cdot b/n)$.

The overall amortized deletion time satisfies the recurrence

$$D(n) \;\le\; D(n/b) \;+\; O(\Delta Q_0(n)+\Delta\log n+D_0(n)+bP(n)/n+I(n/b)).$$

Space. The total space of the data structure satisfies the recurrence

$$S(n) \;\le\; S(n/b)+O(n+S_0(n)).$$

Solving the recurrences. Assume that $P_0(n)/n$, $S_0(n)/n$, $Q_0(n)/\log n$, $I_0(n)$, and $D_0(n)$ are monotonically increasing. Clearly, $S(n)=O(S_0(n))$.

- *In the case when $Q_0(n)/n^{\varepsilon}$, $P_0(n)/n^{1+\varepsilon}$, and $D_0(n)/n^{\varepsilon}$ are monotonically increasing*: We set $b=2$ and $\Delta=\max\{c, D_0(n)/Q_0(n)\}$ for some constant $c\ge 2$. Then
  - $P(n)\le P(n/c)+O(nQ_0(n)+P_0(n))$, which solves to $P(n)=O(nQ_0(n)+P_0(n))$;
  - $I(n)\le(1+O(1/c))I(n/2)+O(Q_0(n)+P_0(n)/n)$, which solves to $I(n)=O(Q_0(n)+P_0(n)/n)$, by picking a sufficiently large constant c (depending on ε);
  - $D(n)\le D(n/2)+O(Q_0(n)+D_0(n)+P_0(n)/n)$, which solves to $D(n)=O(Q_0(n)+D_0(n)+P_0(n)/n)$.

- *Otherwise*: We set $\Delta = \max\{t, D_0(n)/Q_0(n)\}$ for some fixed parameter $t \geq 2$. Then
  - $P(n) \leq P(n/t) + O(nQ_0(n) + P_0(n))$, which solves to $P(n) = O(nQ_0(n) + P_0(n))$;
  - $I(n) \leq (1 + O(1/t))I(n/b) + O(bQ_0(n) + bP_0(n)/n)$, which solves to $I(n) = O((bQ_0(n) + bP_0(n)/n) \cdot \log_b n \cdot (1 + O(1/t))^{\log_b n})$;
  - $D(n) \leq D(n/b) + O(tQ_0(n) + D_0(n)) + O((bQ_0(n) + bP_0(n)/n) \cdot \log_b n \cdot (1 + O(1/t))^{\log_b n})$, which solves to $D(n) = O((tQ_0(n) + D_0(n)) \cdot \log_b n) + O((bQ_0(n) + bP_0(n)/n) \cdot (\log_b n)^2 \cdot (1 + O(1/t))^{\log_b n})$.

  We now set $t = \max\{2b, \log_b n\}$ (this ensures that $\Delta \geq 2b$ and $(1 + O(1/t))^{\log_b n} = O(1)$). Then $I(n) = O(bQ_0(n) \log n + b(P_0(n)/n) \log n)$ and $D(n) = O(bQ_0(n) \log^2 n + D_0(n) \log_b n + b(P_0(n)/n) \log^2 n)$.

Theorem 1. *Assume that $P_0(n)/n$, $S_0(n)/n$, $Q_0(n)/\log n$, $I_0(n)$, and $D_0(n)$ are monotonically increasing. There is a data structure for the dynamic closest pair problem with $O(nQ_0(n) + P_0(n))$ preprocessing time and $O(S_0(n))$ space, with*

(i) *$O(Q_0(n) + P_0(n)/n)$ amortized insertion time and $O(Q_0(n) + P_0(n)/n + D_0(n))$ amortized deletion time, if $Q_0(n)/n^\varepsilon$, $P_0(n)/n^{1+\varepsilon}$, and $D_0(n)/n^\varepsilon$ are monotonically increasing for some constant $\varepsilon > 0$; or*

(ii) *$O(bQ_0(n) \log n + b(P_0(n)/n) \log n)$ amortized insertion time, and $O(bQ_0(n) \log^2 n + b(P_0(n)/n) \log^2 n + D_0(n) \log_b n)$ amortized deletion time, for any given b.*

In (ii), the simplest option is to set b to a constant. In the case when $D_0(n)$ is larger than $(Q_0(n) + P_0(n)/n) \log^{1+\varepsilon} n$, we may even set $b = \log^\varepsilon n$ for a small $(\log \log n)$-factor improvement in the final deletion time bound.

References

[1] P. K. Agarwal, A. Efrat, and M. Sharir. Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. *SIAM J. Comput.*, 29(3):912–953, 1999.

[2] P. K. Agarwal, D. Eppstein, and J. Matoušek. Dynamic half-space reporting, geometric optimization, and minimum spanning trees. In *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 80–89, 1992.

[3] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[4] J. Cardinal and D. Eppstein. Lazy algorithms for dynamic closest pair with arbitary distance measures. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 112–119, 2004.

[5] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *J. ACM*, 57(3):16:1–16:15, 2010.

[6] T. M. Chan. Dynamic geometric data structures via shallow cuttings. In *Proceedings of the 35th Symposium on Computational Geometry (SoCG)*, pages 24:1–24:13, 2019.

[7] T. M. Chan and A. Efrat. Fly cheaply: On the minimum fuel consumption problem. *J. Algorithms*, 41(2):330–337, 2001.

[8] D. P. Dobkin and S. Suri. Maintenance of geometric extrema. *J. ACM*, 38(2):275–298, 1991.

[9] D. Eppstein. Dynamic Euclidean minimum spanning trees and extrema of binary functions. *Discrete & Computational Geometry*, 13:111–122, 1995.

[10] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *ACM Journal of Experimental Algorithmics*, 5:1, 2000. Preliminary version in SODA 1998.

[11] D. Eppstein. Incremental and decremental maintenance of planar width. *J. Algorithms*, 37(2):570–577, 2000.

[12] D. Eppstein and J. Erickson. Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. *Discrete & Computational Geometry*, 22(4):569–592, 1999.

[13] H. Kaplan, W. Mulzer, L. Roditty, P. Seiferth, and M. Sharir. Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications. In *Proceedings of the 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2495–2504, 2017.