# Time-Windowed Closest Pair*

Timothy M. Chan† Simon Pratt†

**Abstract**

Given a set of points in any constant dimension, each of which is associated with a time during which that point is active, we design a data structure with $O(n \log n)$ space that can find the closest pair of active points within a query interval of time in $O(\log \log n)$ time using a quadtree-based approach in the word-RAM model.

## 1 Introduction

Let $P$ be a set of $n$ points in $\mathbb{R}^d$. Additionally, let each point $p \in P$ be associated with a time $t_p$ at which that point is active. In the *time-windowed closest pair problem*, we want to preprocess $P$ into a data structure that can efficiently determine the closest pair of points that are active during an interval of time $[t_1, t_2]$ called a *time window*.

Time-windowed geometric problems are motivated by *Geographic Information System* (GIS) data which sometimes consists not only of longitude, latitude, and altitude coordinates but also time. Bannister *et al.* [3] examined time-windowed versions of convex hull, approximate spherical range searching, and approximate nearest neighbor queries. Their solution for this last problem is to store the points in a balanced binary search tree indexed by time, and store the Z-order (also known as Morton or shuffle order) [4] of subsets of the points. However, the time-windowed closest pair problem appears more challenging, because it involves minimizing *quadratically* many pairs.

We can consider the time of each point as its coordinate in the $(d+1)^{\text{th}}$ dimension. If we do so, the problem becomes finding the closest pair of points within a strip in $\mathbb{R}^{d+1}$. Sharathkumar and Gupta [10] wrote a technical report in which they solve the problem of finding the closest pair in 2 dimensions within a query range, a special case of which is when the range is a strip. Their solution can be used to solve the time-windowed closest pair problem in 1 dimension in $O\big(n \log^2 n\big)$ space and $O(\log n)$ query time.

Our approach answers queries in $O(\log \log n)$ time using $O(n \log n)$ space and $O(n \log n \log \log n)$ preprocessing time in the $w$-bit word-RAM model for any constant dimension $d$. (The algorithm finds the exact closest pair.) Here, we assume the time value of each point is an integer from 0 to $n-1$ and that each point has a distinct time value. If time is given as a $w$-bit integer instead, we can pre-sort the time values and replace them by their ranks; this adds the cost of a predecessor search to the query time (which is at most $O\Big(\min\{\log w, \log_w n, \sqrt{\log n / \log \log n}\}\Big)$ [9]).

The main idea behind our approach is to compute a centroid cell $B$ of a quadtree, consider $O(n)$ pairs in which one point is inside the centroid cell and the other is outside, then recurse on $P \cap B$ and $P \setminus B$.

The idea of the centroid cell of a quadtree is due to Arya *et al.* [2]. In that paper they describe a data structure related to quadtrees called a *Balanced Box Decomposition* (BBD) tree, which they use to answer approximate nearest neighbor queries efficiently.

## 2 Preliminaries

We work in the $w$-bit word-RAM model, which models the computer as a sequence of $w$-bit memory locations each indexed by a $w$-bit integer. In this model, we assume that $w \geq \log n$ and standard operations on words take constant time.

Let $P$ be a set of $n$ points in $\mathbb{R}^d$, and $B$ is a hypercube (which we call a *cell*) containing those points. To build the *quadtree* for $P$, we divide $B$ into $2^d$ congruent child hypercubes. For each of these child hypercubes which contain more than a single point, we recursively build a quadtree for that box.

The following lemma, similar to Lemma 4 from Arya *et al.* [2], bounds the number of points in a hypercube by a constant with respect to the hypercube's side length and the distance between the closest pair within that hypercube.

**Lemma 1 (Packing)** *If a point set has closest-pair distance at least $r$ and lies in a $d$-dimensional hypercube with side length at most $b \cdot r$, then there are less than $c_0(b+1)^d$ points, where $c_0$ is some constant that depends only on $d$. We call $c_0$ the* packing constant*.*

The following lemma is due to Arya *et al.* [2] (Lemma 1).

**Lemma 2 (Centroid)** *Given a point set $P$ containing $n$ points, there exists a quadtree cell $B$, which we call a*

*Research supported by The Natural Sciences and Engineering Research Council of Canada and the Ontario Graduate Scholarship.

†{tmchan,s2pratt}@cs.uwaterloo.ca

centroid cell, *such that* $|P \cap B| \leq \alpha n$ *and* $|P \setminus B| \leq \alpha n$ *for some constant* $\alpha < 1$ *that depends only on* $d$.

Recursive application of this lemma gives a data structure on a set of points $P$, called a *balanced quadtree* [5], defined as a binary tree where the root stores $B$, the left subtree is the balanced quadtree for $P \cap B$, and the right subtree is the balanced quadtree for $P \setminus B$ where $B$ is a centroid cell of $P$.

We have the following lemma due to Chan [5] (Observation 3.2, Lemma 3.3), that says if we draw a constant number of grids over our points, each shifted by some amount, then we can guarantee that any pair of points must be in the same cell in at least one such grid. Since quadtree cells are related to grid cells, this also implies that the closest pair will be in the same quadtree cell if we build a constant number of quadtrees.

**Lemma 3 (Shifting)** *Suppose* $d$ *is even. Let* $v^{(j)} = (\lfloor j2^w/(d+1) \rfloor, \ldots, \lfloor j2^w/(d+1) \rfloor) \in \mathbb{R}^d$. *For any points* $p$ *and* $q$ *and* $r = 2^\ell$ *such that* $||p-q||_\infty \leq r$, *there exists* $j \in \{0, 1, \ldots, d\}$ *such that* $p + v^{(j)}$ *and* $q + v^{(j)}$ *belong to the same* $c_1 r$*-grid cell, where* $c_1$ *is the smallest power of 2 bigger than or equal to* $2d+2$. *We call* $c_1$ *the* shifting constant.

While the preceding lemma requires that $d$ be even, for odd values of $d$ we can use $d+1$.

## 3 Decision Problem

Before we solve the time-windowed closest pair problem, it helps to consider the decision problem version, in which we are additionally given a fixed distance $r$ and we want to preprocess $P$ into a data structure which can efficiently determine, for any query time window, if there exists a pair of active points $pq$ such that the distance between $p$ and $q$ is at most $r$. We call such a pair a *satisfying pair*.

The main idea of our approach is to use a constant number of shifted grids, which by Lemma 3 ensures that any two points will appear in the same cell together in at least one such shifted grid. For each point, we consider a constant number of its time-order predecessors and successors within the same cell, which by Lemma 1 we know must include a satisfying pair if one exists. From there, we reduce the problem to a standard 2-dimensional dominance range searching problem.

### 3.1 Computing Candidate Satisfying Pairs

We begin by bucketing the points of $P$ into grid cells with side length $c_1 r'$, where $c_1$ is the shifting constant from Lemma 3 and $r'$ is the smallest power of 2 bigger than or equal to $r$. Each grid cell is assigned a label $\ell$ and for each point $p$ we create a tuple $(\ell, t_p, p)$. We can determine the label of the grid cell containing each point by hashing in $O(n)$ total expected time.

For each cell, we build a time-ordered array of the points within that cell. This is done by running radix sort on the tuples created in the previous step, sorting first by grid cell label, and then by time. The radix sort takes $O(n)$ time.

For each such point $p$, we consider its $c_0(c_1+1)^d$ predecessors and the same number of successors in the time-ordered array, where $c_0$ is the packing constant from Lemma 1. Let $q$ be such a predecessor or successor. If the distance between $p$ and $q$ is at most $r$, then $t_p$ and $t_q$ form a *candidate satisfying pair*.

We do the preceding steps $d+1$ times, where each time the cells are shifted by $v^{(j)}$ for $j \in \{0, 1, \ldots, d\}$ as defined in Lemma 3. We union together the results to build the full set of candidate satisfying pairs.

**Lemma 4** *There are* $O(n)$ *candidate satisfying pairs.*

**Proof.** Over the $d+1$ shifts, the total is upper-bounded by $(d+1) \cdot c_0(c_1+1)^d n = O(n)$.

□

**Lemma 5** *If a time window contains a satisfying pair, then the time window must contain a candidate satisfying pair.*

**Proof.** Let $pq$ be a satisfying pair for the window which is closest in terms of time order. From Lemma 3, there exists $j \in \{0, 1, \ldots, d\}$ such that $p + v^{(j)}$ (which we will call $p'$) and $q + v^{(j)}$ (which we will call $q'$) are in the same grid cell of side length $c_1 r$. Since $p'$ and $q'$ are active, all points between them in time order must also be active. No two points strictly between $p'$ and $q'$ can have distance smaller than $r$, for otherwise we would have a satisfying pair that is closer than $pq$ in terms of time order. By Lemma 1 there are less than $c_0(c_1+1)^d$ points strictly between $p'$ and $q'$. Therefore $p'q'$ must be among the candidate satisfying pairs.

□

### 3.2 Reduction to 2D Dominance Range Emptiness

Now that the number of pairs we need to consider is reduced to $O(n)$, we would like to store these pairs in a data structure to support efficient querying. Specifically, given a query window $[t_1, t_2]$ we wish to determine if there exists a candidate satisfying pair $pq$ such that $t_1 \leq \min\{t_p, t_q\}$ and $t_2 \geq \max\{t_p, t_q\}$.

Consider each candidate satisfying pair as a point in 2 dimensions with coordinates $(-\min\{t_p, t_q\}, \max\{t_p, t_q\})$. Our query problem is equivalent to determining whether the quadrant $(-\infty, -t_1] \times (-\infty, t_2]$ contains any of these points. This is exactly the *2D dominance range emptiness* problem. This problem can be solved by computing

the *minima* of the 2D point set [8] and testing whether the query point is above or below the *staircase* formed by the minima. Computing the minima of $O(n)$ points takes $O(n)$ time by a standard sweep-line algorithm, assuming that the $x$-coordinates have been pre-sorted. Since the $x$-coordinates are in $\{0, \ldots, n-1\}$, pre-sorting takes $O(n)$ time.

We can use an array to store the $y$-value of the staircase at every $x$-value; this requires $O(n)$ words of space.

To save space, we can build a succinct rank/select data structure [7] with all the same time bounds but using just $2n + o(n)$ bits of space. We do so by considering the staircase as a monotone chain (after negating the $x$-coordinates) through the $n \times n$ grid from the origin to $(n-1, n-1)$. This grid is effectively a plot with start time on the $x$-axis, and end time on the $y$-axis. We can encode a monotone chain as a sequence of $2n$ bits. Starting at the origin, whenever the chain moves upwards from end time $i$ to $i+1$, we store a 0 bit. Similarly, whenever the chain moves rightwards, we store a 1 bit. The answer to the query is yes if and only if $t_2 \geq rank(select(t_1))$, where $select(i)$ denotes the position of the $i^{\text{th}}$ 1 in the sequence and $rank(j)$ denotes the number of 1s in the first $j$ positions of the sequence. The rank and select operations take constant time.

We have thus proven the following result:

**Theorem 6** *The decision problem version of the time-windowed closest pair problem in any fixed dimension can be solved in $O(1)$ time using $O(n)$ bits of space and $O(n)$ expected preprocessing time in the word-RAM model.*

## 4 Closest Pair

To solve the original time-windowed closest pair problem, the main new idea is to replace shifted grids with shifted balanced quadtrees. For each point outside of the centroid cell, we consider a constant number of its time-order predecessors and successors within the centroid cell. We then recurse separately on the points inside and outside of the centroid cell. This divide-and-conquer approach gives us $O(n \log n)$ candidate pairs. From there, we reduce the problem to a 2-dimensional dominance range minimum problem.

### 4.1 Computing Candidate Pairs

We describe our algorithm to generate candidate pairs recursively. We first compute the centroid cell $B$ of the given point set $P$. Define the set of neighbors $N(p)$ of a point $p$ as its $c_0(2c_1+1)^d$ time-order predecessors and successors within the centroid cell $B$, where $c_0$ is the packing constant from Lemma 1 and $c_1$ is the shifting constant from Lemma 3. For each point $p$ outside of the centroid, we consider each pair $pq$ for $q \in N(p)$ as a *candidate pair*. We then recurse on $P \cap B$ and $P \setminus B$.

We run the preceding algorithm $d+1$ times, where each time the quadtrees are shifted by $v^{(j)}$ for $j \in \{0, 1, \ldots, d\}$ as defined in Lemma 3. We union together the results to build the full set of candidate pairs.

**Lemma 7** *There are $O(n \log n)$ candidate pairs.*

**Proof.** For each fixed shift, the number of candidate pairs is given by the recurrence $P(n) \leq P(n_1) + P(n_2) + c_0(2c_1+1)^d n$, where $n_1$ and $n_2$ are the number of points inside and outside of the centroid respectively.

Since $n_1 + n_2 = n$ and $n_1, n_2 \leq \alpha n$, the recurrence solves to $P(n) = O(n \log n)$. □

**Lemma 8** *The closest pair for any time window must be among the candidate pairs.*

**Proof.** Let $pq$ be the closest pair in the window, with distance $r$. From Lemma 3, there exists $j \in \{0, 1, \ldots, d\}$ such that $p + v^{(j)}$ (which we will call $p'$) and $q + v^{(j)}$ (which we will call $q'$) are in the same quadtree cell of side length $c_1 r'$ where $r'$ is the smallest power of 2 greater than $r$.

There are 3 cases. Either $p', q'$ are both inside or outside of the centroid cell $B$, or one is inside and the other is outside of $B$. The first 2 cases can be handled by induction. Now we are in case 3, so suppose $q'$ is inside the centroid. (The case where $p'$ is inside the centroid is symmetric.)

From Lemma 1, there are no more than $c_0(2c_1+1)^d$ active points in the centroid cell $B$, since $B$ has side length at most $2c_1 r$. Since $p$ and $q$ are active during the time window, all points between them in time order must also be active. Therefore, there are less than $c_0(2c_1+1)^d$ points between $p$ and $q$ in time order, so $q \in N(p)$. □

### 4.2 Reduction to 2D Dominance Range Minimum

Now that the number of pairs we need to consider is reduced to $O(n \log n)$, we would like to store these pairs in a data structure to support efficient querying. Specifically, given a query window $[t_1, t_2]$ we wish to find a candidate pair $pq$ such that $t_1 \leq \min\{t_p, t_q\}$ and $t_2 \geq \max\{t_p, t_q\}$ while minimizing the distance $d(p, q)$.

Consider each candidate pair as a weighted point in 2 dimensions with coordinates $(-\min\{t_p, t_q\}, \max\{t_p, t_q\})$ and weight $d(p, q)$. Our query problem is equivalent to finding a point in the quadrant $(-\infty, -t_1] \times (-\infty, t_2]$ with the minimum weight. This is exactly the *2D dominance range minimum* problem, which we can solve by using standard techniques. Namely, we first lift the 2D weighted points to 3D where the weights become

$z$-coordinates. We compute the *staircase polyhedron* of the 3D point set, defined as the region of all points that are not dominated by any input point. Then a query can be answered by finding the highest point on the staircase polyhedron at a given $x$- and $y$-coordinate. Computing the staircase polyhedron is related to the standard problem of computing the *minima* of the 3D point set [8, 1] and can be done by a standard sweep-plane algorithm. For a set of $N$ points in 3D, the sweep-plane algorithm takes $O(N \log\log N)$ time using van Emde Boas trees, assuming that the $x$- and $y$-coordinates have been pre-sorted (the $z$-coordinates need not be pre-sorted). Since the $x$-coordinates are in $\{0, \ldots, n-1\}$, pre-sorting can be done in $O(N+n)$ time.

Finding the highest point of the staircase polyhedron (a monotone polyhedron in 3D) at a query $x$- and $y$-coordinate reduces to point location in a 2D subdivision of $O(N)$ size, after projecting the faces onto the $xy$-plane. We can use Chan's *planar orthogonal point location* structure [6] as a black box to answer queries in $O(\log\log N)$ time using $O(N)$ space and $O(N)$ preprocessing time.

Setting $N = O(n \log n)$ gives our main result:

**Theorem 9** *Time-windowed closest pair queries in any fixed dimension can be answered in $O(\log\log n)$ time using $O(n \log n)$ words of space and $O(n \log n \log\log n)$ preprocessing time in the word-RAM model.*

### 4.3 A Lower Bound on the Number of Candidate Pairs

As a final remark, we point out that any approach which stores all candidate pairs must use $\Omega(n \log n)$ space by proving the following observation.

**Observation 1** *There exists a set of $n$ points, where each point is associated with a time value, such that there are $\Omega(n \log n)$ distinct closest pairs over all possible time windows.*

**Proof.** Our construction works in one dimension. Suppose $n$ is a power of 2. The base case $n = 2$ is trivial. To construct a set $S$ of $n$ points on a line, we first recursively construct a set $S_1$ of $n/2$ points, and duplicate $S_1$ to create $S_2$. We increase the labels of points in $S_2$ by $n/2$ and we shift the points along the line by $\delta$ for a sufficiently small $\delta > 0$ (less than half of the closest pair distance in $S_1$). Since the time labels of $S_1$ and $S_2$ are disjoint, any closest pair between points in $S$ remains a closest pair for some time window. Symmetrically, we have the same closest pairs between points in $S_2$. In addition, for each time value $i \in \{1, \ldots, n/2\}$, the pair of points with time values $i$ and $i + n/2$ is a closest pair for the time window $[i, i+n/2]$, because the pair has the smallest possible distance $\delta$, and any other pair with distance $\delta$ has time values of the form $j$ and $j + n/2$, which can't both lie inside $[i, i+n/2]$. This gives $n/2$ additional closest pairs. Therefore, the number of distinct closest pairs is given by the recurrence $C(n) \geq 2C(n/2) + n/2$, which solves to $C(n) = \Omega(n \log n)$.

(Note: the construction can alternatively be described without recursion using *bit-reversal permutations*.) □

### References


[1] P. Afshani. Fast computation of output-sensitive maxima in a word RAM. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1414–1423. SIAM, 2014.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[3] M. J. Bannister, W. E. Devanny, M. T. Goodrich, J. A. Simons, and L. Trott. Windows into geometric events: Data structures for time-windowed querying of temporal point sets. In *Proceedings of the 26th Canadian Conference on Computational Geometry*, 2014.

[4] M. Bern, D. Eppstein, and S.-H. Teng. Parallel construction of quadtrees and quality triangulations. *International Journal of Computational Geometry & Applications*, 9(06):517–532, 1999.

[5] T. M. Chan. Approximate nearest neighbor queries revisited. *Discrete & Computational Geometry*, 20(3):359–373, 1998.

[6] T. M. Chan. Persistent predecessor search and orthogonal point location on the word RAM. *ACM Transactions on Algorithms (TALG)*, 9(3):22, 2013.

[7] J. I. Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science*, pages 37–42. Springer, 1996.

[8] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[9] Pătraşcu, Mihai. Predecessor search. In *Encyclopedia of Algorithms*. 2008.

[10] R. Sharathkumar and P. Gupta. Range-aggregate proximity queries. Technical report, IIIT/TR/2007/80, IIIT Hyderabad, 2007.