# Project Report

An Implementation Of:
A Simple Randomized $O(n\,log\,n)$–Time Closest-Pair Algorithm in Doubling Metrics

Minh Thang Cao

13 July 2020

## 1 Introduction

Implementation of an algorithm helps us observe its efficiency and behavior in practice. In this report, I will briefly explain each part of the closest-pair doubling algorithm [1], show the program's implementation along with practical running time analysis, and some implementation techniques I used. The theoretical information in this report fully refers to the work of A. Maheshwari, W. Mulzer and M. Smid, see [1].

Given a metric space $(P, dist)$, with doubling dimension $d$, whose $P$ is a set of $n$ points. The closest pair of points is the two points with the distance $\delta_0$ that satisfies $dist(p_1, p_2) \geq \delta_0$ for any point $p_1, p_2 \in P$. Also, the doubling dimension $d$ of a metric space indicates that for every point $p$ in $P$ and every real number $R > 0$, the $ball_P(p, R)$ can be covered by at most $2^d$ balls in $P$ of radius $R/2$, see [1, Section 2]. By using the definition and properties of the doubling metric space and its doubling dimension, the algorithm will find the closest-pair distance without the direct use of the points' coordinates in $O(n\,log\,n)$ time.

The closest-pair algorithm consists of three smaller parts:

1. Computing a separating annulus, denoted SepAnn$(S, n, d, \mu, c)$
2. The refinement of SepAnn$(S, n, d, \mu, c)$, denoted SparseSepAnn$(S, n, d, t)$
3. The main recursive closest-pair algorithm, denoted ClosestPair$(S, n, d)$

Throughout the paper, let:

- $(P, dist)$ be a finite metric space in which $P$ is the set of all points, and $dist$ is the function that calculate the distance between any two points
- $d$ be the space's doubling dimension
- $S$ be a non-empty subset of $P$

***<u>Note</u>***: I will only mention about 2D points because of the extremely high running time of the algorithm in the space more than 3D (in which the doubling dimension is approximately at least $\boldsymbol{log_2 21}$ that gives us the base case with more than 3,000,000 points).

## 2 Algorithm 1: Computing a separating annulus

An important part of the main closest-pair algorithm is finding a separating annulus in the subset $S$. I will briefly describe this algorithm in the next subsection, due to A. Maheshwari, W. Mulzer and M. Smid [1, Section 3.1].

### 2.1 The SepAnn$(S, n, d, \mu, c)$ algorithm

In this section, $\mu \geq 1$ is a real constant number, $c$ is calculated based on $\mu$ (I would say that $c = 2(4\mu)^d$ [1, Remark 1] since $\mu$ is not an integer in this case [1, Section 3.2]).

This algorithm picks a uniformly random point $p$ from the subset $S$ then finds the smallest ball centered at $p$, denoted $ball_S(p, R_p)$, that contains at least $n/c$ point. If the outer ball $ball_S(p, \mu R_p)$ contains at most $n/2$ points, it returns $p$ and $R_p$. If not, this procedure is repeated until the condition is satisfied. This algorithm's pseudocode is given below, see [1, Section 3.1].

```
Algorithm 1: SparseSepAnn$(S, n, d, t)$
Input: Let $S$ be a subset of $P$, of size $n$, $d$ be the metric space's doubling
       dimension, $\mu \geq 1$ and $c > 1$ be large enough real numbers.
Output: A point $p \in S$ and a radius $R_p > 0$.
    repeat
        p = a uniformly random point in $S$;
        $R_p = \min\{r > 0 : |ball_S(p, r)| \geq n/c\}$;
    until $|ball_S(p, \mu R_p)| \leq n/2$
    return $p$ and $R_p$
```

*This pseudocode is from [1, Section 3.1]*

### 2.2 Finding the $K^{th}$ smallest element

One step in SepAnn$(S, n, d, \mu, c)$ algorithm is to find the smallest ball which contains at least $n/c$ points. This ball is easy to find using the $k^{th}$ smallest element algorithm. Particularly, in a list of distances between $p$ and all other points in $S$, we pick the $\lceil n/c \rceil$-th smallest element and let it be the radius of the ball we need to find. Thus, all points closer to $p$ are inside this ball.

A very easy approach to find the $k^{th}$ smallest element in a list is to sort it in ascending order, and then simply return the element at the $k^{th}$ place. This sorting algorithm takes $O(n\,log\,n)$ time complexity in the worst case. Fortunately, we can improve the running time to $O(n)$ using a common recursive technique called QuickSelect, which is similar to QuickSort.

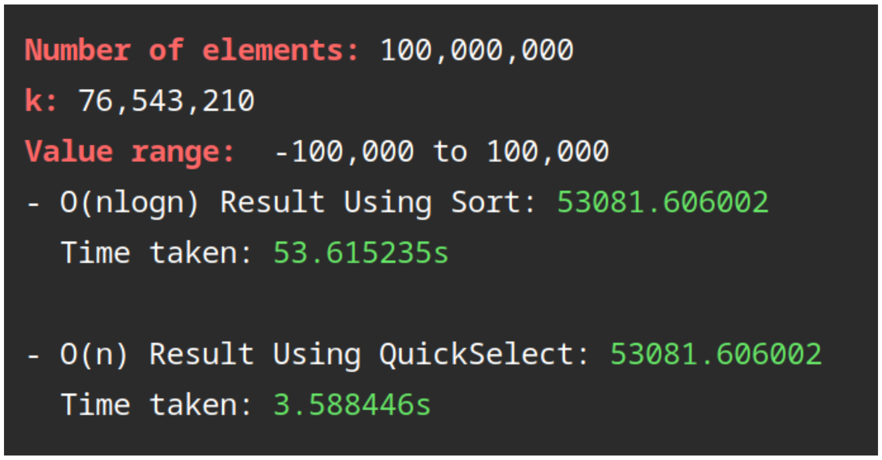
Given an unordered list $D$ which contains $n$ numbers, and a positive integer $k$ satisfies $1 \leq k \leq n$. Each element in $D$ has an index from 0 to $n - 1$. Consider a sublist of $D$, denoted $D[a : b]$, which starts from index $a$ and ends at $b$, inclusively. The base case is when $a = b$, the algorithm returns the only element in that sublist. If it is not the case, the algorithm will choose a random *pivot* in the list. The algorithm then rearranges the list so that all elements smaller and larger than the pivot are to the left and right of it, respectively. Now, the pivot has a new index, says $c$. If $k = c - a + 1$, the chosen pivot is the $k^{th}$ smallest element of $D$, the algorithm returns $D[c]$. If $k < c - a + 1$, the algorithm recurses on the sublist to the left of the pivot, $D[a : c - 1]$. If $k > c - a + 1$, the algorithm recurses on the sublist to the right of the pivot, $D[c + 1 : b]$. This algorithm's pseudocode is given below.

```
Algorithm: KthSmallest($D, a, b, k$)
  Input: Let $D$ be a list of double numbers, the integers $a$ and $b$ respectively
         be the starting and ending indices of a sublist and $k$ be an integer
         refers to the $k^{th}$ smallest element.
  Output: The $k^{th}$ smallest element in $D$, a double number.
      if $a == b$ then
          return $D[i]$
      else
          $p$ = a random element in $D$;
          rearranging:
              all elements smaller than $p$ to the left of $p$;
              all elements larger than $p$ to the right of $p$;
          end;
          $c$ = the current index of $p$ in $D$;
          if $k == c - a + 1$ then
              return $p$
          else if $k < c - a + 1$ then
              return KthSmallest($D, a, c - 1, k$)
          else
              return KthSmallest($D, c + 1, b, k$)
          endif;
      endif;
```

Unlike the original QuickSort algorithm, the QuickSelect recurses only once and on one side after rearranging. This helps the algorithm remain $O(n)$ time complexity. By using the random selection, the pivot on average is close to the middle of the list. Therefore, with the input is an $n$-sized list $D$, the recursive call is on a sublist whose size is a half. Because the rearrangement takes $O(n)$ time, this algorithm takes at most $2n$ time which is $O(n)$. The running time is shown below:

$$O(n) + O(n/2) + O(n/4) + ... \leq O(2n) = O(n)$$

The following image is one of many outputs using different number of elements and $k$ value. I generated 100,000,000 numbers uniformly random in the range [-100000, 100000] with $k$ = 76,543,210.



```
Number of elements: 100,000,000
k: 76,543,210
Value range:  -100,000 to 100,000
- O(nlogn) Result Using Sort: 53081.606002
  Time taken: 53.615235s

- O(n) Result Using QuickSelect: 53081.606002
  Time taken: 3.588446s
```

Figure 1: The outputs of the $O(NlogN)$ algorithm using regular sorting and the $O(n)$ QuickSelect algorithm that recurses only once.

---

With the given number of elements, $k$ and value range, both of the algorithms produced the same result, which is 53081.606002, but the running times of them have a significant difference (For the program's output, see Figure 1).

Since the QuickSelect takes at most $2n$ running time, the regular sorting takes at least $log(n)/2$ times more than it, and the $log$ is in base 2. In this case, there are 100,000,000 numbers, so:

$$\frac{log(n)}{2} = \frac{log(100,000,000)}{2} \approx 13.2877 \text{ times} \leq \frac{53.615235}{3.588446} \approx 14.9411 \text{ times}$$

Since 14.9411 is quite close to 13.2877, the running time of this QuickSelect algorithm in practice is reasonable compared to the theory.

## 2.3 Probability to select a "good" point

Again, the goal of algorithm $\textsc{SepAnn}(S, n, d, \mu, c)$ is to find a "good" point $p$ in $S$. A "good" point $p$ implies that with $R_p$ (the radius of the smallest $ball_S(p, R_p)$ that contains at least $n/c$ points), the $ball_S(p, \mu R_p)$ contains at most $n/2$ points. Due to A. Maheshwari, W. Mulzer and M. Smid [1, Lemma 3], the algorithm has probability at least $1/c$ (about $1/1623.4$ in 2D space) to select a good point uniformly random from $S$.

The number of times the algorithm $\textsc{SepAnn}(S, n, d, \mu, c)$ repeats until it gets a good point depends on the way how the input points are generated. I will show you some ways I use to generate the points based on some patterns which produced different number of times the algorithm repeats.

The first way is **generating uniformly random** points spread throughout the space. This way of generating point produces a surprising behavior of this algorithm. Every time $\textsc{SepAnn}(S, n, d, \mu, c)$ is called, it only repeats *once* until it gets to the base case, that means the probability to get a good point is 100%, see algorithm's data on *GitHub repository*[1]. Since the points are generated randomly, so they change every time, there are no explanations about specific properties of the points in this case. However, since 100% satisfied the probability at least $1/c$, this result is reasonable.

Another way is **generating points evenly in a grid**. Let $n$ be the number of points, $a$ and $b$ be any two factors of $n$ that satisfies $a \times b = n$, $d$ be the distances between any two points which are close together. The points are set into the $a \times b$ grid which means each point is on a corner of a $d$-sided square, see Figure 2.

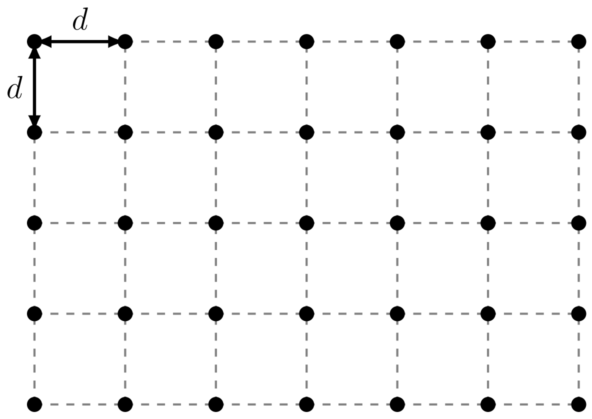


Figure 2: An example of generating points in grid. A set of 35 points that gives us a $5 \times 7$ grid. Points are placed at corners of the squares with side is $d$ which could be any number.

---

When using this grid way to generate points, the algorithm SepAnn$(S, n, d, \mu, c)$ now repeats multiple times which is its expected behavisee Figure 3. Again, the lowest probability in most of trials in this case is about 1/20 which satisfies probability at least $1/c$. For the full grid input data, see the *GitHub repository*[2].

| $n$ | **Repeat times** |
|---|---|
| 200000 | 6 |
| 193836 | 7 |
| 179534 | 2 |
| 172501 | 1 |
| 158381 | 13 |
| 156531 | 3 |
| ... | ... |

| $n$ | **Repeat times** |
|---|---|
| ... | ... |
| 101943 | 4 |
| 96367 | 2 |
| 96154 | 2 |
| 83621 | 6 |
| 83288 | 13 |
| 82515 | 2 |

Figure 3: Given an input of 200,000 points generated in a grid. This is a portion of the data about the number of times the algorithm SepAnn$(S, n, d, \mu, c)$ repeats.

[1]Random points SepAnn data. *https://github.com/ThangMinhCao/closestpairdoubling/blob/master/report/images%26_data/closest_pair/random_generation/random_sep_ann_data.txt*

[2]Grid points SepAnn data. *https://github.com/ThangMinhCao/closestpairdoubling/blob/master/report/images%26_data/closest_pair/grid_generation/grid_sepann_data.txt*

## 3 Algorithm 2: The refinement of $\textsc{SepAnn}(S, n, d, \mu, c)$

With the use of the algorithm $\textsc{SepAnn}(S, n, d, \mu, c)$'s output, this refinement algorithm continues to find an annulus that separates the points in $S$ into different smaller sets of points.

Let $\mu = e$ and $c = 2(4e)^d$, see [1, Remark 1]. The algorithm $\textsc{SepAnn}(S, n, d, e, c)$ returns an annulus's center $p \in S$ and its radius, $R' > 0$. An input of this refined algorithm is $t > 0$, a large enough constant calculated based on $n$. Let $R_i = (1+1/t)^i \cdot R'$, in which $i$ is a uniformly random element inclusively from 1 to $t$. The algorithm then finds an annulus centered at $p$ with the inner radius $R_{i-1}$ and outer radius $R_i$, denoted $A_i = annulus_S(p, R_{i-1}, R_i)$. If the annulus contains at most $n/t$ points then we are done. The algorithm returns $p$ and $R_{i-1}$. Otherwise, the previous procedure is repeated until the condition is satisfied. The pseudocode of this refined algorithm is given below, see [1, Section 3.2].

```
Algorithm 2: SparseSepAnn($S, n, d, t$)
Input: Let $S$ be a subset of $P$, of size $n \geq 2(4e)^d + 1$, $d$ be the space's doubling
       dimension, $t \geq 1$ be an integer.
Output: A point $p \in S$ and a radius $R > 0$.
    $c = 2(4e)^d$;
    let $p \in S$ and $R' > 0$ be the output of algorithm SepAnn($S, n, d, e, c$);
    repeat
        $i$ = a uniformly random element in $\{1, 2, \ldots, t\}$;
        $R_i = (1 + 1/t)^i \cdot R'$;
        $R_{i-1} = (1 + 1/t)^{i-1} \cdot R'$;
        $A_i = annulus_S(p, R_{i-1}, R_i)$;
        $s = |A_i|$;
    until $s \leq n/t$;
    $R = R_{i-1}$;
    return $p$ and $R$
```

*This pseudocode is from [1, Section 3.2]*

## 4 Algorithm 3: The main closest-pair algorithm

Using the result of $\textsc{SparseSepAnn}(S, n, d, t)$, this main algorithm $\textsc{ClosestPair}(S, n, d)$ recursively compute the closest distance.

Let $S$ be a $n$-sized subset of $P$, $d$ be the metric space $(P, dist)$'s doubling dimension. In the base case, that is when $n < 2(16e)^d$, the algorithm uses brute-force to compute the closest distance. If this is not the case, the algorithm sets $t = \lfloor \frac{1}{16e}(n/2)^{1/d} \rfloor$. This algorithm then runs $\textsc{SparseSepAnn}(\textsc{S,n,d,t})$ with this $t$ value, and get the output $p \in S$ and $R > 0$ which is the inner radius of the sparse annulus. The outer radius of it is $(1 + 1/t)R$. This algorithm now recursively calls itself on two subset of $S$ which are points contained in $ball_S(p, (1+1/t)R)$ and outside of $ball_S(p, R)$. Finally, the smaller distance from two recursive calls, $\delta_0$, is returned. Below is the algorithm's pseudocode, see [1, Section 4.1].

```
Algorithm 3: ClosestPair($S, n, d$)
  Input: Let $S$ be a subset of $P$, of size $n \geq 2$, $d$ be the space's doubling dimension.
  Output: A real number $\delta_0$ satisfies the two properties in [1, Lemma 5].
    if $n < 2(16e)^d$ then
        $\delta_0$ = the closest-pair distance in $S$ using brute-force;
    else
        $t = \lfloor \frac{1}{16e}(n/2)^{1/d} \rfloor$;
        let $p \in S$ and $R > 0$ be the output of algorithm SparseSepAnn($S, n, d, t$);
        $S_1 = ball_S(p, R)$;
        $S_2 = annulus_S(p, R, (1 + 1/t)R)$;
        $S_3 = S \setminus (S_1 \cup S_2)$;
        $n' = |S_1 \cup S2|$;
        $n'' = |S_2 \cup S3|$;
        $\delta' =$ ClosestPair($S_1 \cup S_2, n', d$);
        $\delta'' =$ ClosestPair($S_2 \cup S_3, n'', d$);
        $\delta_0 = min(\delta', \delta'')$;
    endif;
    return $\delta_0$
```

*This pseudocode is from [1, Section 4.1]*

## 5 The Implementation

This implementation of the closest-pair doubling algorithm of A. Maheshwari, W. Mulzer and M. Smid [1] is written in C++ since it is a very common and fast programming language with high-level supports of object-oriented programming that can help us organize the program efficiently (see the implementation's *GitHub repository*[3] for the original source code). The implementation and its components will be describe and explain throughout this section.

To begin with, a set of points, which does not necessarily refers to `set` in Computer Science, is always one of the input parameters of each partial algorithm. In this implementation, I would choose `vector`, a very popular built-in data structure of C++. Since its size can automatically change when we add or remove an element, it is easy to work with and manage its behavior when implementing. Not only for storing points, `vector` will be used throughout the implementation to store some other information.

### 5.1 The *Point* and *PointList* classes

Because the algorithm works around the points, and they are not just numbers (they have their own coordinates), so creating a `Point` class is quite essential. Moreover, with the addition of the class function `distance_to`, the algorithm can calculate the distance between any two points easily without directly using the points' coordinates. This function works as the input function *dist* that is provided with the metric space $(P, dist)$. Implementation code of the class is given below.

[3]GitHub repository of the implementation. *https://github.com/ThangMinhCao/closestpairdoubling*

```cpp
#include <vector>
#include <cmath>

class Point {
  private:
    std::vector<double> coordinate;

  public:
    explicit Point(std::vector<double> coordinate) {
      this->coordinate = std::move(coordinate);
    }
    Point()= default;

    double distance_to(const Point& another_point) {
      double total_square = 0;
      for (int i = 0; i < coordinate.size(); i++) {
        double iDiff = coordinate[i] - another_point.getCoordinate()[i];
        total_square += (iDiff) * (iDiff);
      }
      return sqrt(total_square);
    }
    std::vector<double> getCoordinate() const {
      return coordinate;
    }
}
```

## References

[1] A. Maheshwari, W. Mulzer and M. Smid. *A Simple Randomized $O(n \log n)$–Time Closest-Pair Algorithm in Doubling Metrics*, 2020. https://arxiv.org/abs/2004.05883