Project Report

An Implementation Of:

A Simple Randomized $O(n \log n)$–Time Closest-Pair
Algorithm in Doubling Metrics

Minh Thang Cao

23 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 δ_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 $\textsc{SepAnn}(S, n, d, \mu, c)$

2. The refinement of $\textsc{SepAnn}(S, n, d, \mu, c)$, denoted $\textsc{SparseSepAnn}(S, n, d, t)$

3. The main recursive closest-pair algorithm, denoted $\textsc{ClosestPair}(S, n, d)$

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

Note: I will only mention 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 $\log_2 21$ 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, μ, c) algorithm

In this section, $\mu \geq 1$ is a real constant number, c is calculated based on μ (I would say that $c = 2(4\mu)^d$ [1, Remark 1] since μ 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 Kth smallest element

One step in SEPANN(S, n, d, μ, c) algorithm is to find the smallest ball which contains at least n/c points. This ball is easy to find using the kth 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 kth smallest element in a list is to sort it in ascending order, and then simply return the element at the kth 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 kth 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, $D[c + 1 : b]$, with a different input k (which is $k - c + a - 1$) since this is the k of the subset to the right, not the whole set of points, and the elements' indices starts from 0 again when we recurse. 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 kth smallest element.

Output: The kth 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$(D, c + 1, b, k - c + a - 1)$
        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 numbers of elements and k values. I generated $100\,000\,000$ numbers uniformly random in the range [-100000, 100000] with $k = 76{,}543{,}210$.

Figure 1: The outputs of the $O(n \log n)$ 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.

3   Algorithm 2: The refinement of $\text{SEPANN}(S, n, d, \mu, c)$

With the use of the algorithm $\text{SEPANN}(S, n, d, \mu, c)$'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 $\text{SEPANN}(S, n, d, e, c)$ 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}. This algorithm has the expected $O(cn)$ time complexity, see [1, Section 3.2]. Otherwise, the previous procedure is repeated until the condition is satisfied. The pseudocode of this refined algorithm is given below.

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 $R > 0$.

$\quad c = 2(4e)^d$;

$\quad$ let $p \in S$ and $R' > 0$ be the output of algorithm SEPANN(S, n, d, e, c);

$\quad$ **repeat**

$\quad\quad\quad\quad i = $ a uniformly random element in $\{1, 2, \dots, t\}$;

$\quad\quad\quad\quad R_i = (1 + 1/t)^i \cdot R'$;

$\quad\quad\quad\quad R_{i-1} = (1 + 1/t)^{i-1} \cdot R'$;

$\quad\quad\quad\quad A_i = annulus_S(p, R_{i-1}, R_i)$;

$\quad\quad\quad\quad s = |A_i|$;

$\quad$ **until** $s \leq n/t$;

$\quad R = R_{i-1}$;

$\quad$ return p and R
---

4 Algorithm 3: The main closest-pair algorithm

Using the result of SPARSESEPANN(S, n, d, t), this main algorithm CLOSESTPAIR(S, n, d) recursively compute the closest distance.

Let S be an 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 SPARSESEPANN(S, n, d, t) with this t value, and gets the output $p \in S$ and $R > 0$ which is the inner radius of the sparse annulus. The outer radius of it is $(1 + 1/t)R$.

Let S_1 be the set of points contained in $ball_S(p, R)$, S_2 be the set of points contained in $annulus_S(p, R, (1 + 1/t)R$ and S_3 be the set that contains all the points outside of the outer radius $(1 + 1/t)R$. This algorithm now recursively calls itself on two subsets of S which are points contained in $ball_S(p, (1 + 1/t)R)$ (which is $S1 \cup S2$) and outside of $ball_S(p, R)$ (which is $S2 \cup S3$). Finally, the smaller distance output from two recursive calls, $\delta_0 = min(\delta_1, \delta_2)$, 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 δ_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 δ_0

This pseudocode is from [1, Section 4.1]

5 The Implementation

This implementation of the closest-pair doubling algorithm 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*[1] for the source code). The implementation and its components will be described and explain throughout this section. Also, the code and description of each function will be given below.

To begin with, a set of points, which does not necessarily refer 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 Utility functions and classes

5.1.1 Random number generator classes

Due to A. Maheshwari, W. Mulzer and M. Smid, the closest-pair doubling algorithm uses randomized technique in some parts to find the targets in linear time. Therefore, generating

---

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

random numbers is important. I created two random number generator classes, one for integer and one for double, to make the program more simple. The generators' code is given below.

```cpp
#include <random>

class RandomInt{
  public:
      RandomInt(int start, int end)
            : generator{std::random_device{}()}, int_dist(start, end) {}
      int next() { return int_dist(generator); }

  private:
      std::mt19937 generator;
      std::uniform_int_distribution<> int_dist;
};

class RandomDouble{
  public:
      RandomDouble(int start, int end)
            : generator{std::random_device{}()}, double_dist(start, end) {}
      double next() { return double_dist(generator); }

  private:
      std::mt19937 generator;
      std::uniform_real_distribution<> double_dist;
};
```

Source [2]: master branch → closestpairdoubling/include/RandomGenerator.h

For each class, I create a uniform distribution with a given range that corresponds to the number type and an `mt19937` random generator with a `random_device` input. These two variables will be initialized when a random generator object is instantiated. Each class also has a `next()` function that can be called to return a random number.

5.1.2 *Point* and *PointList* classes

Since points are fundamental elements of the algorithm, and they are not just numbers (they have their own coordinates), 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 $(P, dist)$. The implementation code of the class is given below.

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

class Point {
  private:
```

---

[2]Path to the source file in the implementation's GitHub repository.

```
6      std::vector<double> coordinate;
7
8    public:
9      explicit Point(std::vector<double> coordinate) {
10       this->coordinate = std::move(coordinate);
11     }
12     Point()= default;
13     // distance is calculated using Pythagorean Theorem
14     double distance_to(const Point& another_point) {
15       double total_square = 0;
16       for (int i = 0; i < coordinate.size(); i++) {
17         double iDiff = coordinate[i] - another_point.getCoordinate()[i];
18         total_square += (iDiff) * (iDiff);
19       }
20       return sqrt(total_square);
21     }
22     std::vector<double> getCoordinate() const {
23       return coordinate;
24     }
25 }
```

Source: master branch → closestpairdoubling/include/Point.h

Because there is not only one method to generate points (see the description later in Section 6), so if all of them are implemented in the `main()`, the function will be quite huge and complicated. Thus, I created another utility class to make the code more organized, `PointList`. The `PointList` class consists of a `vector` named *points* that store all the points in a subset of P and some point generating functions. This class is used as subsets of P and to initialize all the points in P at the start of the program.

```
1  #include "Point.h"
2  #include "RandomGenerator.h"
3
4  class PointList {
5    public:
6      std::vector<Point> points;
7
8      // initializes all points uniformly random, see Section 6 for more details
9      void random_initializer(int dimension, int point_num,
10                             int coor_start_range, int coor_end_range)
11     {
12       RandomDouble random_double(coor_start_range, coor_end_range);
13       for (int i = 0; i < point_num; i++) {
14         std::vector<double> coor;
15         // number of coordinates is the dimension value
16         for (int j = 0; j < dimension; j++) {
17           coor.push_back(random_double.next());
18         }
19         points.push_back(Point(coor));
20       }
21     }
22
```

8

```
23    // initializes points in grid, see Section 6 for more details
24    void grid_initializer(double dist, int xFactor, int yFactor, int position)
25    {
26      // the grid starts at coordinates (position, position)
27      int y = 0;
28      while (y < yFactor) {
29        int x = 0;
30        while (x < xFactor) {
31          std::vector<double> coor;
32          coor.push_back(position + (x++) * dist);
33          coor.push_back(position + y * dist);
34          points.push_back(Point(coor));
35        }
36        y++;
37      }
38    }
39 }
```

Source: master branch → closestpairdoubling/include/PointList.h

5.1.3 K^{th} smallest element finder

The Kth smallest algorithm has its own class. In this class, I define a shorter type name `DVect` which is `std::vector<double>`. The QuickSelect algorithm is divided into three functions, the main is `get`, and two assistances `swap` and `partition`. The function `get` returns the kth smallest element using QuickSelect, and the other using the normal sort. Although I only use the QuickSelect technique, I create both to compare their time complexity, see Section 2.2.

```
1  #include <vector>
2
3  typedef std::vector<double> DVect;
4
5  class KthSmallest {
6    private:
7      static void swap (double *a, double *b);
8      static int partition(DVect& distances, int start, int end);
9
10   public:
11     // get k-th smallest element with QuickSelect
12     static double get(DVect& distances, int start, int end, int k);
13     // get k-th smallest element with normal sorting
14     static double get_with_sorting(DVect distances, int k);
15 };
```

Source: master branch → closestpairdoubling/include/KthSmallest.h

Due to its simplicity and irrelevant uses (only for testing and analyzing), the implementation of the normal sorting technique will not be listed in this report, see the *GitHub repository* for the code. The following is the implementation code of the QuickSelect algorithm.

```cpp
1  #include "KthSmallest.h"
2  #include "RandomGenerator.h"
3
4  void KthSmallest::swap (double *a, double *b) {
5    double temp = *a;
6    *a = *b;
7    *b = temp;
8  }
9
10 int KthSmallest::partition(DVect& distances, int start, int end) {
11   double pivot = distances[end];
12   int slow = start;
13   int fast = start;
14   // moving the smaller elements to the left and larger elements to the right
15   while (fast < end) {
16     if (distances[slow] > pivot) {
17       while (fast < end and distances[fast] > pivot) {
18         fast++;
19       }
20       if (distances[fast] < pivot) {
21         swap(&distances[slow], &distances[fast]);
22       } else {
23         break;
24       }
25     }
26     fast++;
27     slow++;
28   }
29   // swapping the pivot from end back to its original place
30   swap(&distances[slow], &distances[end]);
31   return slow;
32 }
33
34 double KthSmallest::get(DVect& distances, int start, int end, int k) {
35   if (start == end) {return distances[start];}
36   //  swapping the random pivot to the end
37   RandomInt random_int_gen = RandomInt(start, end);
38   swap(&distances[random_int_gen.next()], &distances[end]);
39   int cur_pivot = partition(distances, start, end);
40   if (k == cur_pivot - start + 1) {
41     return distances[cur_pivot];
42   } else if (k < cur_pivot - start + 1) {
43     return get(distances, start, cur_pivot - 1, k);
44   } else {
45     return get(distances, cur_pivot + 1, end, k - cur_pivot + start - 1);
46   }
47 }
```

Source: master branch → closestpairdoubling/src/KthSmallest.cpp

The function `swap(double *a, double *b)` swaps two elements in a vector by swaps the values at the addresses that the pointers `*a` and `*b` point to. Thus, their values in a vector are swapped while their identities are still the same.

The function `partition(DVect& distances, int start, int end)` takes the last element in `distances` to be the pivot. Then, it rearranges the vector so that every element smaller and larger than pivot is to the left and right of it. The technique I use in this function is creating two pointers `fast` and `slow`. Both pointers move towards the end step by step together until `distances[slow] > pivot`, then it stops. The `fast` keep moving until `distances[fast] < pivot`, then the value at two indices are swapped. This procedure is repeated until the `fast` reach the end of the vector The `pivot` is now swapped back to the `slow` position, where it is supposed to be.

A generic call is `get(DVect& distances, int start, int end, int k)`. It first checks if the sub-vector has only one element (as well as `start == end`) then returns it. Otherwise, it selects a random pivot in the vector then swaps it with the last element. Now, the `partition` function is called to return the index of the pivot after rearranging. The base case is when there are exactly k elements from index 0 to the current pivot in the original vector (which is `cur_pivot - start + 1`), so it is the kth smallest element. If there are more than k elements, the function is recursively called on the sub-vector to the left of the pivot. If there are less than k elements, the function is called on the sub-vector to the right, but now the k should be decreased by `cur_pivot - start + 1`. Due to Section 2.2, this procedure takes $O(n)$ time complexity.

5.1.4 The 2D algorithm for testing

In the project, I also implement the 2D closest-pair algorithm using Divide-and-conquer algorithm due to Bentley and Shamos, see [5]. This algorithm is very helpful in testing and debugging the closest-pair doubling algorithm because of its short running time. I will briefly describe the algorithm below.

Given a set of point P, the base case is when P contains less than 2 points. If there is only one, the algorithm returns positive ∞. If there are two points, the algorithm basically returns the distance between them. Otherwise, P is sorted in ascending order based on the points' x-coordinate. Then, it chooses a vertical line, ℓ, that divides the set P into two halves, P_1 contains $\lfloor n/2 \rfloor$ points and P_2 contains $\lceil n/2 \rceil$ points. The algorithm now recursively calls itself on P_1 and P_2, so the minimum of the two outputs would be the temporary closest-pair distance, $\delta_0 = min(\delta_1, \delta_2)$. It is obvious that there could be a pair of points (one from P_1 and the other from P_2) which has a closer distance than the temporary δ_0.

To combine the two set P_1 and P_2, we create a strip centered at ℓ with its left end and right end are the lines ℓ_1 and ℓ_2 so that both lines are δ_0 away from ℓ. All the points in the strip are now sorted based on their y-coordinate. The algorithm then can loop through the strip and check if there is any smaller distance than δ_0. Fortunately, for each point in the strip, the algorithm just needs to check the distances between the current point with its 7 successive points, that have y-coordinates $\geq$ the current point's y-coordinate. If there is any closer distance, it is assigned to δ_0. Finally, δ_0 is returned.

To keep the report on point and less complicated, the code of this 2D algorithm will not be listed. See the implementation on my *GitHub repository*[3].

---

[3]The implementation of the 2D algorithm with merge sort algorithm. *https://github.com/ThangMinh-Cao/closestpairdoubling/tree/master/utils*

5.2 The closest-pair doubling algorithm

The implementation of the closest-pair algorithm still consists of three main functions, `sep_ann`, `sparse_sep_ann` and `closest_pair`. Moreover, the `brute_force` function is added to be used in the base case. The following is the algorithm's class.

```cpp
#include "PointList.h"

typedef std::vector<double> DVect;

class ClosestPairDoubling {
  private:
    static std::tuple<Point, double, DVect> sep_ann(PointList &S, int n,
        double mu, double c);
    static std::pair<Point, double> sparse_sep_ann(PointList &S, int n, double
        d, int t);
  public:
    ClosestPairDoubling() = default;
    static double brute_force(PointList &S);
    static double closest_pair(PointList &S, double d, int recursion=0);
};
```

Source: master branch → closestpairdoubling/include/ClosestPairDoubling.h

5.2.1 The SEPANN(S, n, d, μ, c) algorithm

The `sep_ann` function starts by defining some important variables to be used in its loop. Let `p` be the chosen random point in S, `Rp` be the radius of the smallest ball containing at least n/c points, `distances_from_p` be the vector stores the distances between p and all points in S and `outer_ball_count` be the number of points in the outer ball, $ball_S(p, \mu R_p)$. To select a random point p, I initialize an integer random generator `int_gen` with the range from 0 to $n-1$, possible indices of points in S.

Since c is calculated based on d which could be done outside the function, I removed the input d of the `sep_ann` function.

```cpp
std::tuple<Point, double, DVect>
    ClosestPairDoubling::sep_ann(PointList &S, int n, double mu, double c)
{
  Point p;
  double Rp = -1.0;
  DVect distances_from_p;
  int outer_ball_count = (n / 2) + 1;
  RandomInt int_gen(0, n - 1);

  while (Rp == -1.0 or outer_ball_count > std::floor(n / 2)) {
    distances_from_p.clear();

    // selecting the random point p and calculate all the distances
    int random_index = int_gen.next();
    p = S.points[random_index];
```

```
16      for (const Point& point: S.points) {
17        distances_from_p.push_back(p.distance_to(point));
18      }
19      Rp = KthSmallest::get(distances_from_p, 0, (int)distances_from_p.size(),
           ceil(n / c));
20
21      // counting the number of point in the outer ball
22      outer_ball_count = 0;
23      for (double dist: distances_from_p) {
24        if (dist <= mu * Rp) {
25          outer_ball_count++;
26        }
27      }
28    }
29    return std::make_tuple(p, Rp, distances_from_p);
30 }
```

Source: master branch → closestpairdoubling/src/ClosestPairDoubling.cpp

Each time the *while* loop is repeated, `distances_from_p` is cleared to get rid of the previous loop's result. The function uses `int_gen` to generate a random point in S, then it stores distances between p and every point in S including itself into `distances_from_p`. The QuickSelect function from `KthSmallest` class is now used to get the *ceil*(n/c)-smallest element in `distances_from_p`, then it is assigned to `Rp`. The function loops through the distance vector to count the number of points inside the outer ball and assigns it to `outer_ball_count`. If `outer_ball_count` $< \lfloor n/2 \rfloor$ then we are done, the function returns a tuple containing `p`, `Rp` and `distances_from_p`. If this is not the case, the procedure is repeated until the condition is satisfied. Note that, I return `distances_from_p` here to use it in SPARSESEPANN without spending time to loop through the vector again.

5.2.2 The refined algorithm SPARSESEPANN(S, n, d, t)

Let `sep_ann_res` be the output of the function `sep_ann(S, n, e, c)`, `Ai_size` be the number of points $annulus_S(p, R_{i-1}, R_i)$ contains. Let `p`, `R_prime` and `distance_from_p` be the variables store the values which are derived from `sep_ann_res`. The implementation code of SPARSESEPANN is given below.

```
1 std::pair<Point, double>
2   ClosestPairDoubling::sparse_sep_ann(PointList &S, int n, double d, int t)
3 {
4   const double e = std::exp(1.0); // the Euler constant
5   double c = 2 * pow(4 * e, d);
6   std::tuple<Point, double, DVect> sep_ann_res = sep_ann(S, n, e, c);
7
8   // followings are results derived from sep_ann_res
9   Point p = std::get<0>(sep_ann_res);
10  double R_prime = std::get<1>(sep_ann_res);
11  DVect distances_from_p = std::get<2>(sep_ann_res);
```

```
12
13    int Ai_size = n / t + 1;
14    RandomInt range_t_random(1, t);
15
16    double R;
17    while (Ai_size > n / t) {
18      int random_i = range_t_random.next();
19      double Ri = pow(1 + 1.0 / t, random_i) * R_prime;
20      R = pow(1 + 1.0 / t, random_i - 1) * R_prime;
21      Ai_size = 0;
22      for (double dist: distances_from_p) {
23        if (R <= dist and dist <= Ri) {
24          Ai_size++;
25        }
26      }
27    }
28    return std::pair<Point, double> {p, R};
29 }
```

Source: master branch → closestpairdoubling/src/ClosestPairDoubling.cpp

The algorithm first selects a random value i in the range from 1 to t using the random generator `range_t_random`, then it calculates the inner and outer radius of $annulus_S(p, R_{i-1}, R_i)$ in which $R_i = (1 + 1/t)^i$. An inner `for` loop goes through `distance_from_p` to count the number of points in the annulus and then stores the number to `Ai_size`. If `Ai_size` $> n/t$, the right annulus is found, then it returns `p` and `R`. Otherwise, the `while` loop repeats the procedure to find the annulus that satisfies `Ai_size` $\leq n/t$.

5.2.3 The main closest-pair algorithm CLOSESTPAIR(S, n, d)

Let `n` be the number of points contained in S, `e` be the Euler constant, `delta0` be the closest-pair distance in S. Furthermore, I combine the three subsets S_1, S_2 and S_3 (see Section 4) to only two that are `S1orS2` and `S2orS3` that represent the subset $S_1 \cup S_2$ and $S_2 \cup S_3$.

```
1  double
2      ClosestPairDoubling::closest_pair(PointList &S, double d, int recursion)
3  {
4    int n = (int)S.points.size();
5    const double e = std::exp(1.0);
6    double delta0; // the closest-pair distance in S
7
8    if (n < 2 * pow(16 * e, d)) {
9      return brute_force(S);
10   } else {
11     int t = floor((1 / (16 * std::exp(1.0))) * pow((double)n / 2, 1 / d));
12     std::pair<Point, double> ssann_result = sparse_sep_ann(S, n, d, t);
13     Point p = ssann_result.first;
14     double R = ssann_result.second;
```

14

```
15    PointList S1orS2 = PointList();
16    PointList S2orS3 = PointList();
17    for (const Point& point: S.points) {
18      double d_to_p = p.distance_to(point);
19      if (d_to_p <= R) {
20        S1orS2.points.push_back(point);
21      } else if (d_to_p > R and d_to_p <= (1 + 1.0/t) * R) {
22        S1orS2.points.push_back(point);
23        S2orS3.points.push_back(point);
24      } else {
25        S2orS3.points.push_back(point);
26      }
27    }
28    std::vector<Point>().swap(S.points); // free the memory that stores S
29    double delta1 = closest_pair(S1orS2, d, ++recursion);
30    double delta2 = closest_pair(S2orS3, d, ++recursion);
31    delta0 = std::min(delta1, delta2);
32  }
33  return delta0;
34 }
```

Source: master branch → closestpairdoubling/src/ClosestPairDoubling.cpp

With $t = \lfloor \frac{1}{16e}(n/2)^{1/d} \rfloor$, `sparse_sep_ann(S,b,d,t)` is called, and the outputs of it are assigned into `p` and `R`. The `for` loop goes through S to build the two sets by adding corresponding points to them using the distance from p to each point. Let `d_to_p` be the distance from p to the current point in the `for` loop. If `d_to_p` $\leq R$, the current point is in the $ball_S(p, R)$, so it is added to `S1orS2`. If $R <$ `d_to_p` $\leq (1+1/t)R$, the point is in the $annulus_S(p, R, (1+1/t)R)$, then it is added to both `S1orS2` and `S2orS3`. Otherwise, the point is appended into only `S2orS3` since it is outside the outer radius.

Because every point in S is now in `S1orS2` and `S2orS3`, it is not necessary to keep the vector `S` as well as its memory usage, which could be stacked up after many recursive calls and cause memory overload. Thus, after storing all the points into `S1orS2` and `S2orS3`, I clear the vector `S` and its data in the memory using the function `vector::swap()` to swap `S` with an empty vector. Note that, I will make sure to pass the full set of points **by value** or pass a copy of it to the first call of the algorithm, so the original one will not be changed. Therefore, it can not affect the execution of `brute_force` for testing later.

After that, the outputs of two recursive calls, one on `S1orS2` and one on `S2orS3`, will be stored in `delta1` and `delta2` variables. Finally, the smaller one of them will be assigned into `delta0` and returned.

5.3 The closest-pair distance of non-Euclidean spaces

The algorithm runs correctly in a special case in which the given metric space is non-Euclidean of doubling dimension 1, see [3, Section 2.1] for more details. Given a metric space $S = \{p_1, p_2, \ldots, p_n\}$ that contains n "points", denoted p_m with $1 \leq m \leq n$. Specifically, the value of p_m represents an integer between a given range. Let i and j be any number that satisfies $1 \leq i \leq j \leq n$, the distance between p_i and p_j is defined:

15

$$dist(p_i, p_j) = \begin{cases} 0 & \text{if } i = j \\ 4^{max(i,j)} & \text{if } i \neq j \end{cases}$$

Using the definition, the closest-pair algorithm finds the closest distance in this metric space normally without changing any core detail. The only problem is that the distance between any two points could be extremely large.

For instance, let the upper bound of the points' value be a big number, say 100, so $p_i \leq p_j \leq 100$. Let $p_i = 100$, then $dist(p_i, p_j) = 4^{100} > 1.6 \times 10^{60}$ which is much larger than the maximum 32-bit integer that C++ can handle. Therefore, I decided to use a library named *Boost*[4] which can handle numbers with precisions up to 1024 bits. When changing the current algorithm to use *Boost*, I only need to replace the `double` type of normal distances by `cpp_int` type which can automatically choose the corresponding precision for each distance. Also, I needed to make some minor changes, so the algorithm can be able to run correctly. Since there are not many changes and they do not affect the main theory of the algorithm, the implementation for this non-Euclidean will not be listed in this report. See the `strange_metric` branch[5] of my GitHub repository for the implementation.

6 The practical running time and behavior

After doing many tests, I found out that the algorithm's behavior will be different based on the way the points are arranged on the metric space. There are two approaches to generate the points that give me the most obvious differences.

1. **Generating uniformly random** points spread throughout the space. To do this, we can just use any of the random number generators to get the points' random coordinates in a given range, see Section 5.1.1 and Section 5.1.2 for more details.

2. **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. Start at any position, 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.
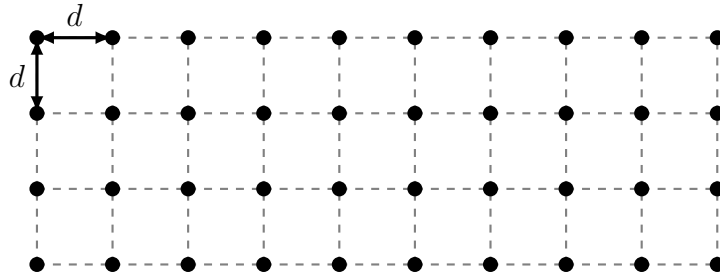

Figure 2: An example of points in a grid. A set of 40 points that gives us a 4×10 grid. Points are placed at corners of the squares whose sides are equal to d that could be any number.

---

[4]The *Boost* library's website. *https://www.boost.org/*

[5]The implementation of the algorithm on non-Euclidean metrics. *https://github.com/ThangMinhCao/closestpairdoubling/tree/strange_metric*

6.2 The practical running time

To have the running time that is close to the theory, the number of points in P must be extremely large since the base case of the algorithm is when $n < 2(16e)^d$, which is already a quite large number (approximately $79\,550$ points in 2D and more than $3\,000\,000$ points in 3D, for example). To correctly compare the algorithm's running time, we can use brute-force which takes $O(n)$ time or the 2D algorithm for only 2D points.

6.3 Probability to select a "good" point of SEPANN(S, n, d, μ, c)

Again, the goal of algorithm SEPANN(S, n, d, μ, 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. Like the practical running time, the number of times the algorithm SEPANN(S, n, d, μ, c) repeats until it gets a good point also depends on the way how the input points are generated.

Generating point randomly produces a surprising behavior of this algorithm. Every time SEPANN(S, n, d, μ, 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*[6]. 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.

On the other hand, when using the grid way to generate points, the algorithm SEPANN(S, n, d, μ, c) now repeats multiple times which is its expected behavior, see an example output in Figure 3. Again, the lowest probability in most of the 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*[7].

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, μ, c) repeats.

---

[6]Random points SepAnn data. *https://github.com/ThangMinhCao/closestpairdoubling/blob/master/rep ort/images_%26_data/closest_pair/random_generation/random_sep_ann_data.txt*
[7]Grid points SepAnn data. *https://github.com/ThangMinhCao/closestpairdoubling/blob/master/report/ images_%26_data/closest_pair/grid_generation/grid_sepann_data.txt*

7 Concluding remarks

I have briefly described the theory of the closest-pair doubling algorithm, and the implementation of it. Even we do not have the ability to run the algorithm with a huge number of points for more accurate running time outputs, the algorithm works exactly as we expected in practice:

1. For each example with different number of points in P, the closest-pair distance is correctly returned (the same as the brute-force's) without the use of the points' coordinates. Its running time is obviously much less than the result from brute-force.
2. For any way that the points are arranged on the metric space, the number of times the algorithm $\textsc{SepAnn}(S, n, d, \mu, c)$ repeats satisfies the probability $1/c$ in theory. Therefore, in practice, this procedure runs in linear time.
3. The algorithm can correctly find the closest-pair distances on non-Euclidean spaces of doubling dimension 1 without changing any core details. Also, its behavior is mostly the same as when the input is a normal space.

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

[2] J. L. Bentley and M. I. Shamos. Divide-and-conquer in multidimensional space. In *Proceedings of the 8th ACM Symposium on the Theory of Computing*, pages 220–230, 1976.

[3] M. Smid. *The Weak Gap Property in Metric Spaces of Bounded Doubling Dimension.* https://people.scs.carleton.ca/~michiel/weakgap.pdf