Project Report

An Implementation Of:

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

Minh Thang Cao

06 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].

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

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 $\textsc{SepAnn}(S, n, d, \mu, c)$ 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. I will rewrite this algorithm's pseudocode below, from [1, Section 3.1]:

Algorithm 1: $\textsc{SepAnn}(S, n, d, \mu, c)$

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;

2.2 Finding the K^{th} 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 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.

```
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: 53086.024078
  Time taken: 59.389754s

- O(n) Result Using QuickSort Technique: 53086.024078
  Time taken: 3.889757s
```

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 53086.024078, but the running times of them have a significant difference (For the output capture, 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:

$$\frac{log(n)}{2} = \frac{log(100,000,000)}{2} \approx 13.2877 \text{ times} \leq \frac{59.389754}{3.889757} \approx 15.268 \text{ times}$$

Since 15.268 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 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 $1/c$ to select a good point uniformly random from S. This declaration is clear and true in practice.

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. Furthermore, to have a clearer and more accurate observation, the number of input points should be very large, which is much larger than the base case. Thus, all the practical data such as probability and time complexity would be closer to their theoretical ones. For this reason, it is very difficult to generate that number of points manually, so I will show you some ways I generated 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].

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 reducing the number of points in the output ball.

Let $\mu = e$ and $c = 2(4e)^d$, see [1, Remark 1]. The algorithm $\textsc{SepAnn}(S, n, d, e, c)$ returns the output 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}. If this is not the case, the previous procedure is repeated until the condition is satisfied.

4 Algorithm 3: The main closest-pair algorithm

5 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*[2] for the source code).

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

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

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