Chapter 3

Complexity, the Changing Minimum and Closest Pair

By Sariel Har-Peled, May 29, 2013[①]

> The events of 8 September prompted Foch to draft the later legendary signal: "My centre is giving way, my right is in retreat, situation excellent. I attack." It was probably never sent.
> – John Keegan, The first world war.

3.1 Las Vegas and Monte Carlo algorithms

Definition 3.1.1. A ***Las Vegas algorithm*** is a randomized algorithms that *always* return the correct result. The only variant is that it's running time might change between executions.

An example for a Las Vegas algorithm is the **QuickSort** algorithm.

Definition 3.1.2. A ***Monte Carlo algorithm*** is a randomized algorithm that might output an incorrect result. However, the probability of error can be diminished by repeated executions of the algorithm.

The **MinCut** algorithm was an example of a Monte Carlo algorithm.

3.1.1 Complexity Classes

I assume people know what are Turing machines, **NP**, **NPC**, RAM machines, uniform model, logarithmic model. **PSPACE**, and **EXP**. If you do now know what are those things, you should read about them. Some of that is covered in the randomized algorithms book, and some other stuff is covered in any basic text on complexity theory.

Definition 3.1.3. The class **P** consists of all languages L that have a polynomial time algorithm **Alg**, such that for any input Σ^*, we have

- $x \in L \Rightarrow$ **Alg**(x) accepts,
- $x \notin L \Rightarrow$ **Alg**(x) rejects.

Definition 3.1.4. The class **NP** consists of all languages L that have a polynomial time algorithm **Alg**, such that for any input Σ^*, we have:

(i) If $x \in L \Rightarrow$ then $\exists y \in \Sigma^*$, **Alg**(x, y) accepts, where $|y|$ (i.e. the length of y) is bounded by a polynomial in $|x|$.

(ii) If $x \notin L \Rightarrow$ then $\forall y \in \Sigma^*$**Alg**(x, y) rejects.

Definition 3.1.5. For a complexity class $\mathcal{C}$, we define the complementary class co-$\mathcal{C}$ as the set of languages whose complement is in the class $\mathcal{C}$. That is

$$co - \mathcal{C} = \left\{ L \;\middle|\; \overline{L} \in \mathcal{C} \right\},$$

where $\overline{L} = \Sigma^* \setminus L$.

It is obvious that **P** = co−**P** and **P** $\subseteq$ **NP** $\cap$ co−**NP**. (It is currently unknown if **P** = **NP** $\cap$ co−**NP** or whether **NP** = co−**NP**, although both statements are believed to be false.)

Definition 3.1.6. The class **RP** (for Randomized Polynomial time) consists of all languages L that have a randomized algorithm **Alg** with worst case polynomial running time such that for any input $x \in \Sigma^*$, we have

(i) If $x \in L$ then **Pr**[**Alg**(x) accepts] $\geq 1/2$.

(ii) $x \notin L$ then **Pr**[**Alg**(x) accepts] $= 0$.

An **RP** algorithm is a Monte Carlo algorithm, but this algorithm can make a mistake only if $x \in L$. As such, co−**RP** is all the languages that have a Monte Carlo algorithm that make a mistake only if $x \notin L$. A problem which is in **RP** $\cap$ co−**RP** has an algorithm that does not make a mistake, namely a Las Vegas algorithm.

Definition 3.1.7. The class **ZPP** (for Zero-error Probabilistic Polynomial time) is the class of languages that have Las Vegas algorithms in expected polynomial time.

Definition 3.1.8. The class **PP** (for Probabilistic Polynomial time) is the class of languages that have a randomized algorithm **Alg** with worst case polynomial running time such that for any input $x \in \Sigma^*$, we have

(i) If $x \in L$ then **Pr**$\Big[$**Alg**(x) accepts$\Big] > 1/2$.

(ii) If $x \notin L$ then **Pr**$\Big[$**Alg**(x) accepts$\Big] < 1/2$.

The class **PP** is not very useful. Why?

Well, lets think about it. A randomized algorithm that just return yes/no with probability half is almost in **PP**, as it return the correct answer with probability half. An algorithm is in **PP** needs to be slightly better, and be correct with probability better than half, but how much better can be made to be arbitrarily close to $1/2$. In particular, there is no way to do effective amplification with such an algorithm.

Definition 3.1.9. The class **BPP** (for Bounded-error Probabilistic Polynomial time) is the class of languages that have a randomized algorithm **Alg** with worst case polynomial running time such that for any input $x \in \Sigma^*$, we have

(i) If $x \in L$ then $\mathbf{Pr}[\mathbf{Alg}(x) \text{ accepts}] \geq 3/4$.

(ii) If $x \notin L$ then $\mathbf{Pr}[\mathbf{Alg}(x) \text{ accepts}] \leq 1/4$.

3.2 How many times can a minimum change, before it is THE minimum?

Let $a_1, \ldots, a_n$ be a set of n numbers, and let us randomly permute them into the sequence $b_1, \ldots, b_n$. Next, let $c_i = \min_{k=1}^{i} b_i$, and let X be the random variable which is the number of distinct values that appears in the sequence $c_1, \ldots, c_n$. What is the expectation of X?

Lemma 3.2.1. *In expectation, the number of times the minimum of a prefix of n randomly permuted numbers change, is $O(\log n)$. That is $\mathbf{E}[X] = O(\log n)$.*

Proof: Consider the indicator variable X_i, such that $X_i = 1$ if $c_i \neq c_{i-1}$. The probability for that is $\leq q1/i$, since this is the probability that the smallest number if $b_1, \ldots, b_i$ is b_i. As such, we have $X = \sum_i X_i$, and $\mathbf{E}[X] = \sum_i \mathbf{E}[X_i] = \sum_{i=1}^{n} \frac{1}{i} = O(\log n)$. ∎

3.3 Closest Pair

Assumption 3.3.1. Throughout the discourse, we are going to assume that every hashing operation takes (worst case) constant time. This is quite a reasonable assumption when true randomness is available (using for example perfect hashing [CLRS01]). We probably will revisit this issue later in the course.

For a real positive number r and a point $\mathsf{p} = (x, y)$ in $\mathbb{R}^2$, define

$$\mathrm{G}_r(\mathsf{p}) := \left(\left\lfloor \frac{x}{r} \right\rfloor r \ , \ \left\lfloor \frac{y}{r} \right\rfloor r \right) \in \mathbb{R}^2.$$

We call r the ***width*** of the ***grid*** G_r. Observe that G_r partitions the plane into square regions, which we call grid ***cells***. Formally, for any $i, j \in \mathbb{Z}$, the intersection of the half-planes $x \geq ri$, $x < r(i+1)$,

$y \geq rj$ and $y < r(j+1)$ is said to be a ***grid cell***. Further we define a ***grid cluster*** as a block of 3×3 contiguous grid cells.

For a point set P, and a parameter r, the partition of P into subsets by the grid G_r, is denoted by $\mathrm{G}_r(\mathsf{P})$. More formally, two points $\mathsf{p}, \mathsf{q} \in \mathsf{P}$ belong to the same set in the partition $\mathrm{G}_r(\mathsf{P})$, if both points are being mapped to the same grid point or equivalently belong to the same grid cell.

Note, that every grid cell C of G_r, has a unique ID; indeed, let $\mathsf{p} = (x, y)$ be any point in C, and consider the pair of integer numbers $\mathrm{id}_C = \mathrm{id}(\mathsf{p}) = (\lfloor x/r \rfloor, \lfloor y/r \rfloor)$. Clearly, only points inside C are going to be mapped to id_C. This is very useful, since we can store a set P of points inside a grid efficiently. Indeed, given a point p, compute its $\mathrm{id}(\mathsf{p})$. We associate with each unique id a data-structure that stores all the points falling into this grid cell (of course, we do not maintain such data-structures for grid cells which are empty). So, once we computed $\mathrm{id}(\mathsf{p})$, we fetch the data structure for this cell, by using hashing. Namely, we store pointers to all those data-structures in a hash table, where each such data-structure is indexed by its unique id. Since the ids are integer numbers, we can do the hashing in constant time.

We are interested in solving the following problem.

Problem 3.3.2. Given a set P of n points in the plane, find the pair of points closest to each other. Formally, return the pair of points realizing $\mathcal{CP}(\mathsf{P}) = \min_{\mathsf{p},\mathsf{q} \in P} \|\mathsf{pq}\|$.

Lemma 3.3.3. *Given a set P of n points in the plane, and a distance r, one can verify in linear time, whether or not $\mathcal{CP}(\mathsf{P}) < r$ or $\mathcal{CP}(\mathsf{P}) \geq r$.*

Proof: Indeed, store the points of P in the grid G_r. For every non-empty grid cell, we maintain a linked list of the points inside it. Thus, adding a new point p takes constant time. Indeed, compute $\mathrm{id}(p)$, check if $\mathrm{id}(p)$ already appears in the hash table, if not, create a new linked list for the cell with this ID number, and store p in it. If a data-structure already exist for $\mathrm{id}(p)$, just add p to it.

This takes $O(n)$ time. Now, if any grid cell in $\mathrm{G}_r(\mathsf{P})$ contains more than, say, 9 points of P, then it must be that the $\mathcal{CP}(\mathsf{P}) < r$. Indeed, consider a cell C containing more than four points of P, and partition C into 3×3 equal squares. Clearly, one of those squares must contain two points of P, and let C' be this square. Clearly, the diameter of $C' = \mathrm{diam}(C)/3 = \sqrt{r^2 + r^2}/3 < r$. Thus, the (at least) two points of P in C' are in distance smaller than r from each other.

Thus, when we insert a point p, we can fetch all the points of P that were already inserted, for the cell of P, and the 8 adjacent cells. All those cells, must contain at most 9 points of P (otherwise, we would already have stopped since the $\mathcal{CP}(\cdot)$ of inserted points, is smaller than r). Let S be the set of all those points, and observe that $|S| \leq 9 \cdot 9 = O(1)$. Thus, we can compute by brute force the closest point to p in S. This takes $O(1)$ time. If $\mathsf{d}(p, S) < r$, we stop, otherwise, we continue to the next point, where $\mathsf{d}(p, S) = \min_{s \in S} \|ps\|$.

Overall, this takes $O(n)$ time. As for correctness, first observe that if $\mathcal{CP}(\mathsf{P}) > r$ then the algorithm would never make a mistake, since it returns '$\mathcal{CP}(\mathsf{P}) < r$' only after finding a pair of points of P with distance smaller than r. Thus, assume that p, q are the pair of points of P realizing the closest pair, and $\|pq\| = \mathcal{CP}(\mathsf{P}) < r$. Clearly, when the later of them, say p, is being inserted, the set S would contain q, and as such the algorithm would stop and return "$\mathcal{CP}(\mathsf{P}) < r$". ∎

Lemma 3.3.3 hints on a natural way to compute $\mathcal{CP}(\mathsf{P})$. Indeed, permute the points of P in arbitrary fashion, and let $P = \langle p_1, \ldots, p_n \rangle$. Next, let $r_i = \mathcal{CP}\Big(\{p_1, \ldots, p_i\}\Big)$. We can check if

$r_{i+1} < r_i$, by just calling the algorithm for Lemma 3.3.3 on P_{i+1} and r_i. In fact, if $r_{i+1} < r_i$, the algorithm of Lemma 3.3.3, would give us back the distance r_{i+1} (with the other point realizing this distance).

In fact, consider the "good" case, where $r_{i+1} = r_i = r_{i-1}$. Namely, the length of the shortest pair does not change. In this case, we do not need to rebuild the data structure of Lemma 3.3.3, for each point. We can just reuse it from the previous iteration. Thus, inserting a single point takes constant time, as long as the closest pair does not change.

Things become bad, when $r_i < r_{i-1}$. Because then, we need to rebuild the grid, and reinsert all the points of $\mathsf{P}_i = \langle p_1, \ldots, p_i \rangle$ into the new grid $\mathsf{G}_{r_i}(\mathsf{P}_i)$. This takes $O(i)$ time.

So, if the closest pair radius, in the sequence $r_1, \ldots, r_n$ changes only k times, then the running time of our algorithm would be $O(nk)$. In fact, we can do even better.

Theorem 3.3.4. *Let* P *be a set of n points in the plane, one can compute the closest pair of points of* P *in expected linear time.*

Proof: Pick a random permutation of the points of P, let $\langle p_1, \ldots, p_n \rangle$ be this permutation. Let $r_2 = \|p_1 p_2\|$, and start inserting the points into the data structure of Lemma 3.3.3. In the ith iteration, if $r_i = r_{i-1}$, then this insertion takes constant time. If $r_i < r_{i-1}$, then we rebuild the grid and reinsert the points. Namely, we recompute $\mathsf{G}_{r_i}(\mathsf{P}_i)$.

To analyze the running time of this algorithm, let X_i be the indicator variable which is 1 if $r_i \neq r_{i-1}$, and 0 otherwise. Clearly, the running time is proportional to

$$R = 1 + \sum_{i=2}^{n} (1 + X_i \cdot i) .$$

Thus, the expected running time is

$$\mathbf{E}[R] \quad = \quad 1 + \mathbf{E}\left[1 + \sum_{i=2}^{n} (1 + X_i \cdot i) \right] = n + \sum_{i=2}^{n} (\mathbf{E}[X_i] \cdot i) = n + \sum_{i=2}^{n} i \cdot \mathbf{Pr}[X_1 = 1] ,$$

by linearity of expectation and since for an indicator variable X_i, we have that $\mathbf{E}[X_i] = \mathbf{Pr}[X_i = 1]$.

Thus, we need to bound $\mathbf{Pr}[X_i = 1] = \mathbf{Pr}[r_i < r_{i-1}]$. To bound this quantity, fix the points of P_i, and randomly permute them. A point $\mathsf{q} \in \mathsf{P}_i$ is called ***critical***, if $\mathcal{CP}(\mathsf{P}_i \setminus \{\mathsf{q}\}) > \mathcal{CP}(\mathsf{P}_i)$. If there are no critical points, then $r_{i-1} = r_i$ and then $\mathbf{Pr}[X_i = 1] = 0$. If there is one critical point, than $\mathbf{Pr}[X_i = 1] = 1/i$, as this is the probability that this critical point, would be the last point in the random permutation of P_i.

If there are two critical points, and let p, q be this unique pair of points of P_i realizing $\mathcal{CP}(\mathsf{P}_i)$. The quantity r_i is smaller than r_{i-1}, if either p or q are p_i. But the probability for that is $2/i$ (i.e., the probability in a random permutation of i objects, that one of two marked objects would be the last element in the permutation).

Observe, that there can not be more than two critical points. Indeed, if p and q are two points that realize the closest distance, than if there is a third critical point r, then $\mathcal{CP}(\mathsf{P}_i \setminus \{\mathsf{r}\}) = \|\mathsf{pq}\|$, and r is not critical.

We conclude that

$$\mathbf{E}[R] = n + \sum_{i=2}^{n} i \cdot \mathbf{Pr}[X_1 = 1] \leq n + \sum_{i=2}^{n} i \cdot \frac{2}{i} \leq 3n.$$

As such, the expected running time of this algorithm is $O\Big(\mathbf{E}\Big[R\Big]\Big) = O(n)$. ■

Theorem 3.3.4 is a surprising result, since it implies that ***uniqueness*** (i.e., deciding if n real numbers are all distinct) can be solved in linear time. However, there is a lower bound of $\Omega(n \log n)$ on uniqueness, using the comparison tree model. This reality dysfunction, can be easily explained, once one realizes that the model of computation of Theorem 3.3.4 is considerably stronger, using hashing, randomization, and the floor function.

3.4 Bibliographical notes

Section 3.1 follows [MR95, Section 1.5]. The closest-pair algorithm follows Golin *et al.* [GRSS95]. This is in turn a simplification of a result of Rabin [Rab76]. Smid provides a survey of such algorithms [Smi00].

Bibliography

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press / McGraw-Hill, 2001.

[GRSS95] M. Golin, R. Raman, C. Schwarz, and M. Smid. Simple randomized algorithms for closest pair problems. *Nordic J. Comput.*, 2:3–27, 1995.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[Rab76] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, 1976.

[Smi00] M. Smid. Closest-point problems in computational geometry. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, pages 877–935. Elsevier, 2000.