Chapter 35

Backwards analysis

By Sariel Har-Peled, November 28, 2018[1] Version: 1.0

The idea of ***backwards analysis*** (or backward analysis) is a technique to analyze randomized algorithms by imagining as if it was running backwards in time, from output to input. Most of the more interesting applications of backward analysis are in Computational Geometry, but nevertheless, there are some other applications that are interesting and we survey some of them here.

35.1. How many times can the minimum change?

Let $\Pi = \pi_1 \ldots \pi_n$ be a random permutation of $\{1, \ldots, n\}$. Let $\mathcal{E}_i$ be the event that π_i is the minimum number seen so far as we read Π; that is, $\mathcal{E}_i$ is the event that $\pi_i = \min_{k=1}^{i} \pi_k$. Let X_i be the indicator variable that is one if $\mathcal{E}_i$ happens. We already seen, and it is easy to verify, that $\mathbb{E}[X_i] = 1/i$. We are interested in how many times the minimum might change[2]; that is $Z = \sum_i X_i$, and how concentrated is the distribution of Z. The following is maybe surprising.

Lemma 35.1.1. *The events $\mathcal{E}_1, \ldots, \mathcal{E}_n$ are independent (as such, variables $X_1, \ldots, X_n$ are independent).*

Proof: The trick is to think about the sampling process in a different way, and then the result readily follows. Indeed, we randomly pick a permutation of the given numbers, and set the first number to be π_n. We then, again, pick a random permutation of the remaining numbers and set the first number as the penultimate number (i.e., π_{n-1}) in the output permutation. We repeat this process till we generate the whole permutation.

Now, consider $1 \leq i_1 < i_2 < \ldots < i_k \leq n$, and observe that $\mathbb{P}\left[\mathcal{E}_{i_1} \,\middle|\, \mathcal{E}_{i_2} \cap \ldots \cap \mathcal{E}_{i_k}\right] = \mathbb{P}\left[\mathcal{E}_{i_1}\right]$, since by our thought experiment, $\mathcal{E}_{i_1}$ is determined after all the other variables $\mathcal{E}_{i_2}, \ldots, \mathcal{E}_{i_k}$. In particular, the variable $\mathcal{E}_{i_1}$ is inherently not effected by these events happening or not. As such, we have

$$\begin{aligned}\mathbb{P}\left[\mathcal{E}_{i_1} \cap \mathcal{E}_{i_2} \cap \ldots \cap \mathcal{E}_{i_k}\right] &= \mathbb{P}\left[\mathcal{E}_{i_1} \,\middle|\, \mathcal{E}_{i_2} \cap \ldots \cap \mathcal{E}_{i_k}\right] \mathbb{P}\left[\mathcal{E}_{i_2} \cap \ldots \cap \mathcal{E}_{i_k}\right] \\ &= \mathbb{P}\left[\mathcal{E}_{i_1}\right] \mathbb{P}\left[\mathcal{E}_{i_2} \cap \mathcal{E}_{i_2} \cap \ldots \cap \mathcal{E}_{i_k}\right] = \prod_{j=1}^{k} \mathbb{P}\left[\mathcal{E}_{i_j}\right] = \prod_{j=1}^{k} \frac{1}{i_j},\end{aligned}$$

by induction. ∎

Theorem 35.1.2. *Let $\Pi = \pi_1 \ldots \pi_n$ be a random permutation of $1, \ldots, n$, and let Z be the number of times, that π_i is the smallest number among $\pi_1, \ldots, \pi_i$, for $i = 1, \ldots, n$. Then, we have that for $t \geq 2e$ that $\mathbb{P}\left[Z > t \ln n\right] \leq 1/n^{t \ln 2}$, and for $t \in \left[1, 2e\right]$, we have that $\mathbb{P}\left[Z > t \ln n\right] \leq 1/n^{(t-1)^2/4}$.*

[2]The answer, my friend, is blowing in the permutation.

Proof: Follows readily from Chernoff's inequality, as $Z = \sum_i X_i$ is a sum of independent indicator variables, and, since by linearity of expectations, we have

$$\mu = \mathbb{E}\Big[Z\Big] = \sum_i \mathbb{E}\Big[X_i\Big] = \sum_{i=1}^{n} \frac{1}{i} \geq \int_{x=1}^{n+1} \frac{1}{x} \mathrm{d}x = \ln(n+1) \geq \ln n.$$

Next, we set $\delta = t - 1$, and use Chernoff inequality. ■

35.2. Yet another analysis of QuickSort

Rephrasing QuickSort. We need to restate QuickSort in a slightly different way for the backward analysis to make sense.

We conceptually can think about QuickSort as being a randomized incremental algorithm, building up a list of numbers in the order they are used as pivots. Consider the execution of QuickSort when sorting a set P of n numbers. Let $\langle \mathsf{p}_1, \ldots, \mathsf{p}_n \rangle$ be the random permutation of the numbers picked in sequence by QuickSort. Specifically, in the ith iteration, it randomly picks a number p_i that was not handled yet, pivots based on this number, and then recursively handles the subproblems.

Specifically, assume that at the end of the ith iteration, a set $\mathsf{P}_i = \{\mathsf{p}_1, \ldots, \mathsf{p}_i\}$ of pivots has already been handled by the algorithm. That is, the algorithm have these pivots in sorted orders $\mathsf{p}'_1 < \mathsf{p}'_2 < \ldots < \mathsf{p}'_i$. In addition, the numbers that were not handled yet $\mathsf{P} \setminus \mathsf{P}_i$, are partitions into sets $\mathsf{Q}_0, \ldots, \mathsf{Q}_i$, where all the numbers in $\mathsf{P} \setminus \mathsf{P}_i$ between p'_i and p'_{i+1} are in the set Q_i, for all i. In the $(i+1)$th iteration, QuickSort randomly picks a pivot $\mathsf{p}_{i+1} \in \mathsf{P} \setminus \mathsf{P}_i$, identifies the set Q_j that contains it, splits this set according to the pivot into two sets (i.e., a set for the smaller elements, and a set for the bigger elements). The algorithm QuickSort continues in this fashion till all the numbers were pivots.

Lemma 35.2.1. *Consider QuickSort being executed on a set P of n numbers. For any element $\mathsf{q} \in \mathsf{P}$, in expectation, q participates in $O(\log n)$ comparisons during the execution of QuickSort.*

Proof: Consider a specific element $\mathsf{q} \in \mathsf{P}$. For any subset $B \subseteq \mathsf{P}$, let $U(B)$ be the two closest numbers in B having q in between them in the original ordering of P. In other words, $U(B)$ contains the (at most) two elements that are the endpoints of the interval of $\mathbb{R} \setminus B$ that contains q. Let X_i be the indicator variable of the event that the pivot p_i used in the ith iteration is in $U(\mathsf{P}_i)$. That is, q got compared to the ith pivot when it was inserted. Clearly, the total number of comparisons q participates in is $\sum_i X_i$.

Now, we use backward analysis. Consider the state of the algorithm just after i pivots were handled (i.e., the end of the ith iteration). Consider the set $\mathsf{P}_i = \{\mathsf{p}_1, \ldots, \mathsf{p}_i\}$ and imagine that you know only what elements are in this set, but the internal ordering is not known to you. As such, as there are (at most) two elements in $U(\mathsf{P}_i)$, the probability that $\mathsf{p}_i \in U(\mathsf{P}_i)$ is at most $2/i$.

As such, the expected number of comparisons q participates in is $\mathbb{E}[\sum_i X_i] \leq \sum_{i=1}^{n} 2/i = O(\log n)$, as desired. This also implies that QuickSort takes $O(n \log n)$ time in expectation. ■

Exercise 35.2.2. *Prove using backward analysis that QuickSort takes $O(n \log n)$ with high probability.*

It is not true that the indicator variables $X_1, X_2, \ldots$ are independent (this is quite subtle and not easy to see, as such extending directly the proof of Theorem 35.1.2 for this case does not work.

35.3. Closest pair: Backward analysis in action

We are interested in solving the following problem:

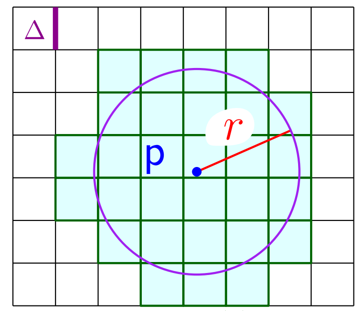
Problem 35.3.1. Given a set P of n points in the plane, find the pair of points closest to each other. Formally, return the pair of points realizing $\mathcal{CP}(\mathsf{P}) = \min_{\mathsf{p} \neq \mathsf{q},\, \mathsf{p},\mathsf{q} \in \mathsf{P}} \|\mathsf{p} - \mathsf{q}\|$.

35.3.1. Definitions

Definition 35.3.2. For a real positive number Δ and a point $\mathsf{p} = (\mathsf{p}_1, \ldots, \mathsf{p}_d) \in \mathbb{R}^d$, define $\mathsf{G}_\Delta(\mathsf{p})$ to be the grid point $(\lfloor \mathsf{p}_1/\Delta \rfloor \Delta, \ldots, \lfloor \mathsf{p}_d/\Delta \rfloor \Delta)$.

We call Δ the ***width*** or ***sidelength*** of the ***grid*** G_Δ. Observe that G_Δ partitions $\mathbb{R}^d$ into cubes, which are grid ***cells***. The grid cell of p is uniquely identified by the integer point $\mathrm{id}(\mathsf{p}) = \left(\lfloor \mathsf{p}_1/\Delta \rfloor, \ldots, \lfloor \mathsf{p}_d/\Delta \rfloor \right)$.

For a number $r \geq 0$, let $\mathsf{N}_{\leq r}(\mathsf{p})$ denote the set of grid cells in distance $\leq r$ from p, which is the ***neighborhood*** of p. Note, that the neighborhood also includes the grid cell containing p itself, and if $\Delta = \Theta(r)$ then $|\mathsf{N}_{\leq r}(\mathsf{p})| = \Theta\big((2 + \lceil 2r/\Delta \rceil)^d\big) = \Theta(1)$. See figure on the right.

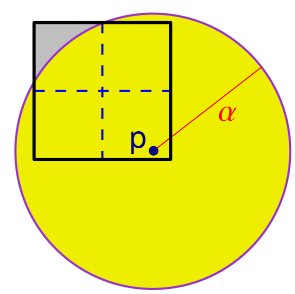


$\mathsf{N}_{\leq r}(\mathsf{p})$

35.3.2. Back to the problem

The following is an easy standard ***packing argument*** that underlines, under various disguises, many algorithms in computational geometry.

Lemma 35.3.3. *Let P be a set of points contained inside a square $\square$, such that the sidelength of $\square$ is $\alpha = \mathcal{CP}(\mathsf{P})$. Then $|\mathsf{P}| \leq 4$.*



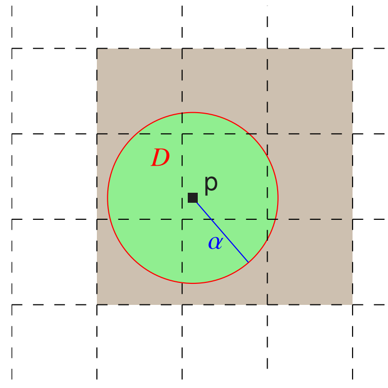
Proof: Partition $\square$ into four equal squares $\square_1, \ldots, \square_4$, and observe that each of these squares has diameter $\sqrt{2}\alpha/2 < \alpha$, and as such each can contain at most one point of P; that is, the disk of radius α centered at a point $\mathsf{p} \in \mathsf{P}$ completely covers the subsquare containing it; see the figure on the right. Note that the set P can have four points if it is the four corners of $\square$. ■

Lemma 35.3.4. *Given a set P of n points in the plane and a distance α, one can verify in linear time whether $\mathcal{CP}(\mathsf{P}) < \alpha$, $\mathcal{CP}(\mathsf{P}) = \alpha$, or $\mathcal{CP}(\mathsf{P}) > \alpha$.*

Proof: Indeed, store the points of P in the grid G_α. 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. Specifically, compute $\mathrm{id}(\mathsf{p})$, check if $\mathrm{id}(\mathsf{p})$ 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 linked list already exists for $\mathrm{id}(\mathsf{p})$, just add p to it. This takes $O(n)$ time overall.

Now, if any grid cell in $\mathsf{G}_\alpha(\mathsf{P})$ contains more than, say, 4 points of P, then it must be that the $\mathcal{CP}(\mathsf{P}) < \alpha$, by Lemma 35.3.3.

Thus, when we insert a point p, we can fetch all the points of P that were already inserted in the cell of p and the 8 adjacent cells (i.e., all the points stored in the cluster of p); that is, these are the cells of the grid G_α that intersects the disk $D = \mathrm{disk}(\mathsf{p}, \alpha)$ centered at p with radius α; see the figure on the right. If there is a point closer to p than α that was already inserted, then it must be stored in one of these 9 cells (since it must be inside D). Now, each one of those cells must contain at most 4 points of P by Lemma 35.3.3 (otherwise, we would already have stopped since the $\mathcal{CP}(\cdot)$ of the inserted points is smaller than α). Let S be the set of all those points, and observe that $|S| \le 9 \cdot 4 = O(1)$. Thus, we can compute, by brute force, the closest point to p in S. This takes $O(1)$ time. If $\mathbf{d}(\mathsf{p}, S) < \alpha$, we stop; otherwise, we continue to the next point.



Overall, this takes at most linear time.

As for correctness, observe that the algorithm returns '$\mathcal{CP}(\mathsf{P}) < \alpha$' only after finding a pair of points of P with distance smaller than α. So, assume that p and q are the pair of points of P realizing the closest pair and that $\|\mathsf{p} - \mathsf{q}\| = \mathcal{CP}(\mathsf{P}) < \alpha$. Clearly, when the later point (say p) is being inserted, the set S would contain q, and as such the algorithm would stop and return '$\mathcal{CP}(\mathsf{P}) < \alpha$'. Similar argumentation works for the case that $\mathcal{CP}(\mathsf{P}) = \alpha$. Thus if the algorithm returns '$\mathcal{CP}(\mathsf{P}) > \alpha$', it must be that $\mathcal{CP}(\mathsf{P})$ is not smaller than α or equal to it. Namely, it must be larger. Thus, the algorithm output is correct. ∎

Remark 35.3.5. Assume that $\mathcal{CP}(\mathsf{P} \setminus \{\mathsf{p}\}) \ge \alpha$, but $\mathcal{CP}(\mathsf{P}) < \alpha$. Furthermore, assume that we use Lemma 35.3.4 on P, where $\mathsf{p} \in \mathsf{P}$ is the last point to be inserted. When p is being inserted, not only do we discover that $\mathcal{CP}(\mathsf{P}) < \alpha$, but in fact, by checking the distance of p to all the points stored in its cluster, we can compute the closest point to p in $\mathsf{P} \setminus \{\mathsf{p}\}$ and denote this point by q. Clearly, pq is the closest pair in P, and this last insertion still takes only constant time.

35.3.3. Slow algorithm

Lemma 35.3.4 provides a natural way of computing $\mathcal{CP}(\mathsf{P})$. Indeed, permute the points of P in an arbitrary fashion, and let $\mathsf{P} = \langle \mathsf{p}_1, \ldots, \mathsf{p}_n \rangle$. Next, let $\alpha_{i-1} = \mathcal{CP}(\{\mathsf{p}_1, \ldots, \mathsf{p}_{i-1}\})$. We can check if $\alpha_i < \alpha_{i-1}$ by using the algorithm of Lemma 35.3.4 on P_i and α_{i-1}. In fact, if $\alpha_i < \alpha_{i-1}$, the algorithm of Lemma 35.3.4 would return '$\mathcal{CP}(\mathsf{P}_i) < \alpha_{i-1}$' and the two points of P_i realizing α_i.

So, consider the "good" case, where $\alpha_i = \alpha_{i-1}$; that is, the length of the shortest pair does not change when p_i is being inserted. In this case, we do not need to rebuild the data-structure of Lemma 35.3.4 to store $\mathsf{P}_i = \langle \mathsf{p}_1, \ldots, \mathsf{p}_i \rangle$. We can just reuse the data-structure from the previous iteration that was used by P_{i-1} by inserting p_i into it. Thus, inserting a single point takes constant time, as long as the closest pair does not change.

Things become problematic when $\alpha_i < \alpha_{i-1}$, because then we need to rebuild the grid data-structure and reinsert all the points of $\mathsf{P}_i = \langle \mathsf{p}_1, \ldots, \mathsf{p}_i \rangle$ into the new grid $\mathsf{G}_{\alpha_i}(\mathsf{P}_i)$. This takes $O(i)$ time.

In the end of this process, we output the number α_n, together with the two points of P that realize the closest pair.

Observation 35.3.6. *If the closest pair distance, in the sequence $\alpha_1, \ldots, \alpha_n$, changes only t times, then the running time of our algorithm would be $O(nt + n)$. Naturally, t might be $\Omega(n)$, so this algorithm might take quadratic time in the worst case.*

35.3.4. Linear time algorithm

Surprisingly[3], we can speed up the above algorithm to have linear running time by spicing it up using randomization.

We pick a random permutation of the points of P and let $\langle \mathsf{p}_1, \ldots, \mathsf{p}_n \rangle$ be this permutation. Let $\alpha_2 = \|\mathsf{p}_1 - \mathsf{p}_2\|$, and start inserting the points into the data-structure of Lemma 35.3.4. We will keep the invariant that α_i would be the closest pair distance in the set P_i, for $i = 2, \ldots, n$.

In the ith iteration, if $\alpha_i = \alpha_{i-1}$, then this insertion takes constant time. If $\alpha_i < \alpha_{i-1}$, then we know what is the new closest pair distance α_i (see Remark 35.3.5), rebuild the grid, and reinsert the i points of P_i from scratch into the grid G_{α_i}. This rebuilding of $\mathsf{G}_{\alpha_i}(\mathsf{P}_i)$ takes $O(i)$ time.

Finally, the algorithm returns the number α_n and the two points of P_n realizing it, as the closest pair in P.

Lemma 35.3.7. *Let t be the number of different values in the sequence $\alpha_2, \alpha_3, \ldots, \alpha_n$. Then $\mathbb{E}[t] = O(\log n)$. As such, in expectation, the above algorithm rebuilds the grid $O(\log n)$ times.*

Proof: For $i \geq 3$, let X_i be an indicator variable that is one if and only if $\alpha_i < \alpha_{i-1}$. Observe that $\mathbb{E}[X_i] = \mathbb{P}[X_i = 1]$ (as X_i is an indicator variable) and $t = \sum_{i=3}^{n} X_i$.

To bound $\mathbb{P}[X_i = 1] = \mathbb{P}[\alpha_i < \alpha_{i-1}]$, we (conceptually) fix the points of P_i and randomly permute them. A point $\mathsf{q} \in \mathsf{P}_i$ is ***critical*** if $\mathcal{CP}(\mathsf{P}_i \setminus \{\mathsf{q}\}) > \mathcal{CP}(\mathsf{P}_i)$. If there are no critical points, then $\alpha_{i-1} = \alpha_i$ and then $\mathbb{P}[X_i = 1] = 0$ (this happens, for example, if there are two pairs of points realizing the closest distance in P_i). If there is one critical point, then $\mathbb{P}[X_i = 1] = 1/i$, as this is the probability that this critical point would be the last point in the random permutation of P_i.

Assume 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 α_i is smaller than α_{i-1} only if either p or q is p_i. 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 cannot be more than two critical points. Indeed, if p and q are two points that realize the closest distance, then if there is a third critical point s, then $\mathcal{CP}(\mathsf{P}_i \setminus \{\mathsf{s}\}) = \|\mathsf{p} - \mathsf{q}\|$, and hence the point s is not critical.

Thus, $\mathbb{P}[X_i = 1] = \mathbb{P}[\alpha_i < \alpha_{i-1}] \leq 2/i$, and by linearity of expectations, we have that $\mathbb{E}[t] = \mathbb{E}\left[\sum_{i=3}^{n} X_i\right] = \sum_{i=3}^{n} \mathbb{E}[X_i] \leq \sum_{i=3}^{n} 2/i = O(\log n)$. ■

Lemma 35.3.7 implies that, in expectation, the algorithm rebuilds the grid $O(\log n)$ times. By Observation 35.3.6, the running time of this algorithm, in expectation, is $O(n \log n)$. However, we can do better than that. Intuitively, rebuilding the grid in early iterations of the algorithm is cheap, and only late rebuilds (when $i = \Omega(n)$) are expensive, but the number of such expensive rebuilds is small (in fact, in expectation it is a constant).

Theorem 35.3.8. *For set P of n points in the plane, one can compute the closest pair of P in expected linear time.*

Proof: The algorithm is described above. As above, let X_i be the indicator variable which is 1 if $\alpha_i \neq \alpha_{i-1}$, and 0 otherwise. Clearly, the running time is proportional to

$$R = 1 + \sum_{i=3}^{n} (1 + X_i \cdot i).$$

[3]Surprise in the eyes of the beholder. The reader might not be surprised at all and might be mildly annoyed by the whole affair. In this case, the reader should read any occurrence of "surprisingly" in the text as being "mildly annoying".

Thus, the expected running time is proportional to

$$\begin{aligned}\mathbb{E}\big[R\big] &= \mathbb{E}\Big[1 + \sum_{i=3}^{n}(1 + X_i \cdot i)\Big] \leq n + \sum_{i=3}^{n} \mathbb{E}\big[X_i\big] \cdot i \leq n + \sum_{i=3}^{n} i \cdot \mathbb{P}\big[X_i = 1\big]\\ &\leq n + \sum_{i=3}^{n} i \cdot \frac{2}{i} \leq 3n,\end{aligned}$$

by linearity of expectation and since $\mathbb{E}[X_i] = \mathbb{P}[X_i = 1]$ and since $\mathbb{P}[X_i = 1] \leq 2/i$ (as shown in the proof of Lemma 35.3.7). Thus, the expected running time of the algorithm is $O(\mathbb{E}[R]) = O(n)$. ■

Theorem 35.3.8 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. Indeed, compute the distance of the closest pair of the given numbers (think about the numbers as points on the x-axis). If this distance is zero, then clearly they are not all unique.

However, there is a lower bound of $\Omega(n \log n)$ on the running time to solve uniqueness, using the comparison model. This "reality dysfunction" can be easily explained once one realizes that the computation model of Theorem 35.3.8 is considerably stronger, using hashing, randomization, and the floor function.

35.4. Computing a good ordering of the vertices of a graph

We are given a $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ be an edge-weighted graph with n vertices and m edges. The task is to compute an ordering $\pi = \langle \pi_1, \ldots, \pi_n \rangle$ of the vertices, and for every vertex $v \in \mathsf{V}$, the list of vertices L_v, such that $\pi_i \in Ł_v$, if π_i is the closet vertex to v in the ith prefix $\langle \pi_1, \ldots, \pi_i \rangle$.

This situation can arise for example in a streaming scenario, where we install servers in a network. In the ith stage there i servers installed, and every client in the network wants to know its closest server. As we install more and more servers (ultimately, every node is going to be server), each client needs to maintain its current closest server.

The purpose is to minimize the total size of these lists $\mathcal{L} = \sum_{v \in \mathsf{V}} |L_v|$.

35.4.1. The algorithm

Take a random permutation $\pi_1, \ldots, \pi_n$ of the vertices V of G. Initially, we set $\delta(v) = +\infty$, for all $v \in \mathsf{V}$.

In the ith iteration, set $\delta(\pi_i)$ to 0, and start Dijkstra from the ith vertex π_i. The Dijkstra propagates only if it improves the current distance associated with a vertex. Specifically, in the ith iteration, we update $\delta(u)$ to $\mathsf{d}_{\mathsf{G}}(\pi_i, u)$ if and only if $\mathsf{d}_{\mathsf{G}}(\pi_i, u) < \delta(u)$ before this iteration started. If $\delta(u)$ is updated, then we add π_i to L_u. Note, that this Dijkstra propagation process might visit only small portions of the graph in some iterations – since it improves the current distance only for few vertices.

35.4.2. Analysis

Lemma 35.4.1. *The above algorithm computes a permutation π, such that $\mathbb{E}\Big[|\mathcal{L}|\Big] = O(n \log n)$, and the expected running time of the algorithm is $O\Big((n \log n + m) \log n\Big)$, where $n = |\mathsf{V}(\mathsf{G})|$ and $m = |\mathsf{E}(\mathsf{G})|$. Note, that both bounds also hold with high probability.*

Proof: Fix a vertex $v \in \mathsf{V} = \{v_1, \ldots, v_n\}$. Consider the set of n numbers $\{\mathsf{d}_\mathsf{G}(v, v_1), \ldots, \mathsf{d}_\mathsf{G}(v, v_n)\}$. Clearly, $\mathsf{d}_\mathsf{G}(v, \pi_1), \ldots, \mathsf{d}_\mathsf{G}(v, \pi_n)$ is a random permutation of this set, and by Lemma 35.1.1 the random permutation π changes this minimum $O(\log n)$ time in expectations (and also with high probability). This readily implies that $|L_v| = O(\log n)$ both in expectations and high probability.

The more interesting claim is the running time. Consider an edge $uv \in \mathsf{E}(\mathsf{G})$, and observe that $\delta(u)$ or $\delta(v)$ changes $O(\log n)$ times. As such, an edge gets visited $O(\log n)$ times, which implies overall running time of $O(n \log^2 n + m \log n)$, as desired.

Indeed, overall there are $O(n \log n)$ changes in the value of $\delta(\cdot)$. Each such change might require one delete-min operation from the queue, which takes $O(\log n)$ time operation. Every edge, by the above, might trigger $O(\log n)$ decrease-key operations. Using Fibonacci heaps, each such operation takes $O(1)$ time. ■

35.5. Computing nets

35.5.1. Basic definitions

35.5.1.1. Metric spaces

Definition 35.5.1. A ***metric space*** is a pair $(\mathcal{X}, \mathbf{d})$ where $\mathcal{X}$ is a set and $\mathbf{d} : \mathcal{X} \times \mathcal{X} \to [0, \infty)$ is a ***metric*** satisfying the following axioms: (i) $\mathbf{d}(x, y) = 0$ if and only if $x = y$, (ii) $\mathbf{d}(x, y) = \mathbf{d}(y, x)$, and (iii) $\mathbf{d}(x, y) + \mathbf{d}(y, z) \geq \mathbf{d}(x, z)$ (triangle inequality).

For example, $\mathbb{R}^2$ with the regular Euclidean distance is a metric space. In the following, we assume that we are given ***black-box access*** to $\mathbf{d}_\mathcal{M}$. Namely, given two points $\mathsf{p}, \mathsf{q} \in \mathcal{X}$, we assume that $\mathbf{d}(\mathsf{p}, \mathsf{q})$ can be computed in constant time.

Another standard example for a finite metric space is a graph G with non-negative weights $\omega(\cdot)$ defined on its edges. Let $\mathsf{d}_\mathsf{G}(x, y)$ denote the shortest path (under the given weights) between any $x, y \in \mathsf{V}(\mathsf{G})$. It is easy to verify that $\mathsf{d}_\mathsf{G}(\cdot, \cdot)$ is a metric. In fact, any ***finite metric*** (i.e., a metric defined over a finite set) can be represented by such a weighted graph.

35.5.1.2. Nets

Definition 35.5.2. For a point set P in a metric space with a metric $\mathbf{d}$, and a parameter $r > 0$, an r-***net*** of P is a subset $C \subseteq \mathsf{P}$, such that

(i) for every $\mathsf{p}, \mathsf{q} \in C$, $\mathsf{p} \neq \mathsf{q}$, we have that $\mathbf{d}(\mathsf{p}, \mathsf{q}) \geq r$, and
(ii) for all $\mathsf{p} \in \mathsf{P}$, we have that $\min_{\mathsf{q} \in C} \mathbf{d}(\mathsf{p}, \mathsf{q}) < r$.

Intuitively, an r-net represents P in resolution r.

35.5.2. Computing nets quickly for a point set in $\mathbb{R}^d$

The results here have nothing to do with backward analysis and are included here only for the sake of completeness.

There is a simple algorithm for computing r-nets. Namely, let all the points in P be initially unmarked. While there remains an unmarked point, p, add p to C, and mark it and all other points in distance $< r$ from p (i.e. we are scooping away balls of radius r). By using grids and hashing one can modify this algorithm to run in linear time. The following is implicit in previous work [Har04], and we include it here for the sake of completeness[4] – it was also described by the authors in [ERH12].

Lemma 35.5.3. *Given a point set $\mathsf{P} \subseteq \mathbb{R}^d$ of size n and a parameter $r > 0$, one can compute an r-net for P in $O(n)$ time.*

Proof: Let G denote the grid in $\mathbb{R}^d$ with side length $\Delta = r/\left(2\sqrt{d}\right)$. First compute for every point $\mathsf{p} \in \mathsf{P}$ the grid cell in G that contains p; that is, $\mathrm{id}(\mathsf{p})$. Let $\mathcal{G}$ denote the set of grid cells of G that contain points of P. Similarly, for every cell $\square \in \mathcal{G}$ we compute the set of points of P which it contains. This task can be performed in linear time using hashing and bucketing assuming the floor function can be computed in constant time. Specifically, store the $\mathrm{id}(\cdot)$ values in a hash table, and in constant time hash each point into its appropriate bin.

Scan the points of P one at a time, and let p be the current point. If p is marked then move on to the next point. Otherwise, add p to the set of net points, C, and mark it and each point $\mathsf{q} \in \mathsf{P}$ such that $\|\mathsf{p} - \mathsf{q}\| < r$. Since the cells of $\mathsf{N}_{\leq r}(\mathsf{p})$ contain all such points, we only need to check the lists of points stored in these grid cells. At the end of this procedure every point is marked. Since a point can only be marked if it is in distance $< r$ from some net point, and a net point is only created if it is unmarked when visited, this implies that C is an r-net.

As for the running time, observe that a grid cell, c, has its list scanned only if c is in the neighborhood of some created net point. As $\Delta = \Theta(r)$, there are only $O(1)$ cells which could contain a net point p such that $c \in \mathsf{N}_{\leq r}(\mathsf{p})$. Furthermore, at most one net point lies in a single cell since the diameter of a grid cell is strictly smaller than r. Therefore each grid cell had its list scanned $O(1)$ times. Since the only real work done is in scanning the cell lists and since the cell lists are disjoint, this implies an $O(n)$ running time overall. ■

Observe that the closest net point, for a point $\mathsf{p} \in \mathsf{P}$, must be in one of its neighborhood's grid cells. Since every grid cell can contain only a single net point, it follows that in constant time per point of P, one can compute each point's nearest net point. We thus have the following.

Corollary 35.5.4. *For a set $\mathsf{P} \subseteq \mathbb{R}^d$ of n points, and a parameter $r > 0$, one can compute, in linear time, an r-net of P, and furthermore, for each net point the set of points of P for which it is the nearest net point.*

In the following, a ***weighted point*** is a point that is assigned a positive integer weight. For any subset S of a weighted point set P, let $|S|$ denote the number of points in S and let $\omega(S) = \sum_{\mathsf{p} \in S} \omega(\mathsf{p})$ denote the total weight of S.

In particular, Corollary 35.5.4 implies that for a weighted point set one can compute the following quantity in linear time.

Algorithm 35.5.5 (net). *Given a weighted point set $\mathsf{P} \subseteq \mathbb{R}^d$, let $\mathcal{N}(r, \mathsf{P})$ denote an r-net of P, where the weight of each net point p is the total sum of the weights of the points assigned to it. We slightly abuse notation, and also use $\mathcal{N}(r, \mathsf{P})$ to designate the algorithm computing this net, which has linear running time.*

[4] Specifically, the algorithm of Har-Peled [Har04] is considerably more complicated than Lemma 35.5.3, and does not work in this settings, as the number of clusters it can handle is limited to $O\left(n^{1/6}\right)$. Lemma 35.5.3 has no such restriction.

35.5.3. Computing an r-net in a sparse graph

Given a $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ be an edge-weighted graph with n vertices and m edges, and let $r > 0$ be a parameter. We are interested in the problem of computing an r-net for G. That is, a set of vertices of G that complies with Definition 35.5.2p7.

35.5.3.1. The algorithm

We compute an r-net in a sparse graph using a variant of Dijkstra's algorithm with the sequence of starting vertices chosen in a random permutation.

Let π_i be the ith vertex in a random permutation π of V. For each vertex v we initialize $\delta(v)$ to $+\infty$. In the ith iteration, we test whether $\delta(\pi_i) \geq r$, and if so we do the following steps:

(A) Add π_i to the resulting net $\mathcal{N}$.

(B) Set $\delta(\pi_i)$ to zero.

(C) Perform Dijkstra's algorithm starting from π_i, modified to avoid adding a vertex u to the priority queue unless its tentative distance is smaller than the current value of $\delta(u)$. When such a vertex u is expanded, we set $\delta(u)$ to be its computed distance from π_i, and relax the edges adjacent to u in the graph.

35.5.3.2. Analysis

While the analysis here does not directly uses backward analysis, it is inspired to a large extent by such an analysis as in Section 35.4p6.

Lemma 35.5.6. *The set $\mathcal{N}$ is an r-net in G.*

Proof: By the end of the algorithm, each $v \in \mathsf{V}$ has $\delta(v) < r$, for $\delta(v)$ is monotonically decreasing, and if it were larger than r when v was visited then v would have been added to the net.

An induction shows that if $\ell = \delta(v)$, for some vertex v, then the distance of v to the set $\mathcal{N}$ is at most ℓ. Indeed, for the sake of contradiction, let j be the (end of) the first iteration where this claim is false. It must be that $\pi_j \in \mathcal{N}$, and it is the nearest vertex in $\mathcal{N}$ to v. But then, consider the shortest path between π_j and v. The modified Dijkstra must have visited all the vertices on this path, thus computing $\delta(v)$ correctly at this iteration, which is a contradiction.

Finally, observe that every two points in $\mathcal{N}$ have distance $\geq r$. Indeed, when the algorithm handles vertex $v \in \mathcal{N}$, its distance from all the vertices currently in $\mathcal{N}$ is $\geq r$, implying the claim. ∎

Lemma 35.5.7. *Consider an execution of the algorithm, and any vertex $v \in \mathsf{V}$. The expected number of times the algorithm updates the value of $\delta(v)$ during its execution is $O(\log n)$, and more strongly the number of updates is $O(\log n)$ with high probability.*

Proof: For simplicity of exposition, assume all distances in G are distinct. Let S_i be the set of all the vertices $x \in \mathsf{V}$, such that the following two properties both hold:

(A) $\mathsf{d}_\mathsf{G}(x, v) < \mathsf{d}_\mathsf{G}(v, \Pi_i)$, where $\Pi_i = \{\pi_1, \ldots, \pi_i\}$.

(B) If $\pi_{i+1} = x$ then $\delta(v)$ would change in the $(i+1)$th iteration.

Let $s_i = |S_i|$. Observe that $S_1 \supseteq S_2 \supseteq \cdots \supseteq S_n$, and $|S_n| = 0$.

In particular, let $\mathcal{E}_{i+1}$ be the event that $\delta(v)$ changed in iteration $(i+1)$ – we will refer to such an iteration as being ***active***. If iteration $(i+1)$ is active then one of the points of S_i is π_{i+1}. However, π_{i+1} has a uniform distribution over the vertices of S_i, and in particular, if $\mathcal{E}_{i+1}$ happens then $s_{i+1} \leq s_i/2$, with probability at least half, and we will refer to such an iteration as being ***lucky***. (It is possible that $s_{i+1} < s_i$ even if $\mathcal{E}_{i+1}$ does not happen, but this is only to our benefit.) After $O(\log n)$ lucky iterations the set S_i is empty, and we are done. Clearly, if both the ith and jth iteration are active, the events that they are each lucky are independent of each other. By the Chernoff inequality, after $c \log n$ active iterations, at least $\lceil \log_2 n \rceil$ iterations were lucky with high probability, implying the claim. Here c is a sufficiently large constant. ∎

Interestingly, in the above proof, all we used was the monotonicity of the sets $S_1, \ldots, S_n$, and that if $\delta(v)$ changes in an iteration then the size of the set S_i shrinks by a constant factor with good probability in this iteration. This implies that there is some flexibility in deciding whether or not to initiate Dijkstra's algorithm from each vertex of the permutation, without damaging the number of times of the values of $\delta(v)$ are updated.

Theorem 35.5.8. *Given a graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, with n vertices and m edges, the above algorithm computes an r-net of G in $O((n \log n + m) \log n)$ expected time.*

Proof: By Lemma 35.5.7, the two δ values associated with the endpoints of an edge get updated $O(\log n)$ times, in expectation, during the algorithm's execution. As such, a single edge creates $O(\log n)$ decrease-key operations in the heap maintained by the algorithm. Each such operation takes constant time if we use Fibonacci heaps to implement the algorithm. ∎

35.6. Bibliographical notes

Backwards analysis was invented/discovered by Raimund Seidel, and the QuickSort example is taken from Seidel [Sei93]. The number of changes of the minimum result of Section 35.1 is by now folklore.

The closet-pair result is Section 35.3 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].

The good ordering of Section 35.4 is probably also folklore, although a similar idea was used by Mendel and Schwob [MS09] for a different problem. Computing nets in $\mathbb{R}^d$, which has nothing to do with backwards analysis, Section 35.5.2, is from Har-Peled and Raichel [HR13].

Computing a net in a sparse graph, Section 35.5.3, is from [EHS14]. While backwards analysis fails to hold in this case, it provide a good intuition for the analysis, which is slightly more complicated and indirect.

Bibliography

[EHS14] D. Eppstein, S. Har-Peled, and A. Sidiropoulos. On the greedy permutation and counting distances. manuscript, 2014.

[ERH12] A. Ene, B. Raichel, and S. Har-Peled. Fast clustering with lower bounds: No customer too far, no shop too small. In submission. `http://sarielhp.org/papers/12/lbc/`, 2012.

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

[Har04] S. Har-Peled. Clustering motion. *Discrete Comput. Geom.*, 31(4):545–565, 2004.

[HR13] S. Har-Peled and B. Raichel. Net and prune: A linear time algorithm for Euclidean distance problems. In *Proc. 45th Annu. ACM Sympos. Theory Comput.* (STOC), pages 605–614, New York, NY, USA, 2013. ACM.

[MS09] M. Mendel and C. Schwob. Fast c-k-r partitions of sparse graphs. *Chicago J. Theor. Comput. Sci.*, 2009, 2009.

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

[Sei93] R. Seidel. Backwards analysis of randomized geometric algorithms. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, volume 10 of *Algorithms and Combinatorics*, pages 37–68. Springer-Verlag, 1993.

[Smi00] M. Smid. Closest-point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 877–935. Elsevier, Amsterdam, The Netherlands, 2000.