Chapter 36

Backwards analysis

By Sariel Har-Peled, April 26, 2022[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.

36.1. 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 36.1.1. *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 $1 \le i_1 < i_2 < \ldots < i_k \le n$, and observe that $\mathbb{P}\Big[\mathcal{E}_{i_1} \,\Big|\, \mathcal{E}_{i_2} \cap \ldots \cap \mathcal{E}_{i_k}\Big] = \mathbb{P}\big[\mathcal{E}_{i_1}\big]$, 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

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

by induction. ∎

Theorem 36.1.2. *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 $t \ge 2e$ that $\mathbb{P}\big[Z > t \ln n\big] \le 1/n^{t \ln 2}$, and for $t \in \big[1, 2e\big]$, we have that $\mathbb{P}\big[Z > t \ln n\big] \le 1/n^{(t-1)^2/4}$.*

[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. ∎

36.2. 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 $\mathcal{L} = \sum_{v \in \mathsf{V}} |L_v|$.

36.2.1. 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.

36.2.2. Analysis

Lemma 36.2.1. *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 $n = |V(\mathsf{G})|$ 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 36.1.1 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. ∎

36.3. Computing nets

36.3.1. Basic definitions

Definition 36.3.1. A ***metric space*** is a pair $(\mathcal{X}, \mathsf{d})$ where $\mathcal{X}$ is a set and $\mathsf{d} : \mathcal{X} \times \mathcal{X} \to [0, \infty)$ is a ***metric*** satisfying the following axioms: (i) $\mathsf{d}(x, y) = 0$ if and only if $x = y$, (ii) $\mathsf{d}(x, y) = \mathsf{d}(y, x)$, and (iii) $\mathsf{d}(x, y) + \mathsf{d}(y, z) \geq \mathsf{d}(x, z)$ (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 $\mathsf{d}_{\mathcal{M}}$. Namely, given two points $\mathsf{p}, \mathsf{u} \in \mathcal{X}$, we assume that $\mathsf{d}(\mathsf{p}, \mathsf{u})$ 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 $x, y \in V(\mathsf{G})$. 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.

36.3.1.1. Nets

Definition 36.3.2. For a point set P in a metric space with a metric d, and a parameter $r > 0$, an r-***net*** of P is a subset $C \subseteq \mathsf{P}$, such that

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

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

36.3.2. 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 36.3.2$_{\mathrm{p3}}$.

36.3.2.1. 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.

36.3.2.2. 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 36.2$_{\mathrm{p2}}$.

Lemma 36.3.3. *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 36.3.4. *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 36.3.5. *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 36.3.4, 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. ∎

36.4. 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 36.1 is by now folklore.

The good ordering of Section 36.2 is probably also folklore, although a similar idea was used by Mendel and Schwob [MS09] for a different problem.

Computing a net in a sparse graph, Section 36.3.2, 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.

References

[EHS14] D. Eppstein, S. Har-Peled, and A. Sidiropoulos. *On the Greedy Permutation and Counting Distances*. manuscript. 2014.

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

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