23
Smoothed Analysis of Algorithms

Smoothed analysis originates from an influential paper of Spielman and Teng, and provides an alternative to worst-case analysis. Assume that we want to analyze an algorithm's cost, e.g., the running time of an algorithm. Let $\text{cost}(A(I))$ be the cost that the algorithm has for input instance I. We will usually group the possible input instances according to their "size" (depending on the problem, this might be the number of jobs, vertices, variables and so on). Let $\mathcal{I}_n$ be the set of all inputs of 'size' n. For a worst case analysis, we are now interested in

$$\max_{I \in \mathcal{I}_n} \text{cost}(A(I)),$$

the maximum cost for any input of size n. Consider Figure 23.1 and imagine that all instances of size n are lined up on the x-axis. The blue line could be the running time of an algorithm that is fast for most inputs but very slow for a small number of instances. The worst case is the height of the tallest peak. For the green curve, which could for example be the running time of a dynamic programming algorithm, the worst case is a tight bound since all instances induce the same cost.

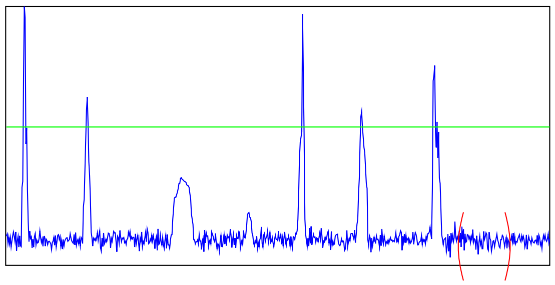

Figure 23.1: An example of a cost function with outliers.

Worst case analysis provides a bound that is true for all instances, but it might be very loose for some or most instances like it happens for the blue curve. When running the algorithm on real world data sets, the worst case instances might not occur, and there are different approaches to provide a more realistic analysis.

The idea behind smoothed analysis is to analyze how an algorithm acts on data that is jittered by a bit of random noise (notice that for real world data, we might have a bit of noise anyway). This is modeled by defining a notion of 'neighborhood' for instances and then analyzing the expected cost of an instance from the neighborhood of I instead of the running time of I.

The choice of the neighborhood is problem specific. We assume that the neighborhood of I is given in form of a probability distribution over all instances in $\mathcal{I}_n$. Thus, it is allowed that all instances in $\mathcal{I}_n$ are in the neighborhood but their influence will be different. The distribution depends on a parameter σ that says how much the input shall be jittered. We denote the neighborhood of I parameterized on σ by $\text{nbrhd}_\sigma(I)$. Then the smoothed complexity is given by

$$\max_{I \in \mathcal{I}_n} E_{I' \sim \text{nbrhd}_\sigma(I)}[\text{cost}(A(I'))]$$

In Figure 23.1, we indicate the neighborhood of one of the peak instances by the red brackets (imagine that the distribution is zero outside of the brackets – we will see one case of this in Section 23.3). The cost is smoothed within this area. For small σ, the bad instances get more isolated so that they dominate the expected value for their neighborhood, for larger σ, their influence decreases. We want the neighborhood to be big enough to smooth out the bad instances.

So far, we have mostly talked about the intuition behind smoothed analysis. The method has a lot of flexibility since the neighborhood can be defined individually for the analysis of each specific problem. We will see two examples in the following sections.

23.1 *The Traveling Salesman Problem*

The traveling salesman problem (TSP) is a classic example of an NP-complete problems with practical importance. In this problem, we are given an undirected weighted graph with (symmetric) edge weights $w_{ij} = w_{ji} \in [0, 1]$, find the minimum weighted cycle that contains all vertices.

On metric spaces there exists a polynomial 1.5-approximation algorithm (and now slightly better!) and on a d-dimensional Euclidean spaces (for fixed d) there are $1 + \varepsilon$ polynomial-time approximation schemes time (due to Sanjeev Arora, and others). In this chapter we consider the 2-OPT local-search heuristic.

23.1.1 *The 2-OPT Heuristic*

Start with an arbitrary cycle (i.e., a random permutation of vertices). In each iteration find two pairs of adjacent vertices (a, b) and (c, d) (i.e. $a \to b$ and $c \to d$ are edges in the cycle) and consider a new candidate cycle obtained by removing edges $a \to b$ and $c \to d$ as well as inserting edges $a \to c$ and $b \to d$. See Figure 23.2. If the new candidate cycle has smaller weight than the current cycle, replace the current cycle with the candidate one and repeat the heuristic. If no quadruplet (a, b, c, d) can decrease the weight, end the algorithm and report the final cycle.

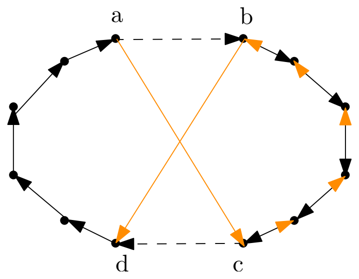

Figure 23.2: Main step of the 2-OPT heuristic: $a \to b, c \to d$ are replaced by $a \to c, b \to d$. The path $b \to \ldots \to c$ is also reversed.

There are two questions to be asked here: (a) how long does it take for us to reach the local optimum, and (b) what is the quality of the local optima?

It is useful to note that the 2-OPT always terminates since there at finite number of cycles (exactly $(n-1)!$ distinct ones) and each iteration strictly decreases the weight. However, there are examples on which the heuristic takes $\Omega(\exp(n))$ time as well as examples where it achieves value of $\Omega(\frac{\log n}{\log \log n}) \cdot OPT$. Despite this poor worst-case behavior, the heuristic performs well in practice, and hence it makes sense to see if smoothed analysis can explain its performance.

23.1.2 *The Smoothing Model and Result*

We set up the probability space by sampling each $w_{ij} \in [0, 1]$ from a independent distribution with probability density functions $f_{ij}(x)$ (note that the distributions can be different for different edges). The densities are bounded by $\frac{1}{\sigma}$, i.e. $f_{ij}(x) \leq \frac{1}{\sigma}$ for all i, j, x. The density can be otherwise completely arbitrary can chosen by the adversary.

We will prove that the expected number of iterations taken by 2-OPT is poly$(n, \frac{1}{\sigma})$. Let X be the the number of iterations; we write its expectation in terms of the probability that the algorithm takes more than t iterations via the formula

$$\begin{aligned} E[X] &= \sum_{t=1}^{\infty} Pr[X \geq t] \\ &= \sum_{t=1}^{(n-1)!} Pr[X \geq t]. \end{aligned} \tag{23.1}$$

Note that the $(n-1)!$ represents the maximum possible number of iterations the algorithm can take, since each swap is strictly improving and hence no permutation can repeat.

To bound this probability, we use the simple fact that a large number of iterations t means there is an iteration where the decrease in weight was very small, and this is an event with low probability.

Lemma 23.1. *The probability of an iteration with improvement (i.e., decrease in cycle weight) in the range $(0, \varepsilon]$ is at most $\frac{n^4 \varepsilon}{\sigma}$.*

Proof. Fix a quadruplet (a, b, c, d). We upper-bound the probability that the $(0, \varepsilon]$ improvement was obtained by replacing $a \to b, c \to d$ with $a \to c, b \to d$. The decrease in weight resulting from this replacement is $-w_{bd} - w_{ac} + w_{ab} + w_{cd} \in (0, \varepsilon]$. By conditioning on w_{bd}, w_{ac}, w_{ab}, the last unconditioned value w_{cd} must lie in some interval $(L, L + \varepsilon]$. By the hypothesis on the distribution density, this can happen with probability at most $\frac{\varepsilon}{\sigma}$, leading us to conclude that for a fixed (a, b, c, d) the probability of such an event is at most $\frac{\varepsilon}{\sigma}$.

By union-bounding over all n^4 quadruplets (a, b, c, d) we can bound the probability by $\frac{n^4 \varepsilon}{\sigma}$. □

Lemma 23.2. *The probability that the algorithm takes at least t iterations is at most $\frac{n^5}{t\sigma}$.*

Proof. Note that $w_{ij} \in [0, 1]$ implies that the weight of any cycle is in $[0, n]$. This implies by pigeonhole principle that there was an iteration where the decrease in weight was at most $\varepsilon := \frac{n}{t}$. By Lemma 23.1 the probability of this event is at most $\frac{n^5}{t\sigma}$, as advertised. □

Theorem 23.3. *The expected number of iterations that the algorithm takes is at most $O\left(\frac{n^6 \log n}{\sigma}\right)$.*

Proof. Using Equation 23.1 and Lemma 23.2 we have:

$$\begin{aligned} E[X] &= \sum_{t=1}^{(n-1)!} Pr[X \geq t] \\ &\leq \sum_{t=1}^{(n-1)!} \frac{n^5}{t\sigma} \\ &= \frac{n^5}{\sigma} \sum_{t=1}^{(n-1)!} \frac{1}{t} \\ &= O\left(\frac{n^6 \log n}{\sigma}\right) \end{aligned}$$

Here we used the fact that $\sum_{t=1}^{N} \frac{1}{t} = O(\log N)$ and $O(\log(n-1)!) = O(n \log n)$. □

An improved bound of blah was given by Details.

23.2 The Simplex Algorithm

We now turn our attention to the simplex algorithm for solving general linear programs. Given a vector $c \in \mathbb{R}^d$ and a matrix $A \in \mathbb{R}^{n \times d}$, this algorithm finds a solution variables $x = (x_1, \ldots, x_d)^\intercal$ that optimizes

$$\begin{aligned} \max \quad & c^\intercal x \\ & Ax \leq \mathbb{1}. \end{aligned}$$

This is a local-search algorithm, which iterates through solutions corresponding to vertices of the feasible solution polyhedron until it finds an optimal solution, or detects that the linear program is unbounded. It always goes from a solution to one of at least the same value, and there are situations where we have to make swaps that do not improve the value. Give example? In case there are many possible next moves, a ***pivot rule*** decides the next solution for the algorithm. Many pivot rules have been proposed. For most of them, we now know inputs where the simplex method iterates through an super-polynomial or exponential number of vertices, and thus has poor worst-case performance. Moreover, there is no pivot rule for which we can prove a polynomial number of steps.

Despite this, the simplex method is widely used to solve linear programs. In their seminal paper, Dan Spielman and Shang-Hua Teng show that the simplex algorithm has a polynomial smoothed complexity for a specific pivot rule, the ***"shadow vertex pivot rule"***.

23.2.1 *The Smoothing Model and Result*

More precisely, they have shown that the simplex method with this pivot rule provides a polynomial algorithm for the following problem:

They consider a slightly more general problem, where the right hand side could be some $b \in \mathbb{R}^n$ instead of $\mathbb{1}$.

Input: vector $c \in \mathbb{R}^d$, matrix $A \in \mathbb{R}^{n \times d}$

Problem: For a random matrix $G \in \mathbb{R}^{n \times d}$ and a random vector $g \in \mathbb{R}^n$ where all entries are chosen independently from a Gaussian distribution $\mathcal{N}(0, \max_i ||a_i||^2)$, solve the following LP:

$$\begin{aligned} \max \quad & c^\intercal x \\ & (A + G)x \leq \mathbb{1} + g. \end{aligned}$$

This is one specific neighborhood model. Notice that for any input $(A, c, \mathbb{1})$, all inputs $(A + G, c, \mathbb{1}, g)$ are potential neighbors, but the probability decreases exponentially when we go 'further away' from the original input. The vector c is not changed, only A and $\mathbb{1}$ are jittered. The variance of the Gaussian distributions scales with the smoothness parameter σ. For $\sigma = 0$, the problem reduces to the standard linear programming problem and the analysis becomes a worst case analysis.

Notice that randomly jittering A and $\mathbb{1}$ means that the feasibility of the linear program can be switched (from feasible to infeasible or vice versa). Thus, jittering the instance and then solving the LP does not necessarily give any solution for the original LP. However, assuming that the input comes from an appropriate noisy source, the following theorem gives a polynomial running time bound.

Theorem 23.4. *The 'smoothed' number of simplex steps executed by the simplex algorithm with the shadow vertex pivot rule is bounded by* $\text{poly}(n, d, 1/\sigma)$ *for the smoothed analysis model described above.*

The original result bounded the number of steps by $\mathcal{O}((nd/\sigma)^{O(1)})$, but the exponents were somewhat large. Roman Vershynin proved an improved bound of $\mathcal{O}(\log^7 n(d^9 + d^3/\sigma^4))$. An improved and simplified analysis due to Daniel Dadush and Sophie Huiberts gives a bound of $\approx d^{3.5}\sigma^{-2} \text{poly}\log n$. More on this later, see this survey chapter.

23.2.2 *The Shadow Vertex Pivot Rule*

We conclude with an informal description of the shadow vertex pivot rule. Consider Figure 23.3. The three-dimensional polytope stands for the polyhedron of all feasible solutions, which is in general d-dimensional. The vector c points in the direction in which we want to maximize (it is plotted with an offset to emphasize the optimal vertex). The shadow pivot rule projects the polyhedron to a two-dimensional space spanned by c and the starting vertex u.

Assume that the original LP has an optimal solution with finite objective value. Then the polyhedron must be bounded in the direction of c. It is also bounded in the direction of u since the start vertex u is optimal for the direction u.

After projecting the polyhedron we intuitively follow the vertices that lie on the convex hull of the projection (moving towards the direction of c)[1]. Notice that the extreme vertex on the c-axis is the projection of an optimal vertex of the original polyhedron.

Since the polyhedron of all feasible solutions is not known, the projection cannot be done upfront. Instead, in each step, the algorithm projects the vectors to the neighbor vertices onto $\text{span}\{c, u\}$ and identifies the neighbor which is the next on the convex hull.

A first step towards proving the result is to show that the shadow has a small number of vertices if the polyhedron $(A + G)x \leq (\mathbb{1} + g)$ is projected onto two *fixed* vectors u and c. The real result for simplex, however, is complicated by the fact that the vector u depends on the polyhedron and on the starting vertex, and so the polyhedron is projected to a subspace that is correlated to the polyhedron itself.

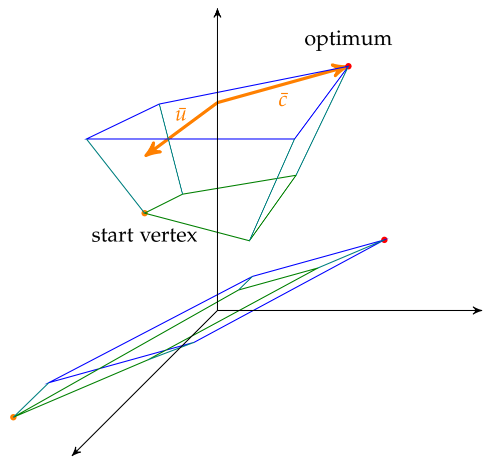

Figure 23.3: Illustration of the shadow vertex pivot rule.

[1] The two-dimensional projection is called the ***shadow*** of the original polyhedron, hence the name of the pivot rule.

Another complication: finding a starting vertex is as hard as solving an LP. Spielman and Teng handle these and other issues; see the original publications.

23.3 *The Knapsack Problem*

Finally, we turn to a problem for which we will give (almost) all the details of the smoothed analysis. The input to the ***knapsack problem*** is a collection of n items, each with size/weight $w_i \in \mathbb{R}_{\geq 0}$ and profit $p_i \in \mathbb{R}_{\geq 0}$, and the goal is to pick a subset of items that maximizes the sum of profits, subject to the total weight of the subset being at most some bound B.

As always, we can write the inputs as vectors $p, w \in \mathbb{R}^n_{\geq 0}$, and hence the solution as $x \in \{0,1\}^n$. For example, with $p = (1,2,3)$, w = $(2,4,5)$ and $B = 10$, the optimal solution is $x = (0,1,1)$ with $p^\intercal x = 5$ and $w^\intercal x = 9$.

The knapsack problem is weakly NP-hard—e.g., if the sizes are integers, then it can be solved by dynamic programming in time $O(nB)$. Notice that perturbing the input by a real number prohibits the standard dynamic programming approach which assumes integral input. Therefore we show a smoothed analysis for a different algorithm, one due to George Nemhauser and Jeff Ullman. The smoothed analysis result is by René Beier, Heiko Röglin, and Berthold Vöcking.

23.3.1 *The Smoothing Model*

One natural neighborhood is to smooth each weight w_i uniformly over an interval of width σ centered at w_i. Figure 23.4 illustrates this. Within the interval, the density function is uniform, outside of the interval, it is zero. The profits are not perturbed.

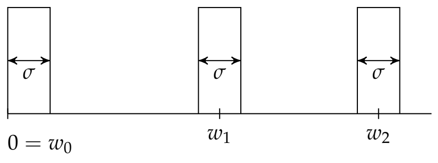

Figure 23.4: Distributions for three weights plotted into the same diagram.

We choose a slightly more general model already used earlier: The profits are fixed (chosen by the adversary), while the weights are chosen randomly and independently. The weight w_i is sampled from a distribution $f_i : [0,1] \to [0, 1/\sigma]$, where the distribution can be different for different i and is chosen by the adversary. The important thing here is that the distribution is bounded by $\frac{1}{\sigma}$ and our complexity will polynomially depend on this value. Note that we made a simplifying assumption that weights are in $[0,1]$ to make our lives easier although it can be avoided. We also assume that it never happens that two solutions get exactly the same weight. This is justified since this is an event with zero probability, so it is almost surely not the case.

23.3.2 *The Nemhauser–Ullman algorithm*

The Nemhauser-Ullman algorithm for the knapsack problem computes the Pareto curve and returns the best solution from the curve. The ***Pareto curve*** for the knapsack problem can defined as follows.

Definition 23.5. Given two knapsack solutions $x_1, x_2 \in \{0,1\}^n$ we say that "*x_1 is dominated by x_2*" if $w^\top x_1 > w^\top x_2$ and $p^\top x_1 \le p^\top x_2$. The *Pareto curve* is defined as the set of all non-dominated solutions (from a particular set of solutions). We also call the points on the curve *Pareto optimal*.

The definition is visualized in Figure 23.5. Note that the optimal solution is always on the Pareto curve. Moreover, if $P(j)$ is the collection of Pareto optimal solutions among the subinstance consisting of just the first j items, then

$$P(j+1) \subseteq P(j) \cup \underbrace{\{S \cup \{j+1\} \mid S \in P(j)\}}_{=:A_j}.$$

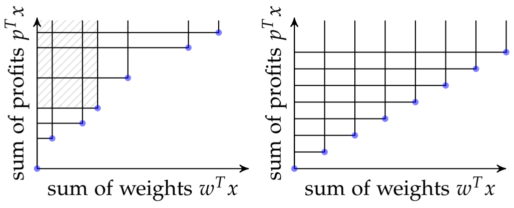

Figure 23.5: Left side: A point set and its Pareto curve. The blue points form the Pareto curve. They belong to the curve because the areas indicated by the black lines does not contain points. For the third point on the curve, this area is specifically highlighted. Right side: Worst Case.

The above containment implies that $P(j+1)$ can be computed from $P(j)$. In other words, a dominated solution by $P(j)$ will still be dominated in $P(j+1)$, so it can be safely forgotten.

If we keep the elements of $P(j)$ in sorted order with regard to the weights, then $P(j+1)$ can be computed in linear time by merging them together. Note that $P(j)$ and A_j are naturally constructed in increasing-weight order. Then we merge them in parallel (using the technique from merge-sort) and eliminate dominated points on the fly. The result is $P(j+1)$ with points stored in increasing-weight order. This technique leads to the following Lemma.

Lemma 23.6. *The Nemhauser-Ullman algorithm can be implemented to have a running time of* $O(\sum_{i=1}^{n} |P(j)|) = O(n \cdot \max_{j\in[n]} \mathbb{E}|P(j)|)$ *in expectation.*

Note that for this analysis we no longer care about the size of the knapsack g. The remainder of the section will be focused on bounding $\mathbb{E}[|P(j)|]$.

23.3.3 *Bounding Expected Size of Pareto Curve*

In the worst-case it is possible that all 2^n solutions are Pareto optimal. (Exercise: put $w_i = 2^i, p_i = 2^i$) In this case, the Nemhauser-Ullman algorithm will take exponential time. However, we will see that the expected size of $P(j)$ is polynomial for all $j \in [n]$ if the instance is σ-smoothed.

Fix any $j \in [n]$, and let $P := P(j)$. The proof idea is to partition the interval $[0,\infty]$ of **weights** into stripes of decreasing width $\varepsilon := 1/k$ for an integer k and then to count the number of Pareto optimal solutions in each stripe. See Figure 23.6. There is always an $\varepsilon > 0$ that is small enough such that each stripe contains at most one point from P (since we assumed no two solutions have the same weight). Denote by $P \cap [a,b]$ the set of all Pareto optimal solutions that have weight in the range $[a,b]$. Thus,

$$|P| = 1 + \lim_{k\to\infty} \sum_{\ell=0}^{\infty} \mathbb{1}\left(P \cap (\frac{\ell}{k}, \frac{\ell+1}{k}] \neq \emptyset\right).$$

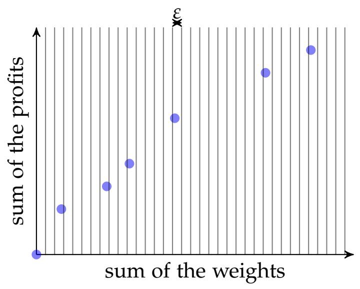

Figure 23.6: Dividing into stripes of width $\varepsilon = 1/k$.

We want to restrict the number of terms in the sum. Since the knapsack has size g, we can ignore all solutions that weight more than g. However, g might still be large. By our simplification, we know that all weights are at most 1. Thus, no solution can weight more than n and it is thus sufficient to consider the interval $[0,n]$. The kn stripes at $(0,1/k], (1/k,2/k], \dots, (n(k-1)/k, nk/k]$ fit into this interval. We thus get that

$$\mathbb{E}[|P|] \leq 1 + \lim_{k\to\infty} \sum_{\ell=0}^{nk} \Pr\left(P \cap (\frac{\ell}{k}, \frac{\ell+1}{k}] \neq \emptyset\right) \qquad (23.2)$$

Now we have to bound the probability that there is a point from P in the interval $(\frac{\ell}{k}, \frac{\ell+1}{k}]$. The following Lemma establishes this.

Lemma 23.7. *For any $t \geq 0$ it holds that* $\Pr(P \cap (t, t+\varepsilon] \neq \emptyset) \leq \frac{n\varepsilon}{\sigma}$.

Proof. Define x^R to be the leftmost point right of t that is on the Pareto curve.

$$x^R := \begin{cases} \arg\min_{x\in P}\{p^\intercal x \mid w^\intercal x > t\} & \text{if the set is nonempty} \\ \perp & \text{else} \end{cases}$$

We define Δ to be the distance between t and x^R:

$$\Delta := \begin{cases} w^\intercal x^R - t & x^R \neq \perp \\ \infty & \text{otherwise} \end{cases}$$

(See Figure 23.7 for a visualization.) Clearly:

$$P \cap (t, t+\varepsilon] \neq \emptyset \quad \iff \quad \Delta \in (0, \varepsilon]$$

The rest of the proof shows that $\Pr(\Delta \in (0,\varepsilon])$ is small.

It is difficult to directly argue about Δ, so we use a set of auxiliary random variables which make the proof clean. For any item $i \in [n]$ we define several random variables: let $x^{UL,-i}$ be the most profitable (highest) point left of t without the item i; and let $x^{*,+i}$ be the leftmost (least weight) point that is higher than $x^{UL,-i}$ and contains i. Formally:

$$x^{UL,-i} := \arg\max_{x\in 2^{[n]}}\{p^\intercal x \mid w^\intercal x \leq t,\, x_i = 0\}$$
$$x^{*,+i} := \arg\min_{x\in 2^{[n]}}\{w^\intercal x \mid p^\intercal x \geq p^\intercal x^{UL,-i},\, x_i = 1\}$$

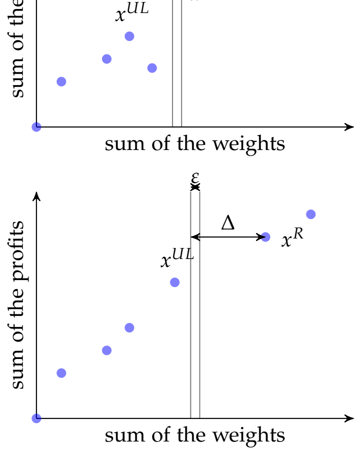

Figure 23.7: Illustration of the definition of x^{UL} and x^R. Δ is only plotted in the right figure.

The reason for defining these variables is that (i) it is easy to bound the probability that $x^{*,+i}$ has weight within an interval $(t, t+\varepsilon]$ and (ii) we will later show that $x^R = x^{*,+i}$ for some i. With this reason in mind we define $\Delta^i = x^{*,+i} - t$ (∞ if undefined).

Subclaim For any item $i \in [n]$ it holds that $\Pr\left[w^\intercal x^{*,+i} \in (t, t+\varepsilon]\right] \leq \frac{\varepsilon}{\sigma}$. In particular, $\Pr\left[\Delta^i \in (0, \varepsilon]\right] \leq \frac{\varepsilon}{\sigma}$.

Proof. Assume we fixed all weights except for i. Then $x^{UL,-i}$ is completely determined. We remind the reader that $x^{*,+i}$ is defined as the leftmost point that contains item i and is higher than $x^{UL,-i}$.

Now we turn our attention to the "precursor" of $x^{*,+i}$ without the item i, namely the item $x^{*,+i} - e_i$ where e_i is the i^{th} basis vector. The claim is that this point is completely determined when we fixed all weights except i. Name that point y (formal definition of this point will be given later). The point y is exactly the one that is leftmost with the condition that $y_i = 0$ and $p^\intercal y + p_i \geq p^\intercal x^{UL,-i}$ (by definition of $x^{*,+i}$). Note that the order of y's does not change when adding w_i. In particular, if y_1 was left of y_2 (had smaller weight), then adding the (currently undetermined) w_i will not change their ordering.

More formally, let $y := \arg\min_{y \in 2^{[n]}} \{w^\intercal y \mid p^\intercal y + p_i \geq p^\intercal x^{UL,-i}, y_i = 0\}$ (we drop the index i from y for clarity). In other words, it is the leftmost solution without i higher than $x^{UL,-i}$ when we add the profit of i to it. It holds that $w^\intercal x^{*,+i} = w^\intercal y + w_i$. Therefore,

$$\begin{aligned}
\Pr\left[w^\intercal x^{*,+i} \in (t, t+\varepsilon]\right] &= \Pr\left[w^\intercal y + w_i \in (t, t+\varepsilon]\right] \\
&= \Pr\left[w_i \in (t - w^\intercal y, t + \varepsilon - w^\intercal y]\right] \\
&\leq \frac{\varepsilon}{\sigma}
\end{aligned}$$

□

Subclaim There exists $i \in [n]$ s.t. $x^R = x^{*,+i}$. In particular, $\exists i$ s.t. $\Delta = \Delta^i$.

Proof. Let x^{UL} be the most profitable (highest) point left of t. Formally:

$$x^{UL} := \arg\max_{x \in P} \{p^\intercal x \mid w^\intercal x \leq t\}$$

Since the zero vector is always Pareto optimal there is always at least one point left of t, so x^{UL} is well-defined. There must be an item i s.t. that is contained in x^R, but is not in x^{UL}. Formally, pick and fix any $i \in [n]$ s.t. $x^R_i = 1, x^{UL}_i = 0$. It is clear that such i must exist since otherwise x^{UL} would have higher weight than x^R. Also, the height of x^R higher than x^{UL} since they are both on the Pareto curve.

Clearly (for this i) it holds that $x^{UL} = x^{UL,-i}$. Assume for the sake of contradiction that $x^{*,+i}$ is not x^R. Then:

- $x^{*,+i}$ must be left of x^R, otherwise we would have $x^{*,+i} = x^R$ and be done.
- $x^{*,+i}$ must be higher than x^{UL} by definition.
- $x^{*,+i}$ cannot be higher than x^R, otherwise x^R would not be on the Pareto curve (since it's left of it). So assume it's below x^R.
- $x^{*,+i}$ cannot be left of t, otherwise we would pick that point to be x^{UL} (since it's higher than x^{UL}).
- The only remaining spot for $x^{*,+i}$ is right of t and left of x^R, but that contradicts the choice of x^R as the leftmost point right of t.

Hence we conclude that $x^R = x^{*,+i}$ for our choice of i, which concludes the proof of the subclaim. □

Combining the above Subclaims, we get

$$\begin{aligned}\Pr[\Delta \in (0,\varepsilon]] &\leq \sum_{i=1}^{n} \Pr\left[\Delta^i \in (0,\varepsilon]\right] \\ &= n\frac{\varepsilon}{\sigma}.\end{aligned}$$

This is equivalent to the statement of the Lemma, hence we are done.

□

Using the above analysis, we conclude with a smoothness theorem.

Theorem 23.8. *For σ-smoothed instances, the expected size of P is bounded by n^2/σ for all $j \in [n]$. In particular, the Nemhauser-Ullman algorithm for the knapsack problem has a smoothed complexity of $O(n^3/\sigma)$ for the smoothness model described in Subsection 23.3.1.*

Proof. By Inequality (23.2), Lemma 23.6 and Lemma 23.7, we conclude that

$$\begin{aligned}\mathbb{E}[|P|] &\leq 1 + \lim_{k\to\infty} \textstyle\sum_{\ell=0}^{nk} \Pr\left(P \cap \left(\frac{\ell}{k}, \frac{\ell+1}{k}\right] \neq \emptyset\right) \\ &\leq 1 + \lim_{k\to\infty} nk \cdot \frac{n\frac{1}{k}}{\sigma} = \frac{n^2}{\sigma}.\end{aligned}$$

□

23.3.4 *More general result*

The result can be extended beyond the knapsack problem. Let Π be a "combinatorial optimization" problem given as

$$\begin{aligned} \max\ & p^{\mathsf{T}} n x \\ s.t.\ & Ax \leq b \\ & x \in \mathcal{S} \end{aligned}$$

where $\mathcal{S} \subseteq \{0,1\}^n$. Observe: the knapsack problem is one such problem. Beier and Vöcking prove the following theorem.

Theorem 23.9. *Problem Π has polynomial smoothed complexity if solving Π on unitarily encoded instances can be done in polynomial time.*

For the knapsack problem, the dynamic programming algorithm has a running time of $O(ng)$, where g is the size of the knapsack. If the instance is encoded in binary, then we need $O(n \log g)$ bits for the input and hence this algorithm is not polynomial-time; however if input is encoded in unary then we use $O(ng)$ to encode the instance, and the dynamic programming algorithm becomes polynomial-time.