Lecture 4: Derandomisation

Sometimes, having a randomised algorithm is wonderful, but what we really need is a *deterministic* version that achieves the same guarantees, but without the drawback of randomness: that is, we want the output to *always* be good (unlike Monte Carlo-type algorithms), and the running time to *always* be bounded (unlike Las Vegas-type algorithms). That is, we would like to be able, given any randomised algorithm $\mathcal{A}$, to "derandomise" it into an equally-good (or not much worse) deterministic version $\mathcal{A}'$. *Can we achieve that?*

Unsurprisingly, the answer is a resounding "we don't know." However, we do have some (limited) techniques to do so, in particular cases. Here, we will see two of them: the *small random seed approach* and the *method of conditional expectations*.

This is actually very much tied to one of the central questions in computational complexity, the P vs. BPP question.

To illustrate this, we will consider as running example the "maximum cut" question:

> Max-Cut: Given an (undirected) graph $G = (V, E)$ on n vertices and m edges, output a cut (A, B) (partition of V) *maximising* the number $c(A, B)$ of edges between A and B.

Of course, we would like an efficient algorithm for that. As a baseline, one could try to "just" find a could algorithm to solve the problem. Unfortunately, this is very unlikely to pan out:

Fact 0.21.1. Max-Cut *is NP-Hard.*

This is annoying, as this strongly hints we should give up on trying to find an efficient algorithm (deterministic, but, also, randomised – this is most likely very hard too) for Max-Cut. An *exact* algorithm, at least: but maybe we can get a good *approximation* algorithm?[8]

Here is an obvious randomised algorithm: *choose a cut (A, B) uniformly at random.* Or, with more words and in pseudocode:

```
$(A, B) \leftarrow (\emptyset, \emptyset)$
for all $v \in V$ do
    $X_v \leftarrow \mathrm{Bern}(1/2)$            ▷ Independent of previous choices
    if $X_v = 1$ then add $v$ to $A$
    else add $v$ to $B$
return $(A, B)$
```

Algorithm 6: Randomised algorithm for Max-Cut.

[8] Recall that an α-approximation algorithm is an algorithm whose output's value is within a factor $\alpha > 0$ of the optimal solution's value.

Is it any good? Maybe surprisingly, not too bad: in expectation, what it returns is a cut with at least *half* as many edges as the best possible:

Theorem 0.22. *For every $G = (V, E)$, the output (A, B) of Algorithm 6 satisfies*

$$\mathbb{E}[c(A,B)] \geq \frac{1}{2}m \geq \frac{1}{2}\mathrm{OPT}(G)\,.$$

Moreover, the algorithm runs in time $O(n)$.

Proof. The proof is immediate by linearity of expectation. Fix any edge $e \in E$ and let X_e denote the indicator random variable "e is a cut edge" (that is, one end is in A, the other in B). It is easy to check that $\mathbb{E}[X_e] = 1/2$ (both endpoints are in A with probability $1/2 \cdot 1/2 = 1/4$, same for both endpoints in B, so an edge crosses with probability $1/2$).

Rewriting $c(A,B) = \sum_{e \in E} X_e$, by linearity of expectation, we get

$$\mathbb{E}[c(A,B)] = \mathbb{E}\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} \mathbb{E}[X_e] = \sum_{e \in E} \frac{1}{2} = \frac{m}{2}$$

and the last part of the statement follows from observing that the best possible cut cannot have more than m edges. □

Can we convert Algorithm 6 into a *deterministic* (and still efficient) algorithm?

Method 1: derandomizing the random seed

Let us get back to the view of a randomized algorithm from the first lecture, as an "algorithm A taking an input x and a string of uniformly random bits $r \in \{0,1\}^*$." Imagine (1) A has a positive probability of returning a good solution; (2) we have a worst-case bound R on the *randomness complexity* of our algorithm, *i.e.*, on the maximum number of random bits it would every need on any input x; and (3) that, given a solution y to the task, that we can *verify* efficiently whether y is a good solution – say, by running another algorithm V on (x, y).

Then the claim is that the following algorithm is a deterministic algorithm that finds a good solution:

Algorithm 7: Derandomization approach (by brute-forcing over the random seed)

```
Require: Input $x$
for all $r \in \{0,1\}^R$ do
    $y \leftarrow A(x; r)$                ▷ Run $A$ on $x$ with randomness $r$
    if $V(x, y) = 1$ then               ▷ Verify if $y$ is a good solution
        return $y$                      ▷ If so, we are done
```

The fact that Algorithm 7 always returns a good solution, under our assumptions (1), (2), and (3), is immediate: there exists *some* choice of the randomness r for which A returns a good solution on

Importantly, this "good random seed" r may not be the same for all x.

input x; once we try this particular r in the loop, then we get a good solution y, and the verifier V successfully detects it.

There is, of course, a catch:

Fact 0.22.1. *Algorithm 7 runs in time $2^R(T_A + T_V)$, where T_A, T_V are the running times of the algorithm A and verifier V.*

In particular, given that R could be quite large (the only *a priori* bound we have is $R \leq T_A$), this could be really bad *even if A and V are efficient*: exponential in the input size, or even worse.

Can you see why?

So what to do? One hope we may have is to get a much better bound on R for some specific algorithms, or even to slightly modify these algorithms to make sure R is small. For instance, if we can design a randomised algorithm which only needs say $R \leq 2\log n$ bits of randomness on inputs of size n, then we get $2^R = n^2$: that's polynomial!

Looking back at Algorithm 6, it *seems* like we are using an awful number of random bits: one for each vertex $v \in V$, so $R = n$ in total. That is definitely not great. And yet, do we actually *need* this many independent random bits? Could we do with a much smaller number and use something like hash functions?

Hash functions are essentially magic: when you know how to use them, they are incredible. When you don't, you end up with a third arm growing out of your ear.

The only part of the proof of Theorem 0.22 where we used the randomness was to argue that each edge $e = (u, v)$ is a cut-edge with probability $1/2$. This argument requires independence of the two random bits involved: the random bit B_u for u, and the random bit B_v for vertex v. That is all:

As long as B_u and B_v are independent for each of the $\binom{n}{2}$ pairs of distinct vertices (u, v), the proof goes through!

That is called *pairwise independence*, and this is a *much* weaker requirement than (full) independence. In particular, we can use good hash functions to get pairwise independence very cheaply – to see how, let us introduce a key definition:

Definition 1. A family of functions $\mathcal{H} \subseteq \{h \colon \mathcal{X} \to \mathcal{Y}\}$ is a *family of pairwise independent hash functions*, or a *strongly universal hash family*, if, for every $x, x' \in \mathcal{X}$ with $x \neq x'$ and every $y, y' \in \mathcal{Y}$,

$$\Pr_{h \sim \mathcal{H}}\left[\, h(x) = y, h(x') = y' \,\right] = \frac{1}{|\mathcal{Y}|^2}$$

where the probability is over the uniformly random choice of $h \in \mathcal{H}$.

Importantly, here x, x', y, y' are not random! We pick a hash function h at random and see where it sends the inputs. So h is a *randomly picked hash function* (among the $|\mathcal{H}|$ choices), not a "random function": once h is picked, there is nothing random anymore.

Why does that help? Take the example of $\mathcal{X} = [n]$ and $\mathcal{Y} = \{0, 1\}$ in the definition above. Picking a hash function $h \in \mathcal{H}$ uniformly at random only takes $\log |\mathcal{H}|$ truly independent random bits. But with these $\log |\mathcal{H}|$ random bits, we obtain $|\mathcal{X}| = n$ random bits

$$h(1), h(2), \dots, h(n) \in \{0, 1\}$$

which are *not* fully independent, but such that *any two of them behaves exactly like a pair of uniformly random bits.* This is exactly what we need! The only missing part is: *do there exist "small" families of pairwise independent hash functions $\mathcal{H} \subseteq \{h\colon [n] \to \{0,1\}\}$?*

Small enough, because we want to use as few "true" random bits as possible, and that will cost us $\log|\mathcal{H}|$ of them.

Fact 0.22.2. *There exists an explicit*[9] *family of pairwise independent hash functions $\mathcal{H} \subseteq \{h\colon [n] \to \{0,1\}\}$ with $|\mathcal{H}| = 2^{\lceil \log(n+1) \rceil}$.*

This is great news! Now we can modify Algorithm 6 to first pick h uniformly at random from this specific $\mathcal{H}$ (this only requires $R \le \lceil \log(n+1) \rceil$ bits of randomness), and then use $X_v \leftarrow h(v)$ as random coin toss for vertex v in Line 3.

Algorithm 8: (Modified) Randomised algorithm for MAX-CUT to use a small random seed.

```
$(A,B) \leftarrow (\emptyset,\emptyset)$
Draw $h\colon V \to \{0,1\}$ uniformly at random from the $\mathcal{H}$ promised
  by Fact 0.22.2, using $R = \lceil \log(n+1) \rceil$ random bits
for all $v \in V$ do
    $X_v \leftarrow h(v)$                              ▷ Pairwise independence
    if $X_v = 1$ then add $v$ to $A$
    else add $v$ to $B$
return $(A,B)$
```

By pairwise independence, the proof of correctness of this (modified) Algorithm 6 goes through exactly as in Theorem 0.22, but now we use much fewer random bits... So, when derandomising the algorithm *via* Algorithm 7, we only pay a factor

$$2^R = 2^{\lceil \log(n+1) \rceil} \le 2(n+1) = O(n)$$

What about the rest? Well, we saw already that $T_A = O(n)$. As for the time T_V it takes to verify a cut (A,B) has size at least $m/2$, this is $T_V = O(m+n)$, and so by Fact 0.22.1 our "derandomised algorithm" has running time at most

$$2^R(T_A + T_V) = O(n(m+n)) .$$

Not bad. But we are missing a small part: one of the assumptions required to derandomise using Algorithm 7 was "(1) A has a positive probability of returning a good solution." We never checked this: all we know is that our randomised algorithm, Algorithm 8, returns a solution that is good (*i.e.*, with value at least $\frac{1}{2}m$) *in expectation*. Does that mean it has a *positive probability* of returning a good solution?
Thankfully, yes: it can be arbitrarily small, but it is positive:

Put differently: "a random variable cannot *always* be strictly below its expectation."

[9] Easy to construct and use. We will prove it in the tutorial!

Fact 0.22.3. *If X is a random variable such that $\mathbb{E}[X]$ exists, then* $\Pr[\,X \geq \mathbb{E}[X]\,] > 0$.

Proof. Given a random variable X with finite expectation $\mu :=$ $\mathbb{E}[X]$, we have $\mathbb{1}_{\{X<\mu\}} + \mathbb{1}_{\{X\geq\mu\}} = 1$, so

$$\begin{aligned}\mu = \mathbb{E}[X] &= \mathbb{E}\left[X\mathbb{1}_{\{X<\mu\}}\right] + \mathbb{E}\left[X\mathbb{1}_{\{X\geq\mu\}}\right] \\ &< \mathbb{E}\left[\mu\mathbb{1}_{\{X<\mu\}}\right] + \mathbb{E}\left[X\mathbb{1}_{\{X\geq\mu\}}\right] \\ &\leq \underbrace{\mu\Pr[\,X<\mu\,]}_{\leq\mu} + \mathbb{E}\left[X\mathbb{1}_{\{X\geq\mu\}}\right].\end{aligned}$$

If $\Pr[\,X \geq \mu\,] = 0$, then $\mathbb{1}_{\{X\geq\mu\}} = 0$ (always), so the second term is zero; and as a result we get $\mu < \mu$, a contradiction. □

Putting it all together, what we have done is going from Algorithm 6 (randomised algorithm) to Algorithm 8 (randomised algorithm using much fewer random bits) to a deterministic algorithm (using the general technique of Algorithm 7). This establishes the following:

Theorem 0.23. *There exists a* deterministic *algorithm A' for* Max-Cut *such that, for every $G = (V, E)$, the output (A, B) of A' satisfies*

$$c(A, B) \geq \frac{1}{2}m \geq \frac{1}{2}\,\mathrm{OPT}(G)\,.$$

Moreover, the algorithm runs in time $O(n \max(m, n))$.

Method 2: the method of conditional expectations

In some cases, the algorithm *does* need a lot of random bits, and there is no clear way to bring the randomness complexity R down. In these cases, there is (sometimes) an other option to use: the *method of conditional expectations*,[10] which we will see now in the context of our randomised algorithm for Max-Cut, Algorithm 6.

The method of conditional expectations essentially consists in looking at the sequence of random choices our algorithm made, and replacing these random choices one by one with deterministic choices which are always "at least as good as what the random choice would give in expectation."

Specifically, our randomised algorithm flips one coin per vertex, and the way we wrote it in Algorithm 6 it is doing so one vertex at a time.[11] Instead of flipping a coin, make the best *greedy* decision for the current bit to choose. For simplicity, let's order the vertices as $v_1, v_2, \dots, v_n$, and write $X_i \in \{0, 1\}$ for the bit X_{v_i} that tells us if $v_i \in A$.

What we will do first is set X_1 deterministically, say, without loss of generality, to 1. Then we will choose X_2 to ensure whatever choice we make *does not decrease* the expectation of $c(A, B)$ (over the remaining choices $X_3, \dots, X_n$, *if* we were to choose those uniformly at random). That is, we want to find an (efficiently computable, and

[10] This is also sometimes called the method of conditional probabilities.

[11] Note that in Algorithm 6, and Algorithm 8, we could actually make all these random choices in parallel. With this method though, we will need to make our choices sequentially.

deterministic) rule that tells us how to set X_{i+1} based on our previous choices $X_1, \dots, X_i$, which would ensure that the conditional expectation of $c(A, B)$ does not decrease:

$$\mathbb{E}[\, c(A,B) \mid X_1, \dots, X_i \,] \overset{\text{want}}{\leq} \mathbb{E}[\, c(A,B) \mid X_1, \dots, X_{i+1} \,] \tag{36}$$

If we had that, we would be in good shape, since then

$$\begin{aligned} \frac{m}{2} &= \mathbb{E}[c(A,B)] \\ &\leq \mathbb{E}[\, c(A,B) \mid X_1 \,] \\ &\leq \mathbb{E}[\, c(A,B) \mid X_1, X_2 \,] \\ &\leq \dots \\ &\leq \mathbb{E}[\, c(A,B) \mid X_1, X_2, \dots, X_n \,] \end{aligned}$$

and that very last term is the value of the cut we finally obtain once we have (deterministically) chosen $X_1, X_2, \dots, X_n$: there is no randomness left or choice remaining to make, we just have our cut (A, B)!

So *how* do we do this "derandomisation"? What is the rule we should follow to choose X_{i+1} based on previous choices in order to guarantee (36) holds? Observe that, for any given $1 \leq i \leq n-1$,

$$\begin{aligned} &\mathbb{E}[\, c(A,B) \mid X_1, \dots, X_i \,] \\ &\quad = \Pr[\, X_{i+1} = 0 \,]\mathbb{E}[\, c(A,B) \mid X_1, \dots, X_i, X_{i+1} = 0 \,] \\ &\qquad\quad + \Pr[\, X_{i+1} = 1 \,]\mathbb{E}[\, c(A,B) \mid X_1, \dots, X_i, X_{i+1} = 1 \,] \\ &\quad = \frac{1}{2}\mathbb{E}[\, c(A,B) \mid X_1, \dots, X_i, X_{i+1} = 0 \,] + \frac{1}{2}\mathbb{E}[\, c(A,B) \mid X_1, \dots, X_i, X_{i+1} = 1 \,] \\ &\quad \leq \max(\mathbb{E}[\, c(A,B) \mid X_1, \dots, X_i, X_{i+1} = 0 \,], \mathbb{E}[\, c(A,B) \mid X_1, \dots, X_i, X_{i+1} = 1 \,]) \end{aligned}$$

where the last inequality uses that $\frac{x+y}{2} \leq \max(x, y)$. So *if* we had a way to efficiently compute the two quantities

$$\mathbb{E}[\, c(A,B) \mid X_1, \dots, X_i, X_{i+1} = 0 \,]$$

and

$$\mathbb{E}[\, c(A,B) \mid X_1, \dots, X_i, X_{i+1} = 1 \,]$$

we could just greedily pick the choice of X_{i+1} corresponding to the maximum of the two, and we would be done.[12]

Luckily: here, we can. Let's take a step back: once we have already chosen $X_1, \dots, X_i$, we have decided where to put the first i vertices $v_1, \dots, v_i$: either in A, or not. Then, our choice for X_{i+1} can only affect the edges with one endpoint being v_{i+1}, so our decision can only impact two types of edges, depending on where their *other* endpoint is:

Phrased differently: at any given stage, $c(A, B)$ is the sum of the contribution of the edges already committed to (both endpoint vertices have been assigned to A, B), and those still open (at least one vertex endpoint not decided yet). The first contribution is fixed, and the expectation of the second is still $\frac{1}{2}$ for each edge.

- that endpoint is a vertex in $v_1, \dots, v_i$: our choice for v_{i+1} will fully determine whether these edges contribute to $c(A, B)$ or not.
- that endpoint is a vertex in $v_{i+2}, \dots, v_n$: our choice for v_{i+1} will leave open whether these edges contribute to $c(A, B)$. That decision will only be made in the future, separately for each of these

[12] Technically, we don't even need to compute the two values, we just need to have a way to figure out which one of the two is largest.

edges, when making the choice of whether to put that second endpoint into A.

When we set X_{i+1}, we only "commit" on the edges of the first type, *and that's all*. Therefore, the best rule is to choose X_{i+1} (whether to put v_{i+1} in A) in order to maximise the number of edges of the first kind that contribute to $c(A,B)$. This is easy to do in $O(m)$ time: for each of the two options for X_{i+1}, count the number of edges of the form (v_j, v_{i+1}) with $1 \leq j \leq i$ that would contribute to the cut:

$$
\begin{aligned}
N_A(i+1) &= \left|\left\{\ 1 \leq j \leq i \ :\ (v_j, v_{i+1}) \in E \text{ and } v_j \in B \ \right\}\right| && (37)\\
N_B(i+1) &= \left|\left\{\ 1 \leq j \leq i \ :\ (v_j, v_{i+1}) \in E \text{ and } v_j \in A \ \right\}\right| && (38)\\
& && (39)
\end{aligned}
$$

and pick whichever of the two options for which that number is the biggest! This will ensure (36) holds.

Algorithm 9: Derandomised algorithm for Max-Cut using the method of conditional expectations.

```
$(A,B) \leftarrow (\emptyset,\emptyset)$
Order the vertices as $v_1,\dots,v_n$ (arbitrarily)
for all $1 \leq i \leq n$ do
    Compute $N_A(i), N_B(i)$ as in Eqs. (37) and (38)
    if $N_A(i) \geq N_B(i)$ then add $v_i$ to $A$
    else add $v_i$ to $B$
return $(A,B)$
```

Overall, what we have shown is the following:

Theorem 0.24. *There exists a* deterministic *algorithm A'' (Algorithm 9) for* Max-Cut *such that, for every $G = (V,E)$, the output (A,B) ofA'' satisfies*

$$c(A,B) \geq \frac{1}{2}m \geq \frac{1}{2}\mathrm{OPT}(G)\,.$$

Moreover, the algorithm runs in time $O(nm)$.

We have seen two general derandomisation techniques:

- If we can show our randomised algorithm uses at most R truly uniformly random bits *and* any that given solution can be efficiently checked, then Fact 0.22.1 and Algorithm 7 provide a way to get a deterministic algorithm "almost as good", at the cost of a factor 2^R in the time complexity.
- Looking at the analysis of the algorithm, we can often achieve the first point by using *hash functions* (only requiring a small truly random seed), provided that the analysis only uses pairwise, or, more generally, k-wise independence.
- If our algorithm has some nice properties (namely, if can efficiently compute the *conditional expectation* of our solution's value given any setting of choices made so far), then the method of conditional expectations provides another powerful way of derandomising algorithms.

A further remark. Everything we have said about MAX-CUT in this chapter (and our algorithms) generalises to weighted graphs (and weighted cuts).

Try it!

Fact 0.24.1. *We can do better than* $1/2$*! There exists an* 0.878*-approximation algorithm – just not as simple. See Section 6.2 of* [13]*. We believe this is optimal, assuming something called the "Unique Games Conjecture" (UGC): but even without UGC, it is known we cannot do better than* $16/17 \approx 0.94$ *unless* $\mathsf{P} = \mathsf{NP}$*.*

A detour: the Probabilistic Method

Our example above with the first method[14] can be viewed an instance of a general proof technique called the *probabilistic method*. Namely, to prove existence of something (*e.g.*, a solution to a problem satisfying some nice properties ("there exists a maximum flow with integral flows"), or an object of a specific type ("there exists a bipartite graph such that XYZ"), etc.), there are several ways: one, very convenient, is to come up with an algorithm which outputs such an object. The algorithmic does not need to be efficient: if it outputs something of a particular type, then such things clearly must exist. This is a *constructive* way to establish existence.

The probabilistic method... doesn't do that. Instead, to prove that there exists some object x (in a big set $\mathcal{X}$) which satisfies some "good" property $P(x)$, we define a probability distribution D over $\mathcal{X}$, and then argue that

$$\Pr_{\mathbf{x}\sim D}[\, P(\mathbf{x}) \text{ holds}\,] > 0 \tag{40}$$

that is, an object $\mathbf{x} \in \mathcal{X}$ chosen *at random* according to D has a non-zero (maybe very small! But non-zero) probability of being "good." Well, if a randomly chosen object happens to be good with some non-zero probability, that means there must exist *some* good objects. . .

The key here is to choose a suitable probability distribution D over $\mathcal{X}$. This is a bit of an art, but often (when $\mathcal{X}$ is a finite set) considering the uniform distribution over $\mathcal{X}$ works.

Here is an example: given a graph $G = (V, E)$, a 2-colouring of the edges of G is a mapping $c\colon E \to \{\text{blue}, \text{red}\}$. Given a colouring of the graph, a set of vertices $S \subseteq V$ is said to be *monochromatic* if all the edges between vertices of S have the same color: $c(e) = \text{red}$ for all $e \in E \cap (S \times S)$, or $c(e) = \text{blue}$ for all $e \in E \cap (S \times S)$.

Take the complete graph on n vertices. Can we find a colouring of its edges such that no subset of 2 vertices is monochromatic (well, no)? No subset of 3 vertices? No subset of k vertices? For which values of k is that possible?

Theorem 0.25 (A sufficient condition on k)**.** *Fix* $0 \leq k \leq n$ *such that*

$$\binom{n}{k} 2^{-\binom{k}{2}} < \frac{1}{2}$$

[13] David P. Williamson and David B. Shmoys. *The design of approximation algorithms*. Cambridge University Press, Cambridge, 2011

[14] When we used Fact 0.22.3 to convert the expected guarantee into a non-zero probability of a good output.

Then, there exists a 2-colouring of the edges of the complete graph K_n such that no subset of k vertices is monochromatic.

Proof. Let's take a random colouring c. More precisely, let's take a *uniformly* random colouring c: each edge $e \in E$ is red or blue with probability $1/2$, and chosen independently from all other edge colours. We want to show that the probability (over the choice of c) that there is no monochromatic set of size k is non-zero; equivalently, that the probability that there exists (at least) one monochromatic subset S of size k is strictly less than 1.

Consider any (fixed) subset S of k vertices. Since S has size k and we start with the complete graph, there are $\binom{k}{2}$ edges between vertices of S; so the probability that our randomly chosen c makes S monochromatic is

$$\begin{aligned}\Pr\left[\substack{\text{all edges are blue or}\\ \text{all edges are red}}\right] &= \Pr[\,\text{all edges are blue}\,] + \Pr[\,\text{all edges are red}\,]\\ &= \frac{1}{2^{\binom{k}{2}}} + \frac{1}{2^{\binom{k}{2}}} = \frac{2}{2^{\binom{k}{2}}}\end{aligned}$$

where we used independence of the choice across edges to get $(1/2)^{\binom{k}{2}}$. That tells us the probability that a given, fixed subset S is monochromatic. So to bound the probability that this happens to *at least one* of them, we use a union bound over all those subsets. There are exactly $\binom{n}{k}$ of them, so by a union bound

$$\begin{aligned}\Pr\left[\substack{\text{there is at least}\\ \text{one monochromatic}\\ \text{subset of size } k}\right] &= \Pr\left[\bigcup_{S:|S|=k} \{S \text{ is monochromatic}\}\right]\\ &\leq \sum_{S:|S|=k} \Pr[\,S \text{ is monochromatic}\,]\\ &\leq \binom{n}{k} \cdot \frac{2}{2^{\binom{k}{2}}}\end{aligned}$$

which is strictly less than 1 whenever $\binom{n}{k}2^{-\binom{k}{2}} < \frac{1}{2}$. We are done. □

Further reading: the (excellent) book by Alon and Spencer[15].

[15] Noga Alon and Joel H. Spencer. *The probabilistic method*. Wiley Series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., Hoboken, NJ, fourth edition, 2016