# Chapter 1

# Motivating Example

In this first part of the book, we review the probabilistic background required for the rest of the text, along with several motivating examples that illustrate the use of these basic probabilistic tools in algorithm design.

We start with a motivating example: a surprisingly fast algorithm—discovered only recently—for one of the oldest problems in graph theory: $(\Delta+1)$ vertex coloring of a graph with maximum degree $\Delta$. The goal of the example is to showcase how some "advanced" algorithms can be quite simple still and yet achieve performances that may sound even counter intuitive at first glance. It also shows the power of *randomization* in algorithm design, a topic which we will come back to repeatedly in this book.

## 1.1 Problem: $(\Delta + 1)$ Vertex Coloring

Given an undirected graph $G = (V, E)$, we use $\Delta$ to denote the maximum degree of $G$. For any integer $c \geqslant 1$, a **proper *c*-coloring** of $G$ is an assignment of colors from $\{1, \ldots, c\}$ to the *vertices* of the graph, such that for every edge $(u, v) \in E$, the color assigned to $u$ and $v$ is different (we may drop the term 'proper' in some places for the simplicity of exposition). Figure 1.1 gives an illustration of these definitions.

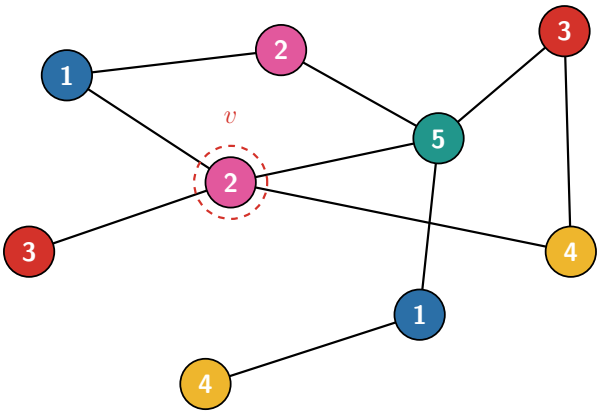


**Figure 1.1:** An example of a proper 5-coloring of a graph $G$ with maximum degree $\Delta = 4$. The numbers in the vertices denote their colors from $\{1, \ldots, 5\}$.

In this chapter, we are interested in the following problem.

**Problem 1.1.** Given an undirected graph $G = (V, E)$ with maximum degree $\Delta$, find a proper $(\Delta + 1)$ coloring of $G$.

A simple observation is that every graph admits such a $(\Delta + 1)$ coloring. In fact, there is a simple greedy algorithm for finding this coloring: color vertices of the graph in some arbitrary order, and when it is vertex $v$'s turn to be colored, check all neighbors of $v$ that are already colored, and find a color missing from all of them. By the pigeonhole principle, since $v$ has at most $\Delta$ neighbors but $(\Delta+1)$ colors to choose from, we can find a missing color for $v$ and move on (see the marked vertex $v$ in Figure 1.1). Once the algorithm terminates, we will end up with a $(\Delta+1)$ coloring of $G$. It is easy to see that this algorithm can be implemented in $O(n\Delta)$ time (see Exercise 1.1 at the end of this chapter).

But, can we find a $(\Delta+1)$ coloring even faster than $O(n\Delta)$ time? At first glance, this seems trivially hopeless – after all, an input graph with maximum degree $\Delta$ may have $\Omega(n\Delta)$ edges and thus even reading the entire input once requires us to spend $\Omega(n\Delta)$ time. It turns out that this intuition is actually not entirely correct and one can indeed obtain even faster algorithms at least for certain ranges of $\Delta$, by using randomization. The first algorithm to achieve this is due to [ACK19] which is rather complicated (and it actually does a lot more than just obtaining a faster $(\Delta + 1)$ coloring algorithm). We will see a different and very simple algorithm for this problem by a recent result of [AY25].

## 1.2 A Faster $(\Delta + 1)$ Coloring Algorithm

Our goal is to design a *randomized* algorithm for finding a $(\Delta+1)$ coloring. There are in general two approaches when it comes to randomized algorithms:

- Either the algorithm always outputs a correct answer and use randomization only to reduce the runtime; in this case, we are interested in the *expected runtime* of the algorithm. These algorithms are often called **Las Vegas** (randomized) algorithms.

  More formally, for each input $x$ and choice of random bits $r$ for the algorithm, let $T(x, r)$ denote the runtime of the algorithm on this particular input $x$ and this choice of random bits. We are interested in minimizing $\mathbb{E}_r[T(x, r)]$ among all inputs $x$.

- The other option is to allow some error in the algorithm and use randomization to obtain the correct answer with high probability. These algorithms are often called **Monte Carlo** (randomized) algorithms.

  More formally, for each input $x$ and choice of random bits $r$ for the algorithm, let $O(x, r) \in$ {'correct','incorrect'} denote whether the output of the algorithm on this particular input and this choice of random bits is correct or not. We are in interested in maximizing

$$\Pr_r(O(x, r) = \text{'correct'}).$$

In many cases, one can also translate algorithms of one type into the other type, and we will see that later chapters.

For this chapter, we will be designing a (Las Vegas) randomized algorithm that always outputs a $(\Delta + 1)$ coloring of any given graph and its expected runtime is

$$O(\frac{n^2}{\Delta} \cdot \log n).$$

Notice that whenever $\Delta = \omega(\sqrt{n \log n})$, the runtime of this algorithm will be $o(n\Delta)$ and thus (possibly) even faster than reading the entire input. In general, we can combine this algorithm and the standard greedy algorithm to obtain an algorithm for $(\Delta+1)$ coloring that runs in expected $O(n\sqrt{n \log n})$ time[1]– for any graph with more than these many edges, this algorithm runs in *sublinear* time!

### 1.2.1 The Algorithm

Our goal is to prove the following theorem due to [AY25].

**Theorem 1.2** ([AY25]). *There is a randomized algorithm that outputs a $(\Delta+1)$ coloring of any given graph $G = (V, E)$ with maximum degree $\Delta$ in $O(n^2 \log n/\Delta)$ expected time. The algorithm assumes $\Delta$ is provided as the input and the graph is presented by its adjacency matrix.*

The algorithm follows the approach of the greedy algorithm quite closely. It will iterate over the vertices in some order (to be determined later). For every vertex $v$, as in the greedy algorithm, it attempts to find a color to assign to $v$ and will not change its decision afterwards. The difference between this algorithm and the standard greedy algorithm is here: instead of going over all neighbors of $v$ to find a missing color for $v$, the new algorithm picks a color $c \in [\Delta+1]$ uniformly at random for $v$ and then go over all vertices that are colored $c$ to see if they are neighbor to $v$; if not, $v$ will be colored $c$ and the algorithm moves on, otherwise, it simply picks another random color (again, from all of $[\Delta+1]$ without even discarding "bad" colors) and continue like this until it can actually color $v$.

**Algorithm 1.3.**

1. Let $C_1, C_2, \ldots, C_{\Delta+1} = \emptyset$. ▷ we use $C_c$ to store vertices that are colored $c$
2. Iterate over vertices in some order (to be determined); for each vertex $v$ in this order:
   (a) Sample a color $c \in [\Delta+1]$ uniformly at random;
   (b) For every vertex $u \in C_c$, i.e., colored $c$ so far, check if $u$ is a neighbor of $v$;
   (c) If none of $C_c$ is neighbor to $v$, color vertex $v$ with $c$ by updating $C_c \leftarrow C_c \cup \{v\}$; otherwise, go to the sampling step in Line (2a) again.

We can observe that if the algorithm does indeed terminate, then the coloring it finds is a proper $(\Delta+1)$-coloring of the input graph: Any new vertex colored does not create a conflict with previously colored vertices (given the algorithm explicitly checks to not color $v$ with a color $c$ if one of its neighbors is already colored $c$) and thus at the end, there cannot be any monochromatic edges in the graph.

The only actual part of the analysis is to figure out how long this algorithm takes (in expectation). Before getting to this however, we need to take care of an important issue.

**Representation of the input.** Graphs are typically presented to algorithms as adjacency list or adjacency matrix. In many cases, the choice of representation does not matter much as we can simply switch between them without much extra cost by simply reading the input once (especially going from adjacency list to adjacency matrix is quite straightforward). Yet, in our setting, we are interested in an algorithm that aims to solve the problem faster than even reading the input once. As such, the representation of the input actually matters in our case.

[1] Basically, run this algorithm when $\Delta \geqslant \sqrt{n \log n}$ and otherwise run the original greedy algorithm.

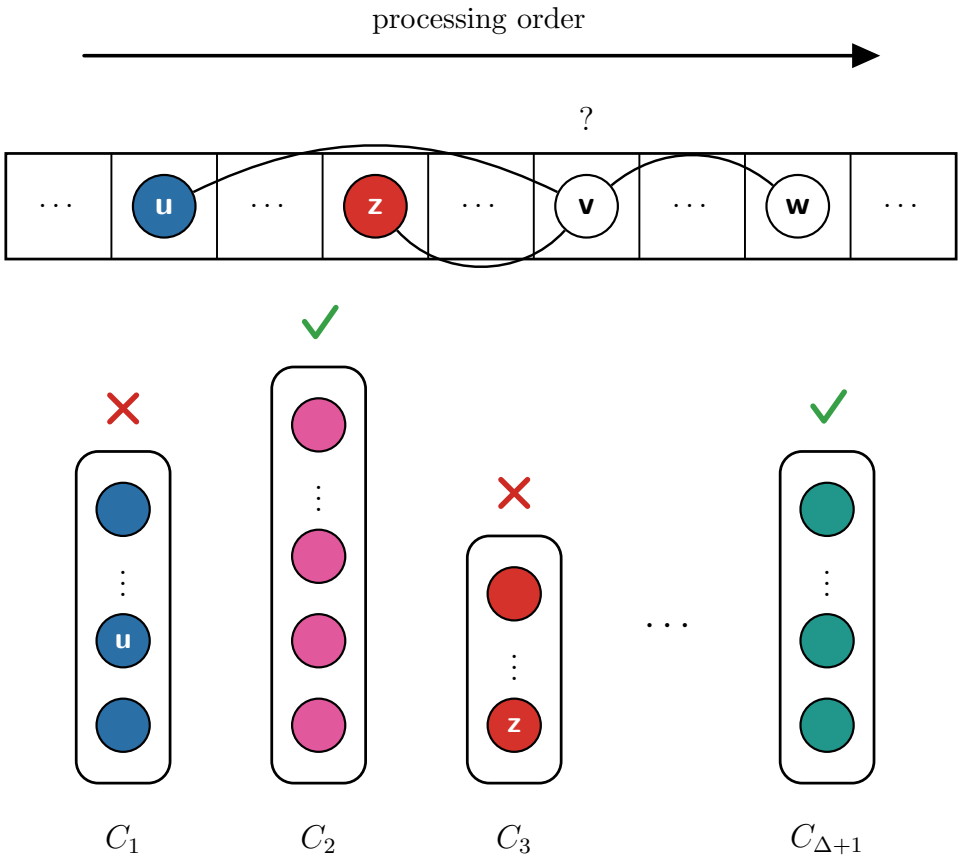


**Figure 1.2:** An illustration of Algorithm 1.3. When coloring a vertex $v$, its neighbors that appear before $v$ in the processing order already belong to some color class; so, if that color class is chosen in Line (2a), the algorithm detects a neighbor of $v$ inside it and choose a different color; for other color classes however, the color is available to $v$ and will be used to color $v$ if sampled. The average size of color classes throughout the algorithm is also at most $n/(\Delta+1)$.

So, what does our algorithm need? It is easy to see that the only access of the algorithm to the graph is in Line (2b): it needs to check for a given vertex $v$ and another vertex $u \in C_c$, whether or not $(u, v)$ is an edge in the graph. For this purpose, having access to the adjacency *matrix* is more helpful as it immediately allows for implementing such a *query* to the input in $O(1)$ time[2]. Thus, as stated in Theorem 1.2 also, we assume the input is presented to the algorithm by its adjacency matrix.

### 1.2.2 Runtime Analysis

For every vertex $v \in V$, define a random variable $X_v$, which is equal to the total number of entries of the adjacency matrix queried by Algorithm 1.3 in Line (2b) when attempting to color the vertex $v$. Basically, if the algorithm "tries" colors $c_1, c_2, \ldots, c_k$ before it finds a color for the vertex $v$, then $X_v := \sum_{i=1}^{k} |C_{c_i}|$. Note that it is possible for $X_v$ to be even infinity (although the probability if this event tends to zero). We have,

$$\text{Runtime of Algorithm 1.3} = O(1) \cdot \sum_{v \in V} X_v. \tag{1.1}$$

[2]This is *not* the only way: for instance, if the input is stored in a way that neighbors of every vertex is stored in a dedicated hash table (or a balanced search tree), we obtain almost the same guarantees.

This is simply because the only real time consuming step of the algorithm is this step and all the rest only take $O(n)$ time (deterministically) which is suppressed in the above asymptotic runtime. Thus, our goal is now to upper bound

$$\mathbb{E}\left[\text{Runtime of Algorithm 1.3}\right] = \mathbb{E}\left[O(1)\cdot\sum_{v\in V} X_v\right] = O(1)\sum_{v\in V}\mathbb{E}\left[X_v\right].$$

Here, the first step is by Eq (1.1) and the second one is by *linearity of expectation*. Thus, our task is now to simply bound $\mathbb{E}\left[X_v\right]$ for any given vertex $v \in V$.

Fix a vertex $v$ and define $\deg^{<}(v)$ as the number of neighbors of $v$ that appear *before* $v$ in the ordering of Algorithm 1.3. In other words, $\deg^{<}(v)$ counts the neighbors of $v$ that already received a color before coloring $v$. We claim the following.

**Claim 1.4.** *For every* $v \in V$

$$\mathbb{E}\left[X_v\right] \leqslant \frac{n}{\Delta + 1 - \deg^{<}(v)}.$$

Before proving this (simple) claim, let us start with two warm-up questions that provide more intuition about the algorithm and its analysis.

- **Probability a random color is "good" for $v \in V$:** What is the probability that when we sample a color for the vertex $v$, it can be used to color $v$, defined as $c$ being **good** for $v$? For this to happen, this color should not be the color of one of the already colored neighbors of $v$ which are $\deg^{<}(v)$ many. Thus, even if all these neighbors receive a different color, we have,

$$\begin{aligned}\Pr\left(c \text{ is } \underline{\text{not}} \text{ used in the neighborhood of } v\right) &= 1 - \Pr\left(c \text{ is used in the neighborhood of } v\right)\\ &\geqslant 1 - \frac{\deg^{<}(v)}{\Delta+1} = \frac{\Delta + 1 - \deg^{<}(v)}{\Delta+1}.\end{aligned}$$

- **Expected number of queries for a random $c \in [\Delta+1]$:** Another question is that when we sample a single color $c \in [\Delta+1]$, how many queries the algorithm needs to do (in expectation) for just this single color, regardless of whether or not this color is "good" for $v$. This can be easily calculated by the definition of expected value to be

$$\frac{1}{\Delta+1}\sum_{c=1}^{\Delta+1}|C_c| \leqslant \frac{n}{\Delta+1},$$

since picking a color $c$ means making $|C_c|$ queries, and the sets $C_1, \ldots, C_{\Delta+1}$ form a partition of (a subset of) vertices of the graph at every point (since every vertex receives a single color).

We now use the same ideas to prove Claim 1.4 also.

*Proof of Claim 1.4.* By the law of conditional expectations, we have,

$$\mathbb{E}\left[X_v\right] = \sum_{c=1}^{\Delta+1}\Pr\left(\text{first sampled color for } v \text{ was } c\right)\cdot\mathbb{E}\left[X_v \mid \text{first sampled color for } v \text{ was } c\right].$$

Now suppose we know the first sampled color for $v$ was $c$, what does it tell us about $X_v$? Firstly, we know that $X_v$ involves making $|C_c|$ queries. Secondly, if $c$ is **good** for $v$ meaning it can be used to color $v$, then we are done. In other words, for every $c$ which is good for $v$, we have,

$$\mathbb{E}\left[X_v \mid \text{first sampled color for } v \text{ was } c \text{ (which is a good color)}\right] = |C_c|.$$

But, what if $c$ is not good for $v$? Then, we spend $|C_c|$ queries and are in exactly the same place as we were when we started (since the algorithm simply does a "goto step" to the beginning of coloring $v$). Thus, for every $c$ which is not good for $v$,

$$\mathbb{E}\left[X_v \mid \text{first sampled color for } v \text{ was } c \text{ (which is not a good color)}\right] = \mathbb{E}\left[|C_c| + X_v\right] = |C_c| + \mathbb{E}\left[X_v\right].$$

Letting $B(v)$ denote the set of colors that are not good for $v$ and noting that $|B(v)| \leqslant \deg^{<}(v)$ (for the same reason as the one when calculating probability a color is not good for $v$), we have,

$$\begin{aligned}
\mathbb{E}\left[X_v\right] &= \sum_{c=1}^{\Delta+1} \Pr\left(\text{first sampled color for } v \text{ was } c\right) \cdot \mathbb{E}\left[X_v \mid \text{first sampled color for } v \text{ was } c\right] \\
&= \sum_{c=1}^{\Delta+1} \frac{1}{\Delta+1} \cdot |C_c| + \sum_{c \in B(v)} \frac{1}{\Delta+1} \cdot \mathbb{E}\left[X_v\right] \\
&\qquad \text{(since the given probability is } 1/\Delta+1 \text{ and by the discussions above)} \\
&\leqslant \frac{n}{\Delta+1} + \frac{\deg^{<}(v)}{\Delta+1} \cdot \mathbb{E}\left[X_v\right].
\end{aligned}$$

Moving $\mathbb{E}\left[X_v\right]$ to the LHS in the equation above and simplifying the terms imply the result. □

At this point, we have proven that

$$\mathbb{E}\left[\text{Runtime of Algorithm 1.3}\right] = O(1) \cdot \sum_{v \in V} \frac{n}{\Delta + 1 - \deg^{<}(v)}. \tag{1.2}$$

To conclude the runtime calculation, we need to bound this RHS. The challenge however is that $\deg^{<}(v)$ can be quite different for different vertices and in fact, the value of this sum can change dramatically depending on the ordering of vertices in the algorithm. This is where we have to specify the ordering of vertices in picking vertices in Algorithm 1.3.

**Fixing the ordering of vertices in Algorithm 1.3.** The solution here is quite simple: we will also pick the ordering of the vertices uniformly at random!

Let $\pi$ denote an ordering of vertices. Eq (1.2) shows that no matter the choice of $\pi$, the runtime of the algorithm can be bounded by the given expression. although now that ordering of vertices is also random, $\deg^{<}(v)$ is a random variable that depends on $\pi$; we denote this by $\deg^{<}_{\pi}(v)$ for more clarity. Thus, by taking the randomness of $\pi$ into account, we will have

$$\begin{aligned}
\mathbb{E}\left[\text{Runtime of Algorithm 1.3}\right] &= \mathop{\mathbb{E}}_{\pi} \mathbb{E}\left[\text{Runtime of Algorithm 1.3} \mid \pi\right] \\
&\qquad \text{(by the law of conditional probabilities)} \\
&= O(1) \cdot \sum_{v \in V} \mathop{\mathbb{E}}_{\pi} \left[\frac{n}{\Delta + 1 - \deg^{<}_{\pi}(v)}\right]. \\
&\qquad \text{(by Eq (1.2) and linearity of expectation)}
\end{aligned}$$

The final step is to bound this RHS for every vertex $v \in V$, which is done in the following claim.

**Claim 1.5.** *For every vertex $v \in V$,*

$$\mathop{\mathbb{E}}_{\pi} \left[\frac{n}{\Delta + 1 - \deg^{<}_{\pi}(v)}\right] \leqslant \frac{n}{\Delta+1} \cdot O(\log \Delta).$$

*Proof.* Let $\deg(v)$ denote the degree of $v$ in the entire graph. By picking $\pi$ randomly, we also obtain that the ordering of the vertices $v \cup N(v)$ (where $N(v)$ are neighbors of $v$) is chosen

uniformly at random. In particular, the relative order of $v$ among $v \cup N(v)$ is chosen uniformly. As such, $\deg_\pi^<(v)$ is chosen uniformly at random from $\{0, 1, \ldots, \deg(v)\}$. Hence,

$$\mathop{\mathbb{E}}_{\pi}\left[\frac{n}{\Delta+1-\deg_\pi^<(v)}\right] = \sum_{d=0}^{\deg(v)} \frac{1}{\deg(v)+1} \cdot \frac{n}{\Delta+1-d}. \qquad \text{(by the discussion above)}$$

Note that for any choice of $\deg(v)$, we have,

$$\sum_{d=0}^{\deg(v)} \frac{1}{\deg(v)+1} \cdot \frac{n}{\Delta+1-d} \leqslant \sum_{d=0}^{\Delta} \frac{1}{\Delta+1} \cdot \frac{n}{\Delta+1-d},$$

because if we let $A := \{n/(\Delta+1-i)\}_{i=0}^{\Delta}$, then, in the LHS, we are taking the average of the *smallest* $\deg(v)+1$ numbers in $A$, whereas in the RHS we are taking the average of all of $A$. Continuing the above then we have,

$$\begin{aligned}\sum_{d=0}^{\Delta} \frac{1}{\Delta+1} \cdot \frac{n}{\Delta+1-d} &= \frac{n}{\Delta+1} \cdot \sum_{i=1}^{\Delta+1} \frac{1}{i} && \text{(by a change of variable)}\\ &\leqslant \frac{n}{\Delta+1} \cdot \Big(\ln(\Delta+1)+1\Big),\end{aligned}$$

where we used the standard inequality that $\sum_{i=1}^{K} 1/i \leqslant \ln(K)+1$ (the series is called the Harmonic series)[3] Putting everything together, we have,

$$\mathop{\mathbb{E}}_{\pi}\left[\frac{n}{\Delta+1-\deg_\pi^<(v)}\right] \leqslant \frac{n}{\Delta+1} \cdot (\ln(\Delta+1)+1),$$

concluding the proof. □

Plugging in all the bounds we proved so far, we have that

$$\mathbb{E}\left[\text{Runtime of Algorithm 1.3}\right] = \sum_{v \in V} \mathbb{E}\left[X_v\right] \leqslant \sum_{v \in V} \frac{n}{\Delta+1} \cdot O(\log \Delta) = O(\frac{n^2}{\Delta} \cdot \log(\Delta)),$$

concluding the proof of Theorem 1.2.

Thus, at this point, we saw an algorithm and its full analysis for finding a $(\Delta+1)$ coloring in *sublinear* time (for certain ranges of $\Delta$). This concludes our first chapter in this book. In the rest of the book, we will see various other algorithms and analysis techniques for them.

## Exercises

**Exercise 1.1.** Given an implementation of the greedy algorithm for Problem 1.1 running in $O(n\Delta)$ time deterministically.

**Exercise 1.2.** Given an implementation of the greedy algorithm for Problem 1.1 running in $O(n+m)$ time deterministically, where $m$ is the number of edges in the graph.

[3] A simple proof of the inequality is the following:

$$\sum_{i=1}^{K} \frac{1}{i} \leqslant 1 + \int_{x=1}^{K} \frac{1}{x}\,dx = 1 + \ln(K) - \ln(1) = 1 + \ln(K). \qquad \text{(the derivative of } \ln(x) \text{ is } 1/x)$$

**Exercise 1.3.** Consider a modification of Algorithm 1.3 wherein the order of vertices is chosen arbitrarily instead of randomly. Show an example of a graph and an ordering of vertices such that

$$\sum_{v \in V} \left[ \frac{n}{\Delta + 1 - \deg_{\pi}^{<}(v)} \right] = \Omega(n^2).$$

This shows for the particular analysis we had for Algorithm 1.3, we crucially need to pick the ordering of vertices randomly.

**Exercise 1.4 (♦).** Consider the algorithm in Exercise 1.3. Either (1) provide an example of a graph and ordering of its vertices such that the algorithm takes $\Omega(n^2)$ time in expectation, or, (2) prove that the algorithm has expected $o(n^2)$ runtime.

*Note:* the *analysis* in this chapter cannot beat the $\Theta(n^2)$ bound (by what you have proved in Exercise 1.3). But, on its own, that is not enough to say the algorithm cannot beat the $\Theta(n^2)$ bound as it may be amenable to a different analysis with more efficient bounds.