Juho Hirvonen and Jukka Suomela:

Distributed Algorithms 2020

https://jukkasuomela.fi/da2020/ · September 20, 2023

Chapter 6

Randomized Algorithms

All models of computing that we have studied so far were based on the formalism that we introduced in Chapter 3: a distributed algorithm A is a state machine whose state transitions are determined by functions $\mathrm{init}_{A,d}$, $\mathrm{send}_{A,d}$, and $\mathrm{receive}_{A,d}$. Everything has been fully deterministic: for a given network and a given input, the algorithm will always produce the same output. In this chapter, we will extend the model so that we can study randomized distributed algorithms.

6.1 Definitions

Let us first define a *randomized distributed algorithms in the* PN *model* or, in brief, a *randomized* PN *algorithm*. We extend the definitions of Section 3.3 so that the state transitions are chosen randomly according to some probability distribution that may depend on the current state and incoming messages.

More formally, the values of the functions init and receive are discrete probability distributions over States_A. The initial state of a node u is a random variable $x_0(u)$ chosen from a discrete probability distribution

$$\mathrm{init}_{A,d}(f(u))$$

that may depend on the local input $f(u)$. The state at time t is a random variable $x_t(u)$ chosen from a discrete probability distribution

$$\mathrm{receive}_{A,d}\big(x_{t-1}(u), m_t(u)\big)$$

that may depend on the previous state $x_{t-1}(u)$ and on the incoming messages $m_t(u)$. All other parts of the model are as before. In particular, function $\mathrm{send}_{A,d}$ is deterministic.

Above we have defined randomized PN algorithms. We can now extend the definitions in a natural manner to define randomized algorithms in the LOCAL model (add unique identifiers) and randomized algorithms in the CONGEST model (add unique identifiers and limit the size of the messages).

6.2 Probabilistic Analysis

In randomized algorithms, performance guarantees are typically probabilistic. For example, we may claim that algorithm *A stops in time T with probability p*.

Note that all probabilities here are over the random choices in the state transitions. We do not assume that our network or the local inputs are chosen randomly; we still require that the algorithm performs well with worst-case inputs. For example, if we claim that algorithm A solves problem Π on graph family $\mathscr{F}$ in time $T(n)$ with probability p, then we can take *any* graph $G \in \mathscr{F}$ and *any* port-numbered network N with G as its underlying graph, and we guarantee that with probability at least p the execution of A in N stops in time $T(n)$ and produces a correct output $g \in \Pi(G)$; as usual, n is the number of nodes in the network.

We may occasionally want to emphasize the distinction between "Monte Carlo" and "Las Vegas" type algorithms:

- Monte Carlo: Algorithm A always stops in time $T(n)$; the output is a correct solution to problem Π with probability p.
- Las Vegas: Algorithm A stops in time $T(n)$ with probability p; when it stops, the output is always a correct solution to problem Π.

However, Monte Carlo algorithms are not as useful in the field of distributed computing as they are in the context of classical centralized algorithms. In centralized algorithms, we can usually take a Monte Carlo

algorithm and just run it repeatedly until it produces a feasible solution; hence we can turn a Monte Carlo algorithm into a Las Vegas algorithm. This is not necessarily the case with distributed algorithms: verifying the output of an algorithm may require global information on the entire output, and gathering such information may take a long time. In this chapter, we will mainly focus on Las Vegas algorithms, i.e., algorithms that are always correct but may occasionally be slow, but in the exercises we will also encounter Monte Carlo algorithms.

6.3 With High Probability

We will use the word *failure* to refer to the event that the algorithm did not meet its guarantees—in the case of a Las Vegas algorithm, it did not stop in time $T(n)$, and in the case of Monte Carlo algorithms, it did not produce a correct output. The word *success* refers to the opposite case.

Usually we want to show that the probability of a failure is negligible. In computer science, we are usually interested in asymptotic analysis, and hence in the context of randomized algorithms, it is convenient if we can show that the success probability approaches 1 when n increases. Even better, we would like to let the user of the algorithm choose how quickly the success probability approaches 1.

This idea is captured in the phrase "*with high probability*" (commonly abbreviated *w.h.p.*). Please note that this phrase is not a vague subjective statement but it carries a precise mathematical meaning: it refers to the success probability of $1 - 1/n^c$, where we can choose any constant $c > 0$. (Unfortunately, different sources use slightly different definitions; for example, it may also refer to the success probability of $1 - O(1)/n^c$ for any constant $c > 0$.)

In our context, we say that algorithm A solves problem Π on graph family $\mathscr{F}$ in time $O(T(n))$ *with high probability* if the following holds:

- I can choose any constant $c > 0$. Algorithm A may depend on this constant.

- Then if I run A in any network N that has its underlying graph in $\mathscr{F}$, the algorithm will stop in time $O(T(n))$ with probability at least $1 - 1/n^c$, and the output is a feasible solution to problem Π.

Note that the $O(\cdot)$ notation in the running time is used to hide the dependence on c. This is a crucial point. For example, it would not make much sense to say that the running time is at most $\log n$ with probability $1 - 1/n^c$ for any constant $c > 0$. However, it is perfectly reasonable to say that the running time is, e.g., at most $c \log n$ or $2^c \log n$ or simply $O(\log n)$ with probability $1 - 1/n^c$ for any constant $c > 0$.

6.4 Randomized Coloring in Bounded-Degree Graphs

In Chapter 4 we presented a *deterministic* algorithm that finds a $(\Delta + 1)$-coloring in a graph of maximum degree Δ. In this section, we will design a *randomized* algorithm that solves the same problem. The running times are different:

- the deterministic algorithm runs in $O(\Delta + \log^* n)$ rounds.
- the randomized algorithm runs in $O(\log n)$ rounds with high probability.

Hence for large values of Δ, the randomized algorithm can be much faster.

6.4.1 Algorithm Idea

A running time of $O(\log n)$ is very typical for a randomized distributed algorithm. Often randomized algorithms follow the strategy that in each step each node picks a value randomly from some probability distribution. If the value conflicts with the values of the neighbors, the node will try again next time; otherwise the node outputs the current value and stops. If we can prove that each node stops in each round with a constant probability, we can prove that after $\Theta(\log n)$ all nodes have stopped

w.h.p. This is precisely what we saw in the analysis of the randomized path-coloring algorithm in Section 1.5.

However, adapting the same strategy to graphs of maximum degree Δ requires some thought. If each node just repeatedly tries to pick a random color from $\{1, 2, \ldots, \Delta+1\}$, the success probability may be fairly low for large values of Δ.

Therefore we will adopt a strategy in which nodes are slightly less aggressive. Nodes will first randomly choose whether they are *active* or *passive* in this round; each node is passive with probability $1/2$. Only active nodes will try to pick a random color among those colors that are not yet used by their neighbors.

Informally, the reason why this works well is the following. Assume that we have a node v with d neighbors that have not yet stopped. Then there are at least $d+1$ colors among which v can choose whenever it is active. If all of the d neighbors were also active and if they happened to pick distinct colors, we would have only a

$$\frac{1}{d+1}$$

chance of picking a color that is not used by any of the neighbors. However, in our algorithm on average only $d/2$ neighbors are active. If we have at most $d/2$ active neighbors, we will succeed in picking a free color with probability at least

$$\frac{d+1-d/2}{d+1} > \frac{1}{2},$$

regardless of what the active neighbors do.

6.4.2 Algorithm

Let us now formalize the algorithm. For each node u, let

$$C(u) = \{1, 2, \ldots, \deg_G(u)+1\}$$

be the *color palette* of the node; node u will output one of the colors of $C(u)$.

In the algorithm, node u maintains the following variables:

- State $s(u) \in \{0, 1\}$
- Color $c(u) \in \{\bot\} \cup C(u)$.

Initially, $s(u) \leftarrow 1$ and $c(u) \leftarrow \bot$. When $s(u) = 1$ and $c(u) \neq \bot$, node u stops and outputs color $c(u)$.

In each round, node u always sends $c(u)$ to each port. The incoming messages are processed as follows, depending on the current state of the node:

- $s(u) = 1$ and $c(u) \neq \bot$:
  - This is a stopping state; ignore incoming messages.
- $s(u) = 1$ and $c(u) = \bot$:
  - Let $M(u)$ be the set of messages received.
  - Let $F(u) = C(u) \setminus M(u)$ be the set of *free colors*.
  - With probability $1/2$, set $c(u) \leftarrow \bot$; otherwise choose a $c(u) \in F(u)$ uniformly at random.
  - Set $s(u) \leftarrow 0$.
- $s(u) = 0$:
  - Let $M(u)$ be the set of messages received.
  - If $c(u) \in M(u)$, set $c(u) \leftarrow \bot$.
  - Set $s(u) \leftarrow 1$.

Informally, the algorithm proceeds as follows. For each node u, its state $s(u)$ alternates between 1 and 0:

- When $s(u) = 1$, the node either decides to be *passive* and sets $c(u) = \bot$, or it decides to be *active* and picks a random color $c(u) \in F(u)$. Here $F(u)$ is the set of colors that are not yet used by any of the neighbors that are stopped.

- When $s(u) = 0$, the node *verifies* its choice. If the current color $c(u)$ conflicts with one of the neighbors, we go back to the initial state $s(u) \leftarrow 1$ and $c(u) \leftarrow \perp$. However, if we were lucky and managed to pick a color that does not conflict with any of our neighbors, we keep the current value of $c(u)$ and switch to the stopping state.

6.4.3 Analysis

It is easy to see that if the algorithm stops, then the output is a proper $(\Delta + 1)$-coloring of the underlying graph. Let us now analyze how long it takes for the nodes to stop.

In the analysis, we will write $s_t(u)$ and $c_t(u)$ for values of variables $s(u)$ and $c(u)$ after round $t = 0, 1, \ldots$, and $M_t(u)$ and $F_t(u)$ for the values of $M(u)$ and $F(u)$ during round $t = 1, 2, \ldots$. We also write

$$K_t(u) = \{v \in V : \{u, v\} \in E,\ s_{t-1}(v) = 1, c_{t-1}(v) = \perp\}$$

for the set of *competitors* of node u during round $t = 1, 3, 5, \ldots$; these are the neighbors of u that have not yet stopped.

First, let us prove that with probability at least $1/4$, a running node succeeds in picking a color that does not conflict with any of its neighbors.

Lemma 6.1. *Fix a node $u \in V$ and time $t = 1, 3, 5, \ldots$. Assume that $s_{t-1}(u) = 1$ and $c_{t-1}(u) = \perp$, i.e., u has not stopped before round t. Then with probability at least $1/4$, we have $s_{t+1}(u) = 1$ and $c_{t+1}(u) \neq \perp$, i.e., u will stop after round $t + 1$.*

Proof. Let $f = |F_t(u)|$ be the number of free colors during round t, and let $k = |K_t(u)|$ be the number of competitors during round t. Note that $f \geq k + 1$, as the size of the palette is one larger than the number of neighbors.

Let us first study the case that u is active. As we have got f free colors, for any given color $x \in \mathbb{N}$ we have

$$\Pr\left[c_t(u) = x \;\middle|\; c_t(u) \neq \perp\right] \leq 1/f.$$

In particular, this holds for any color $x = c_t(v)$ chosen by any active competitor $v \in K_t(u)$:

$$\Pr\big[c_t(u) = c_t(v) \;\big|\; c_t(u) \neq \perp,\ c_t(v) \neq \perp\big] \leq 1/f.$$

That is, we conflict with an active competitor with probability at most $1/f$. Naturally, we cannot conflict with a passive competitor:

$$\Pr\big[c_t(u) = c_t(v) \;\big|\; c_t(u) \neq \perp,\ c_t(v) = \perp\big] = 0.$$

As a competitor is active with probability

$$\Pr\big[c_t(v) \neq \perp\big] = 1/2,$$

and the random variables $c_t(u)$ and $c_t(v)$ are independent, the probability that we conflict with a given competitor $v \in K_t(u)$ is

$$\Pr\big[c_t(u) = c_t(v) \;\big|\; c_t(u) \neq \perp\big] \leq \frac{1}{2f}.$$

By the union bound, the probability that we conflict with some competitor is

$$\Pr\big[c_t(u) = c_t(v) \text{ for some } v \in K_t(u) \;\big|\; c_t(u) \neq \perp\big] \leq \frac{k}{2f},$$

which is less than $1/2$ for all $k \geq 0$ and all $f \geq k+1$. Put otherwise, node u will avoid conflicts with probability

$$\Pr\big[c_t(u) \neq c_t(v) \text{ for all } v \in K_t(u) \;\big|\; c_t(u) \neq \perp\big] > \frac{1}{2}.$$

So far we have studied the conditional probabilities assuming that u is active. This happens with probability

$$\Pr\big[c_t(u) \neq \perp\big] = 1/2.$$

Therefore node u will stop after round $t+1$ with probability

$$\begin{aligned}&\Pr\big[c_{t+1}(u) \neq \perp\big] = \\ &\Pr\big[c_t(u) \neq \perp \text{ and } c_t(u) \neq c_t(v) \text{ for all } v \in K_t(u)\big] > 1/4.\end{aligned}$$ □

Now we can continue with the same argument as what we used in Section 1.5 to analyze the running time. Fix a constant $c > 0$. Define

$$T(n) = 2(c+1)\log_{4/3} n.$$

We will prove that the algorithm stops in $T(n)$ rounds. First, let us consider an individual node. Note the exponent $c+1$ instead of c in the statement of the lemma; this will be helpful later.

Lemma 6.2. *Fix a node $u \in V$. The probability that u has not stopped after $T(n)$ rounds is at most $1/n^{c+1}$.*

Proof. By Lemma 6.1, if node u has not stopped after round $2i$, it will stop after round $2i+2$ with probability at least $1/4$. Hence the probability that it has not stopped after $T(n)$ rounds is at most

$$(3/4)^{T(n)/2} = \frac{1}{(4/3)^{(c+1)\log_{4/3} n}} = \frac{1}{n^{c+1}}.$$ □

Now we are ready to analyze the time until all nodes stop.

Theorem 6.3. *The probability that all nodes have stopped after $T(n)$ rounds is at least $1 - 1/n^c$.*

Proof. Follows from Lemma 6.2 by the union bound. □

Note that $T(n) = O(\log n)$ for any constant c. Hence we conclude that the algorithm stops in $O(\log n)$ rounds with high probability, and when it stops, it outputs a vertex coloring with $\Delta + 1$ colors.

6.5 Quiz

Consider a cycle with 10 nodes, and label the nodes with a random permutation of the numbers $1, 2, \ldots, 10$ (uniformly at random). A node is a *local maximum* if its label is larger than the labels of its two neighbors. Let X be the number of local maxima. What is the expected value of X?

6.6 Exercises

Exercise 6.1 (larger palette)**.** Assume that we have a graph without any isolated nodes. We will design a graph-coloring algorithm A that is a bit easier to understand and analyze than the algorithm of Section 6.4. In algorithm A, each node u proceeds as follows until it stops:

- Node u picks a color $c(u)$ from $\{1, 2, \ldots, 2d\}$ uniformly at random; here d is the degree of node u.
- Node u compares its value $c(u)$ with the values of all neighbors. If $c(u)$ is different from the values of its neighbors, u outputs $c(u)$ and stops.

Present this algorithm in a formally precise manner, using the state-machine formalism. Analyze the algorithm, and prove that it finds a 2Δ-coloring in time $O(\log n)$ with high probability.

Exercise 6.2 (unique identifiers)**.** Design a randomized PN algorithm A that solves the following problem in $O(1)$ rounds:

- As input, all nodes get value $|V|$.
- Algorithm outputs a labeling $f : V \to \{1, 2, \ldots, \chi\}$ for some $\chi = |V|^{O(1)}$.
- With high probability, $f(u) \neq f(v)$ for all nodes $u \neq v$.

Analyze your algorithm and prove that it indeed solves the problem correctly.

In essence, algorithm A demonstrates that we can use randomness to construct unique identifiers, assuming that we have some information on the size of the network. Hence we can take any algorithm B designed for the LOCAL model, and combine it with algorithm A to obtain a PN algorithm B' that solves the same problem as B (with high probability).

▷ *hint A*

Exercise 6.3 (large independent sets)**.** Design a randomized PN algorithm A with the following guarantee: in any graph $G = (V, E)$ of

maximum degree Δ, algorithm A outputs an independent set I such that the *expected* size of the I is $|V|/O(\Delta)$. The running time of the algorithm should be $O(1)$. You can assume that all nodes know Δ.

▷ *hint B*

Exercise 6.4 (max cut problem)**.** Let $G = (V, E)$ be a graph. A *cut* is a function $f : V \to \{0, 1\}$. An edge $\{u, v\} \in E$ is a *cut edge* in f if $f(u) \neq f(v)$. The *size* of cut f is the number of cut edges, and a *maximum cut* is a cut of the largest possible size.

(a) Prove: If $G = (V, E)$ is a bipartite graph, then a maximum cut has $|E|$ edges.

(b) Prove: If $G = (V, E)$ has a cut with $|E|$ edges, then G is bipartite.

(c) Prove: For any $\alpha > 1/2$, there exists a graph $G = (V, E)$ in which the maximum cut has fewer than $\alpha|E|$ edges.

▷ *hint C*

Exercise 6.5 (max cut algorithm)**.** Design a randomized PN algorithm A with the following guarantee: in any graph $G = (V, E)$, algorithm A outputs a cut f such that the *expected* size of cut f is at least $|E|/2$. The running time of the algorithm should be $O(1)$.

Note that the analysis of algorithm A also implies that for any graph there *exists* a cut with at least $|E|/2$.

▷ *hint D*

Exercise 6.6 (maximal independent sets)**.** Design a randomized PN algorithm that finds a maximal independent set in time $O(\Delta + \log n)$ with high probability.

▷ *hint E*

⋆ **Exercise 6.7** (maximal independent sets quickly)**.** Design a randomized distributed algorithm that finds a maximal independent set in time $O(\log n)$ with high probability.

▷ *hint F*

6.7 Bibliographic Notes

Algorithm of Section 6.4 and the algorithm of Exercise 6.1 are from Barenboim and Elkin's book [1, Section 10.1].

6.8 Bibliography

[1] Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Morgan & Claypool, 2013. doi:10.2200/S00520ED1V01Y201307DCT011.

6.9 Hints

A. Pick the labels randomly from a sufficiently large set; this takes 0 communication rounds.

B. Each node u picks a random number $f(u)$. Nodes that are local maxima with respect to the labeling f will join I.

C. For the last part, consider a complete graph with a sufficiently large number of nodes.

D. Each node chooses an output 0 or 1 uniformly at random and stops; this takes 0 communication rounds. To analyze the algorithm, prove that each edge is a cut edge with probability $1/2$.

E. Use the randomized coloring algorithm.

F. Look up "Luby's algorithm".