Juho Hirvonen and Jukka Suomela:
Distributed Algorithms 2020
https://jukkasuomela.fi/da2020/ · December 10, 2020

Chapter 3
PN Model: Port Numbering

Now that we have introduced the essential graph-theoretic concepts, we are ready to define what a "distributed algorithm" is. In this chapter, we will study one variant of the theme: deterministic distributed algorithms in the "port-numbering model". We will use the abbreviation PN for the port-numbering model, and we will also use the term "PN-algorithm" to refer to deterministic distributed algorithms in the port-numbering model. For now, everything will be deterministic—randomized algorithms will be discussed in later chapters.

3.1 Introduction

The basic idea of the PN model is best explained through an example. Suppose that I claim the following:

- *A* is a deterministic distributed algorithm that finds a 2-approximation of a minimum vertex cover in the port-numbering model.

Or, in brief:

- *A* is a PN-algorithm for finding a 2-approximation of a minimum vertex cover.

Informally, this entails the following:

(a) We can take any simple undirected graph $G = (V, E)$.

(b) We can then put together a computer network N with the same structure as G. A node $v \in V$ corresponds to a computer in N, and an edge $\{u, v\} \in E$ corresponds to a communication link between the computers u and v.

(c) Communication takes place through communication ports. A node of degree d corresponds to a computer with d ports that are labeled with numbers $1, 2, \ldots, d$ in an arbitrary order.

(d) Each computer runs a copy of the same deterministic algorithm A. All nodes are identical; initially they know only their own degree (i.e., the number of communication ports).

(e) All computers are started simultaneously, and they follow algorithm A synchronously in parallel. In each synchronous communication round, all computers in parallel

   (1) send a message to each of their ports,
   (2) wait while the messages are propagated along the communication channels,
   (3) receive a message from each of their ports, and
   (4) update their own state.

(f) After each round, a computer can stop and announce its *local output*: in this case the local output is either 0 or 1.

(g) We require that all nodes eventually stop—the *running time* of the algorithm is the number of communication rounds it takes until all nodes have stopped.

(h) We require that

$$C = \{ v \in V : \text{computer } v \text{ produced output } 1 \}$$

is a feasible vertex cover for graph G, and its size is at most 2 times the size of a minimum vertex cover.

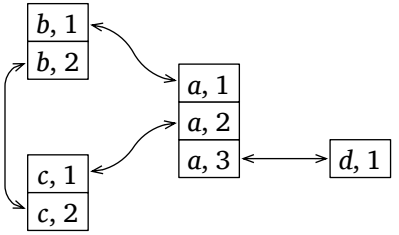


Figure 3.1: A port-numbered network $N = (V, P, p)$. There are four nodes, $V = \{a, b, c, d\}$; the degree of node a is 3, the degrees of nodes b and c are 2, and the degree of node d is 1. The connection function p is illustrated with arrows—for example, $p(a, 3) = (d, 1)$ and conversely $p(d, 1) = (a, 3)$. This network is simple.

Sections 3.2 and 3.3 will formalize this idea.

3.2 Port-Numbered Network

A *port-numbered network* is a triple $N = (V, P, p)$, where V is the set of *nodes*, P is the set of *ports*, and $p \colon P \to P$ is a function that specifies the *connections* between the ports. We make the following assumptions:

(a) Each port is a pair (v, i) where $v \in V$ and $i \in \{1, 2, \dots\}$.

(b) The connection function p is an involution, that is, for any port $x \in P$ we have $p(p(x)) = x$.

See Figures 3.1 and 3.2 for illustrations.

3.2.1 Terminology

If $(v, i) \in P$, we say that (v, i) is the port number i in node v. The *degree* $\deg_N(v)$ of a node $v \in V$ is the number of ports in v, that is, $\deg_N(v) = |\{\, i \in \mathbb{N} : (v, i) \in P \,\}|$.

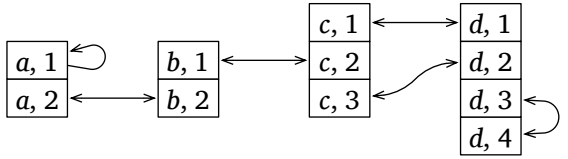


Figure 3.2: A port-numbered network $N = (V, P, p)$. There is a loop at node a, as $p(a, 1) = (a, 1)$, and another loop at node d, as $p(d, 3) = (d, 4)$. There are also multiple connections between c and d. Hence the network is not simple.

Unless otherwise mentioned, we assume that the port numbers are *consecutive*: for each $v \in V$ there are ports $(v, 1), (v, 2), \ldots, (v, \deg_N(v))$ in P.

We use the shorthand notation $p(v, i)$ for $p((v, i))$. If $p(u, i) = (v, j)$, we say that port (u, i) is *connected* to port (v, j); we also say that port (u, i) is connected to node v, and that node u is connected to node v.

If $p(v, i) = (v, j)$ for some j, we say that there is a *loop* at v—note that we may have $i = j$ or $i \neq j$. If $p(u, i_1) = (v, j_1)$ and $p(u, i_2) = (v, j_2)$ for some $u \neq v$, $i_1 \neq i_2$, and $j_1 \neq j_2$, we say that there are *multiple connections* between u and v. A port-numbered network $N = (V, P, p)$ is *simple* if there are no loops or multiple connections.

3.2.2 Underlying Graph

For a simple port-numbered network $N = (V, P, p)$ we define the *underlying graph* $G = (V, E)$ as follows: $\{u, v\} \in E$ if and only if u is connected to v in network N. Observe that $\deg_G(v) = \deg_N(v)$ for all $v \in V$. See Figure 3.3 for an illustration.

3.2.3 Encoding Input and Output

In a distributed system, nodes are the active elements: they can read input and produce output. Hence we will heavily rely on *node labelings*: we can directly associate information with each node $v \in V$.

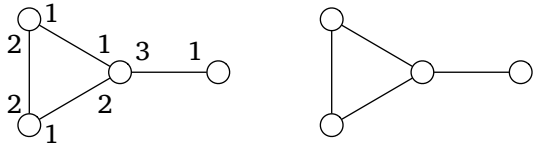


Figure 3.3: (a) An alternative drawing of the simple port-numbered network N from Figure 3.1. (b) The underlying graph G of N.

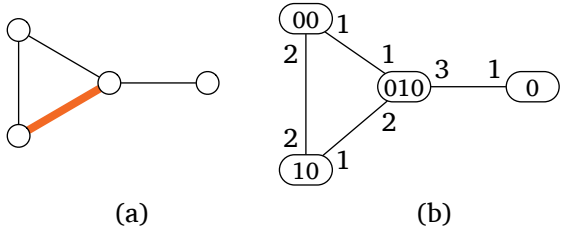


Figure 3.4: (a) A graph $G = (V, E)$ and a matching $M \subseteq E$. (b) A port-numbered network N; graph G is the underlying graph of N. The node labeling $f : V \to \{0,1\}^*$ is an encoding of matching M.

Assume that $N = (V, P, p)$ is a simple port-numbered network, and $G = (V, E)$ is the underlying graph of N. We show that a node labeling $f : V \to Y$ can be used to represent the following graph-theoretic structures; see Figure 3.4 for an illustration.

Node labeling $g : V \to X$**.** Trivial: we can choose $Y = X$ and $f = g$.

Subset of nodes $X \subseteq V$**.** We can interpret a subset of nodes as a node labeling $g : V \to \{0, 1\}$, where g is the indicator function of set X. That is, $g(v) = 1$ iff $v \in X$.

Edge labeling $g : E \to X$**.** For each node v, its label $f(v)$ encodes the values $g(e)$ for all edges e incident to v, in the order of increasing port numbers. More precisely, if v is a node of degree d, its label is a vector $f(v) \in X^d$. If $(v, j) \in P$ and $p(v, j) = (u, i)$, then element j of vector $f(v)$ is $g(\{u, v\})$.

Subset of edges $X \subseteq E$**.** We can interpret a subset of edges as an edge labeling $g : E \to \{0, 1\}$.

Orientation $H = (V, E')$**.** For each node v, its label $f(v)$ indicates which of the edges incident to v are outgoing edges, in the order of increasing port numbers.

It is trivial to compose the labelings. For example, we can easily construct a node labeling that encodes both a subset of nodes and a subset of edges.

3.2.4 Distributed Graph Problems

A *distributed graph problem* Π associates a set of solutions $\Pi(N)$ with each simple port-numbered network $N = (V, P, p)$. A *solution* $f \in \Pi(N)$ is a node labeling $f : V \to Y$ for some set Y of *local outputs*.

Using the encodings of Section 3.2.3, we can interpret all of the following as distributed graph problems: independent sets, vertex covers, dominating sets, matchings, edge covers, edge dominating sets, colorings,

edge colorings, domatic partitions, edge domatic partitions, factors, factorizations, orientations, and any combinations of these.

To make the idea more clear, we will give some more detailed examples.

(a) *Vertex cover*: $f \in \Pi(N)$ if f encodes a vertex cover of the underlying graph of N.

(b) *Minimal vertex cover*: $f \in \Pi(N)$ if f encodes a minimal vertex cover of the underlying graph of N.

(c) *Minimum vertex cover*: $f \in \Pi(N)$ if f encodes a minimum vertex cover of the underlying graph of N.

(d) 2-*approximation of minimum vertex cover*: $f \in \Pi(N)$ if f encodes a vertex cover C of the underlying graph of N; moreover, the size of C is at most two times the size of a minimum vertex cover.

(e) *Orientation*: $f \in \Pi(N)$ if f encodes an orientation of the underlying graph of N.

(f) 2-*coloring*: $f \in \Pi(N)$ if f encodes a 2-coloring of the underlying graph of N. Note that we will have $\Pi(N) = \varnothing$ if the underlying graph of N is not bipartite.

3.3 Distributed Algorithms in the Port-Numbering Model

We will now give a formal definition of a distributed algorithm in the port-numbering model. In essence, a distributed algorithm is a state machine (not necessarily a finite-state machine). To run the algorithm on a certain port-numbered network, we put a copy of the same state machine at each node of the network.

The formal definition of a distributed algorithm plays a similar role as the definition of a Turing machine in the study of non-distributed

algorithms. A formally rigorous foundation is necessary to study questions such as computability and computational complexity. However, we do not usually present algorithms as Turing machines, and the same is the case here. Once we become more familiar with distributed algorithms, we will use higher-level pseudocode to define algorithms and omit the tedious details of translating the high-level description into a state machine.

3.3.1 State Machine

A distributed algorithm A is a state machine that consists of the following components:

(i) Input_A is the set of *local inputs*,

(ii) States_A is the set of states,

(iii) $\text{Output}_A \subseteq \text{States}_A$ is the set of stopping states (*local outputs*),

(iv) Msg_A is the set of possible messages.

Moreover, for each possible degree $d \in \mathbb{N}$ we have the following functions:

(v) $\text{init}_{A,d} \colon \text{Input}_A \to \text{States}_A$ initializes the state machine,

(vi) $\text{send}_{A,d} \colon \text{States}_A \to \text{Msg}_A^d$ constructs outgoing messages,

(vii) $\text{receive}_{A,d} \colon \text{States}_A \times \text{Msg}_A^d \to \text{States}_A$ processes incoming messages.

We require that $\text{receive}_{A,d}(x, y) = x$ whenever $x \in \text{Output}_A$. The idea is that a node that has already stopped and printed its local output no longer changes its state.

3.3.2 Execution

Let A be a distributed algorithm, let $N = (V, P, p)$ be a port-numbered network, and let $f : V \to \text{Input}_A$ be a labeling of the nodes. A *state vector*

is a function $x : V \to \text{States}_A$. The *execution* of A on (N, f) is a sequence of state vectors $x_0, x_1, \ldots$ defined recursively as follows.

The initial state vector x_0 is defined by

$$x_0(u) = \text{init}_{A,d}(f(u)),$$

where $u \in V$ and $d = \deg_N(u)$.

Now assume that we have defined state vector x_{t-1}. Define $m_t : P \to \text{Msg}_A$ as follows. Assume that $(u, i) \in P$, $(v, j) = p(u, i)$, and $\deg_N(v) = \ell$. Let $m_t(u, i)$ be component j of the vector $\text{send}_{A,\ell}(x_{t-1}(v))$.

Intuitively, $m_t(u, i)$ is the message received by node u from port number i on round t. Equivalently, it is the message sent by node v to port number j on round t—recall that ports (u, i) and (v, j) are connected.

For each node $u \in V$ with $d = \deg_N(u)$, we define the message vector

$$m_t(u) = \big(m_t(u, 1), m_t(u, 2), \ldots, m_t(u, d)\big).$$

Finally, we define the new state vector x_t by

$$x_t(u) = \text{receive}_{A,d}\big(x_{t-1}(u), m_t(u)\big).$$

We say that algorithm A *stops in time* T if $x_T(u) \in \text{Output}_A$ for each $u \in V$. We say that A *stops* if A stops in time T for some finite T. If A stops in time T, we say that $g = x_T$ is the *output* of A, and $x_T(u)$ is the *local output* of node u.

3.3.3 Solving Graph Problems

Now we will define precisely what it means if we say that a distributed algorithm A solves a certain graph problem.

Let $\mathscr{F}$ be a family of simple undirected graphs. Let Π and Π' be distributed graph problems (see Section 3.2.4). We say that *distributed algorithm A solves problem Π on graph family $\mathscr{F}$ given Π'* if the following holds: assuming that

(a) $N = (V, P, p)$ is a simple port-numbered network,

(b) the underlying graph of N is in $\mathscr{F}$, and
(c) the input f is in $\Pi'(N)$,

the execution of algorithm A on (N, f) stops and produces an output $g \in \Pi(N)$. If A stops in time $T(|V|)$ for some function $T : \mathbb{N} \to \mathbb{N}$, we say that A solves the problem *in time* T.

Obviously, A has to be compatible with the encodings of Π and Π'. That is, each $f \in \Pi'(N)$ has to be a function of the form $f : V \to \text{Input}_A$, and each $g \in \Pi(N)$ has to be a function of the form $g : V \to \text{Output}_A$.

Problem Π' is often omitted. If A does not need the input f, we simply say that A *solves problem* Π *on graph family* $\mathscr{F}$. More precisely, in this case we provide a trivial input $f(v) = 0$ for each $v \in V$.

In practice, we will often specify $\mathscr{F}$, Π, Π', and T implicitly. Here are some examples of common parlance:

(a) *Algorithm A finds a maximum matching in any path graph*: here $\mathscr{F}$ consists of all path graphs; Π' is omitted; and Π is the problem of finding a maximum matching.

(b) *Algorithm A finds a maximal independent set in k-colored graphs in time k*: here $\mathscr{F}$ consists of all graphs that admit a k-coloring; Π' is the problem of finding a k-coloring; Π is the problem of finding a maximal independent set; and T is the constant function $T : n \mapsto k$.

3.4 Example: Coloring Paths

Recall the fast 3-coloring algorithm for paths from Section 1.3. We will now present the algorithm in a formally precise manner as a state machine. Let us start with the problem definition:

- $\mathscr{F}$ is the family of path graphs.
- Π is the problem of coloring graphs with 3 colors.
- Π' is the problem of coloring graphs with any number of colors.

We will present algorithm A that solves problem Π on graph family $\mathscr{F}$ given Π'. Note that in Section 1.3 we assumed that we have unique identifiers, but it is sufficient to assume that we have some graph coloring, i.e., a solution to problem Π'.

The set of local inputs is determined by what we assume as input:

$$\mathrm{Input}_A = \mathbb{Z}^+.$$

The set of stopping states is determined by the problem that we are trying to solve:

$$\mathrm{Output}_A = \{1, 2, 3\}.$$

In our algorithm, each node only needs to store one positive integer (the current color):

$$\mathrm{States}_A = \mathbb{Z}^+.$$

Messages are also integers:

$$\mathrm{Msg}_A = \mathbb{Z}^+.$$

Initialization is trivial: the initial state of a node is its color. Hence for all d we have

$$\mathrm{init}_{A,d}(x) = x.$$

In each step, each node sends its current color to each of its neighbors. As we assume that all nodes have degree at most 2, we only need to define $\mathrm{send}_{A,d}$ for $d \le 2$:

$$\begin{aligned}\mathrm{send}_{A,0}(x) &= ().\\ \mathrm{send}_{A,1}(x) &= (x).\\ \mathrm{send}_{A,2}(x) &= (x, x).\end{aligned}$$

The nontrivial part of the algorithm is hidden in the receive function. To define it, we will use the following auxiliary function that returns the smallest positive number not in X:

$$g(X) = \min(\mathbb{Z}^+ \setminus X).$$

Again, we only need to define $\text{receive}_{A,d}$ for degrees $d \leq 2$:

$$\text{receive}_{A,0}(x,()) = \begin{cases} g(\varnothing) & \text{if } x \notin \{1,2,3\}, \\ x & \text{otherwise.} \end{cases}$$

$$\text{receive}_{A,1}(x,(y)) = \begin{cases} g(\{y\}) & \text{if } x \notin \{1,2,3\} \\ & \text{and } x > y, \\ x & \text{otherwise.} \end{cases}$$

$$\text{receive}_{A,2}(x,(y,z)) = \begin{cases} g(\{y,z\}) & \text{if } x \notin \{1,2,3\} \\ & \text{and } x > y,\ x > z, \\ x & \text{otherwise.} \end{cases}$$

This algorithm does precisely the same thing as the algorithm that was described in pseudocode in Table 1.1. It can be verified that this algorithm indeed solves problem Π on graph family $\mathscr{F}$ given Π', in the sense that we defined in Section 3.3.3.

We will not usually present distributed algorithms in the low-level state-machine formalism. Typically we are happy with a higher-level presentation (e.g., in pseudocode), but it is important to understand that any distributed algorithm can be always translated into the state machine formalism.

In the next two sections we will give some non-trivial examples of PN-algorithms. We will give informal descriptions of the algorithms; in the exercises we will see how to translate these algorithms into the state machine formalism.

3.5 Example: Maximal Matching in Two-Colored Graphs

In this section we present a distributed *bipartite maximal matching* algorithm: it finds a maximal matching in 2-colored graphs. That is, $\mathscr{F}$ is the family of bipartite graphs, we are given a 2-coloring $f : V \to \{1,2\}$, and the algorithm will output an encoding of a maximal matching $M \subseteq E$.

3.5.1 Algorithm

In what follows, we say that a node $v \in V$ is *white* if $f(v) = 1$, and it is *black* if $f(v) = 2$. During the execution of the algorithm, each node is in one of the states

$$\{\,\mathsf{UR},\, \mathsf{MR}(i),\, \mathsf{US},\, \mathsf{MS}(i)\,\},$$

which stand for "unmatched and running", "matched and running", "unmatched and stopped", and "matched and stopped", respectively. As the names suggest, US and $\mathsf{MS}(i)$ are stopping states. If the state of a node v is $\mathsf{MS}(i)$ then v is matched with the neighbor that is connected to port i.

Initially, all nodes are in state UR. Each black node v maintains variables $M(v)$ and $X(v)$, which are initialized

$$M(v) \leftarrow \varnothing, \quad X(v) \leftarrow \{1, 2, \ldots, \deg(v)\}.$$

The algorithm is presented in Table 3.1; see Figure 3.5 for an illustration.

3.5.2 Analysis

The following invariant is useful in order to analyze the algorithm.

Lemma 3.1. *Assume that u is a white node, v is a black node, and $(u, i) = p(v, j)$. Then at least one of the following holds:*

(a) element j is removed from $X(v)$ before round $2i$,
(b) at least one element is added to $M(v)$ before round $2i$.

Proof. Assume that we still have $M(v) = \varnothing$ and $j \in X(v)$ after round $2i-2$. This implies that v is still in state UR, and u has not sent '*matched*' to v. In particular, u is in state UR or $\mathsf{MR}(i)$ after round $2i-2$. In the former case, u sends '*proposal*' to v on round $2i-1$, and j is added to $M(v)$ on round $2i-1$. In the latter case, u sends '*matched*' to v on round $2i-1$, and j is removed from $X(v)$ on round $2i-1$. □

Now it is easy to verify that the algorithm actually makes some progress and eventually halts.

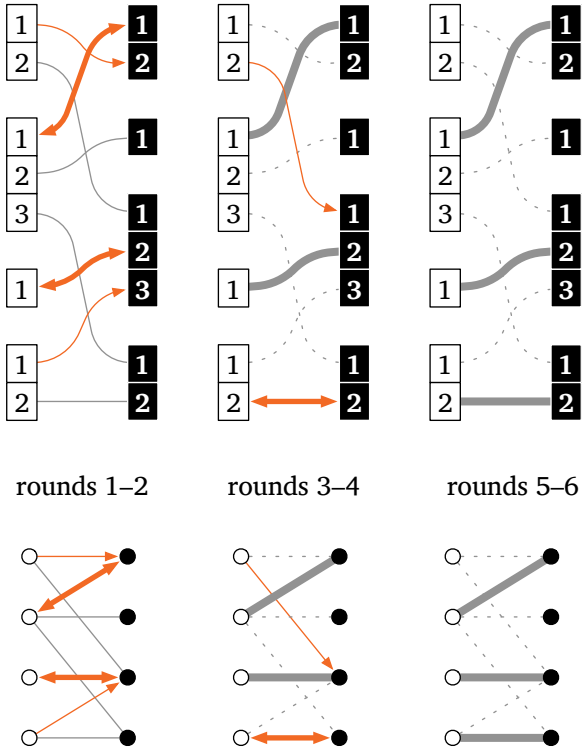


Figure 3.5: The bipartite maximal matching algorithm; the illustration shows the algorithm both from the perspective of the port-numbered network N and from the perspective of the underlying graph G. Arrows pointing right are proposals, and arrows pointing left are acceptances. Wide gray edges have been added to matching M.

Round $2k-1$*, white nodes:*

- State UR, $k \leq \deg_N(v)$: Send '*proposal*' to port (v,k).
- State UR, $k > \deg_N(v)$: Switch to state US.
- State MR(i): Send '*matched*' to all ports.
  Switch to state MS(i).

Round $2k-1$*, black nodes:*

- State UR: Read incoming messages.
  If we receive '*matched*' from port i, remove i from $X(v)$.
  If we receive '*proposal*' from port i, add i to $M(v)$.

Round $2k$*, black nodes:*

- State UR, $M(v) \neq \varnothing$: Let $i = \min M(v)$.
  Send '*accept*' to port (v,i). Switch to state MS(i).
- State UR, $X(v) = \varnothing$: Switch to state US.

Round $2k$*, white nodes:*

- State UR: Process incoming messages.
  If we receive '*accept*' from port i, switch to state MR(i).

Table 3.1: The bipartite maximal matching algorithm; here $k = 1, 2, \ldots$.

Lemma 3.2. *The bipartite maximal matching algorithm stops in time* $2\Delta + 1$*, where* Δ *is the maximum degree of* N*.*

Proof. A white node of degree d stops before or during round $2d + 1 \leq 2\Delta + 1$.

Now let us consider a black node v. Assume that we still have $j \in X(v)$ on round 2Δ. Let $(u, i) = p(v, j)$; note that $i \leq \Delta$. By Lemma 3.1, at least one element has been added to $M(v)$ before round 2Δ. In particular, v stops before or during round 2Δ. □

Moreover, the output is correct.

Lemma 3.3. *The bipartite maximal matching algorithm finds a maximal matching in any two-colored graph.*

Proof. Let us first verify that the output correctly encodes a matching. In particular, assume that u is a white node, v is a black node, and $p(u, i) = (v, j)$. We have to prove that u stops in state $\mathsf{MS}(i)$ if and only if v stops in state $\mathsf{MS}(j)$. If u stops in state $\mathsf{MS}(i)$, it has received an '*accept*' from v, and v stops in state $\mathsf{MS}(j)$. Conversely, if v stops in state $\mathsf{MS}(j)$, it has received a '*proposal*' from u and it sends an '*accept*' to u, after which u stops in state $\mathsf{MS}(i)$.

Let us then verify that M is indeed maximal. If this was not the case, there would be an unmatched white node u that is connected to an unmatched black node v. However, Lemma 3.1 implies that at least one of them becomes matched before or during round 2Δ. □

3.6 Example: Vertex Covers

We will now give a distributed *minimum vertex cover* 3*-approximation* algorithm; we will use the bipartite maximal matching algorithm from the previous section as a building block.

So far we have seen algorithms that assume something about the input (e.g., we are given a proper coloring of the network). The algorithm that we will see in this section makes no such assumptions. We can

run the minimum vertex cover 3-approximation algorithm in any port-numbered network, without any additional input. In particular, we do not need any kind of coloring, unique identifiers, or randomness.

3.6.1 Virtual 2-Colored Network

Let $N = (V, P, p)$ be a port-numbered network. We will construct another port-numbered network $N' = (V', P', p')$ as follows; see Figure 3.6 for an illustration. First, we double the number of nodes—for each node $v \in V$ we have two nodes v_1 and v_2 in V':

$$V' = \{ v_1, v_2 : v \in V \},$$
$$P' = \{ (v_1, i),\ (v_2, i) : (v, i) \in P \}.$$

Then we define the connections. If $p(u, i) = (v, j)$, we set

$$p'(u_1, i) = (v_2, j),$$
$$p'(u_2, i) = (v_1, j).$$

With these definitions we have constructed a network N' such that the underlying graph $G' = (V', E')$ is bipartite. We can define a 2-coloring $f' \colon V' \to \{1, 2\}$ as follows:

$$f'(v_1) = 1 \text{ and } f(v_2) = 2 \text{ for each } v \in V.$$

Nodes of color 1 are called *white* and nodes of color 2 are called *black*.

3.6.2 Simulation of the Virtual Network

Now N is our physical communication network, and N' is merely a mathematical construction. However, the key observation is that we can use the physical network N to efficiently *simulate* the execution of any distributed algorithm A on (N', f'). Each physical node $v \in V$ simulates nodes v_1 and v_2 in N':

(a) If v_1 sends a message m_1 to port (v_1, i) and v_2 sends a message m_2 to port (v_2, i) in the simulation, then v sends the pair (m_1, m_2) to port (v, i) in the physical network.

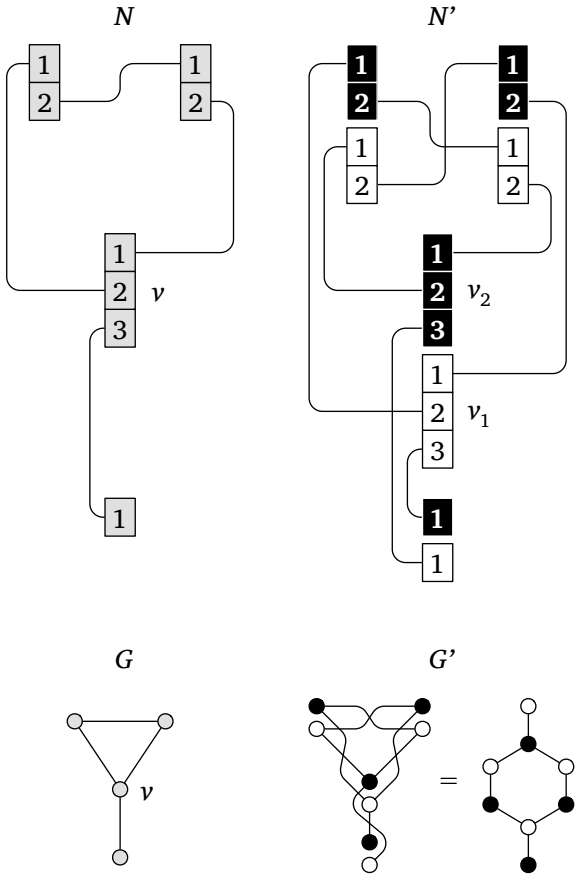


Figure 3.6: Construction of the virtual network N' in the minimum vertex cover 3-approximation algorithm.

(b) If v receives a pair (m_1, m_2) from port (v, i) in the physical network, then v_1 receives message m_2 from port (v_1, i) in the simulation, and v_2 receives message m_1 from port (v_2, i) in the simulation.

Note that we have here reversed the messages: what came from a white node is received by a black node and vice versa.

In particular, we can take the bipartite maximal matching algorithm of Section 3.5 and use the network N to simulate it on (N', f'). Note that network N is not necessarily bipartite and we do not have any coloring of N; hence we would not be able to apply the bipartite maximal matching algorithm on N.

3.6.3 Algorithm

Now we are ready to present the minimum vertex cover 3-approximation algorithm:

(a) Simulate the bipartite maximal matching algorithm in the virtual network N'. Each node v waits until both of its copies, v_1 and v_2, have stopped.

(b) Node v outputs 1 if at least one of its copies v_1 or v_2 becomes matched.

3.6.4 Analysis

Clearly the minimum vertex cover 3-approximation algorithm stops, as the bipartite maximal matching algorithm stops. Moreover, the running time is $2\Delta + 1$ rounds, where Δ is the maximum degree of N.

Let us now prove that the output is correct. To this end, let $G = (V, E)$ be the underlying graph of N, and let $G' = (V', E')$ be the underlying graph of N'. The bipartite maximal matching algorithm outputs a maximal matching $M' \subseteq E'$ for G'. Define the edge set $M \subseteq E$ as follows:

$$M = \big\{ \{u, v\} \in E : \{u_1, v_2\} \in M' \text{ or } \{u_2, v_1\} \in M' \big\}. \tag{3.1}$$

See Figure 3.7 for an illustration. Furthermore, let $C' \subseteq V'$ be the set of nodes that are incident to an edge of M' in G', and let $C \subseteq V$ be the set of nodes that are incident to an edge of M in G; equivalently, C is the set of nodes that output 1. We make the following observations.

(a) Each node of C' is incident to precisely one edge of M'.
(b) Each node of C is incident to one or two edges of M.
(c) Each edge of E' is incident to at least one node of C'.
(d) Each edge of E is incident to at least one node of C.

We are now ready to prove the main result of this section.

Lemma 3.4. *Set C is a 3-approximation of a minimum vertex cover of G.*

Proof. First, observation (d) above already shows that C is a vertex cover of G.

To analyze the approximation ratio, let $C^* \subseteq V$ be a vertex cover of G. By definition each edge of E is incident to at least one node of C^*; in particular, each edge of M is incident to a node of C^*. Therefore $C^* \cap C$ is a vertex cover of the subgraph $H = (C, M)$.

By observation (b) above, graph H has a maximum degree of at most 2. Set C consists of all nodes in H. We will then argue that any vertex cover C^* contains at least a fraction $1/3$ of the nodes in H; see Figure 3.8 for an example. Then it follows that C is at most 3 times as large as a minimum vertex cover.

To this end, let $H_i = (C_i, M_i)$, $i = 1, 2, \ldots, k$, be the connected components of H; each component is either a path or a cycle. Now $C_i^* = C^* \cap C_i$ is a vertex cover of H_i.

A node of C_i^* is incident to at most two edges of M_i. Therefore

$$|C_i^*| \geq |M_i|/2.$$

If H_i is a cycle, we have $|C_i| = |M_i|$ and

$$|C_i^*| \geq |C_i|/2.$$

If H_i is a path, we have $|M_i| = |C_i| - 1$. If $|C_i| \geq 3$, it follows that

$$|C_i^*| \geq |C_i|/3.$$

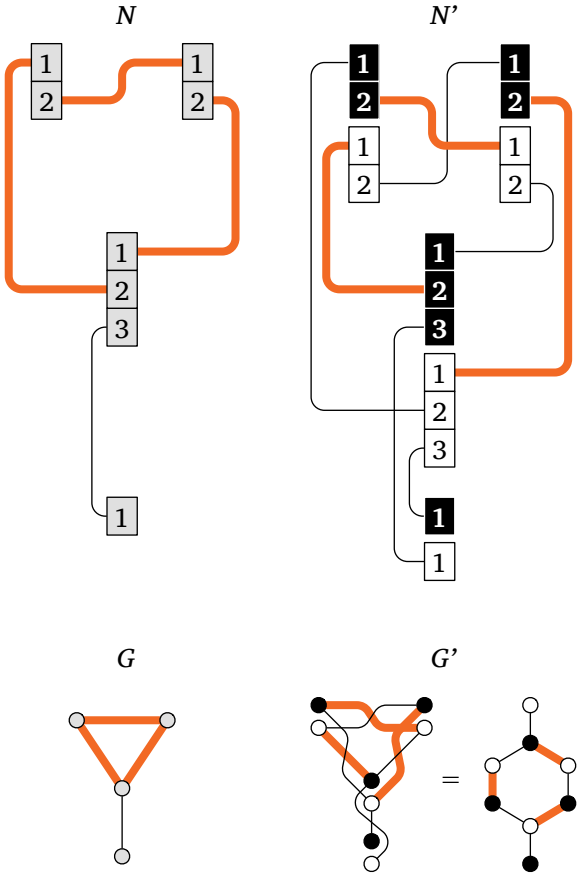
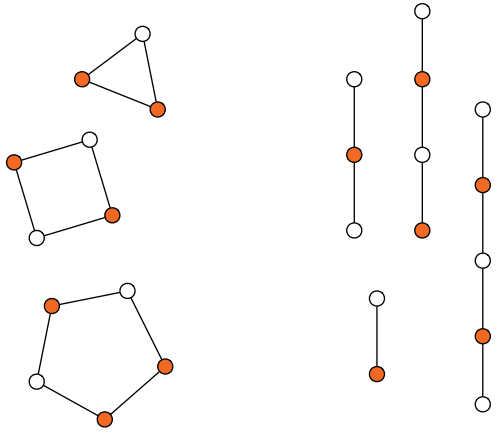


Figure 3.7: Set $M \subseteq E$ (left) and matching $M' \subseteq E'$ (right).



(a) (b)

Figure 3.8: (a) In a cycle with n nodes, any vertex cover contains at least $n/2$ nodes. (b) In a path with n nodes, any vertex cover contains at least $n/3$ nodes.

The only remaining case is a path with two nodes, in which case trivially $|C_i^*| \geq |C_i|/2$.

In conclusion, we have $|C_i^*| \geq |C_i|/3$ for each component H_i. It follows that

$$|C^*| \geq |C^* \cap C| = \sum_{i=1}^{k} |C_i^*| \geq \sum_{i=1}^{k} |C_i|/3 = |C|/3. \qquad \square$$

In summary, the minimum vertex cover algorithm finds a 3-approximation of a minimum vertex cover in any graph G. Moreover, if the maximum degree of G is small, the algorithm is fast: we only need $O(\Delta)$ rounds in a network of maximum degree Δ.

3.7 Quiz

Construct a simple port-numbered network $N = (V, P, p)$ and its underlying graph $G = (V, E)$ that has *as few nodes as possible* and that satisfies the following properties:

- Set E is nonempty.
- If $M \subseteq E$ consists of the edges $\{u, v\} \in E$ with $p(u, 1) = (v, 2)$, then M is a perfect matching of graph G.

Please answer by listing all elements of sets V, E, and P, and by listing all values of p. For example, you might specify a network with two nodes as follows: $V = \{1, 2\}$, $E = \{\{1, 2\}\}$, $P = \{(1, 1), (2, 1)\}$, $p(1, 1) = (2, 1)$, and $p(2, 1) = (1, 1)$.

3.8 Exercises

Exercise 3.1 (formalizing bipartite maximal matching)**.** Present the bipartite maximal matching algorithm from Section 3.5 in a formally precise manner, using the definitions of Section 3.3. Try to make Msg_A as small as possible.

Exercise 3.2 (formalizing vertex cover approximation)**.** Present the minimum vertex cover 3-approximation algorithm from Section 3.6 in a formally precise manner, using the definitions of Section 3.3. Try to make both Msg_A and States_A as small as possible.

▹ *hint A*

Exercise 3.3 (stopped nodes)**.** In the formalism of this chapter, a node that stops will repeatedly send messages to its neighbors. Show that this detail is irrelevant, and we can always re-write algorithms so that such messages are ignored. Put otherwise, a node that stops can also stop sending messages.

More precisely, assume that A is a distributed algorithm that solves problem Π on family $\mathscr{F}$ given Π' in time T. Show that there is another

algorithm A' such that (i) A' solves problem Π on family $\mathscr{F}$ given Π' in time $T + O(1)$, and (ii) in A' the state transitions never depend on the messages that are sent by nodes that have stopped.

Exercise 3.4 (more than two colors)**.** Design a distributed algorithm that finds a maximal matching in k-colored graphs. You can assume that k is a known constant.

Exercise 3.5 (analysis of vertex cover approximation)**.** Is the analysis of the minimum vertex cover 3-approximation algorithm tight? That is, is it possible to construct a network N such that the algorithm outputs a vertex cover that is exactly 3 times as large as the minimum vertex cover of the underlying graph of N?

⋆ **Exercise 3.6** (implementation)**.** Using your favorite programming language, implement a simulator that lets you play with distributed algorithms in the port-numbering model. Implement the algorithms for bipartite maximal matching and minimum vertex cover 3-approximation and try them out in the simulator.

⋆ **Exercise 3.7** (composition)**.** Assume that algorithm A_1 solves problem Π_1 on family $\mathscr{F}$ given Π_0 in time T_1, and algorithm A_2 solves problem Π_2 on family $\mathscr{F}$ given Π_1 in time T_2.

Is it always possible to design an algorithm A that solves problem Π_2 on family $\mathscr{F}$ given Π_0 in time $O(T_1 + T_2)$?

▷ *hint B*

3.9 Bibliographic Notes

The concept of a port numbering is from Angluin's [1] work. The bipartite maximal matching algorithm is due to Hańćkowiak et al. [2], and the minimum vertex cover 3-approximation algorithm is from a paper with Polishchuk [3].

3.10 Bibliography

[1] Dana Angluin. Local and global properties in networks of processors. In *Proc. 12th Annual ACM Symposium on Theory of Computing (STOC 1980)*, 1980. doi:10.1145/800141.804655.

[2] Michał Hańćkowiak, Michał Karoński, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. In *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998)*, 1998.

[3] Valentin Polishchuk and Jukka Suomela. A simple local 3-approximation algorithm for vertex cover. *Information Processing Letters*, 109(12):642–645, 2009. arXiv:0810.2175, doi:10.1016/j.ipl.2009.02.017.

3.11 Hints

A. For the purposes of the minimum vertex cover algorithm, it is sufficient to know which nodes are matched in the bipartite maximal matching algorithm—we do not need to know with whom they are matched.

B. This exercise is not trivial. If T_1 was a constant function $T_1(n) = c$, we could simply run A_1, and then start A_2 at time c, using the output of A_1 as the input of A_2. However, if T_1 is an arbitrary function of $|V|$, this strategy is not possible—we do not know in advance when A_1 will stop.