Chapter 2
Distributed Graph Algorithms

This chapter addresses three basic graph problems encountered in the context of distributed systems. These problems are (a) the computation of the shortest paths between a pair of processes where a positive length (or weight) is attached to each communication channel, (b) the coloring of the vertices (processes) of a graph in $\Delta + 1$ colors (where Δ is the maximal number of neighbors of a process, i.e., the maximal degree of a vertex when using the graph terminology), and (c) the detection of knots and cycles in a graph. As for the previous chapter devoted to graph traversal algorithms, an aim of this chapter is not only to present specific distributed graph algorithms, but also to show that their design is not always obtained from a simple extension of their sequential counterparts.

Keywords Distributed graph algorithm · Cycle detection · Graph coloring · Knot detection · Maximal independent set · Problem reduction · Shortest path computation

2.1 Distributed Shortest Path Algorithms

This section presents distributed algorithms which allow each process to compute its shortest paths to every other process in the system. These algorithms can be seen as "adaptations" of centralized algorithmic principles to the distributed context.

The notations are the same as in the previous chapter. Each process p_i has a set of neighbors denoted $neighbors_i$; if it exists, the channel connecting p_i and p_j is denoted $\langle i, j \rangle$. The communication channels are bidirectional (hence $\langle i, j \rangle$ and $\langle j, i \rangle$ denote the same channel). Moreover, the communication graph is connected and each channel $\langle i, j \rangle$ has a positive length (or weight) denoted $\ell g_i[j]$ (as $\langle i, j \rangle$ and $\langle j, i \rangle$ are the same channel, we have $\ell g_i[j] = \ell g_j[i]$).

2.1.1 A Distributed Adaptation of Bellman–Ford's Shortest Path Algorithm

Bellman–Ford's sequential algorithm computes the shortest paths from one predetermined vertex of a graph to every other vertex. It is an iterative algorithm based

M. Raynal, *Distributed Algorithms for Message-Passing Systems*,
DOI 10.1007/978-3-642-38123-2_2,

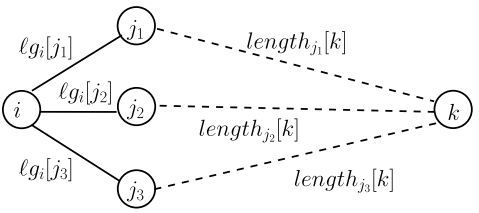

Fig. 2.1 Bellman–Ford's dynamic programming principle

on the dynamic programming principle. This principle and its adaptation to a distributed context are presented below.

Initial Knowledge and Local Variables Initially each process knows that there are n processes and the set of process identities is $\{1, \ldots, n\}$. It also knows its position in the communication graph (which is captured by the set $neighbors_i$). Interestingly, it will never learn more on the structure of this graph. From a local state point of view, each process p_i manages the following variables.

- As just indicated, $\ell g_i[j]$, for $j \in neighhbors_i$, denotes the length associated with the channel $\langle i, j \rangle$.
- $length_i[1..n]$ is an array such that $length_i[k]$ will contain the length of the shortest path from p_i to p_k. Initially, $length_i[i] = 0$ (and keeps that value forever) while $length_i[j] = +\infty$ for $j \neq i$.
- $routing\_to_i[1..n]$ is an array that is not used to compute the shortest paths from p_i to each other process. It constitutes the local result of the computation. More precisely, when the algorithm terminates, for any k, $1 \leq k \leq n$, $routing\_to_i[k] = j$ means that p_j is a neighbor of p_i on a shortest path to p_k, i.e., p_j is an optimal neighbor when p_i has to send information to p_k (where optimality is with respect to the length of the path from p_i to p_k).

Bellman–Ford Principle The dynamic programming principle on which the algorithm relies is the following. The local inputs at each process p_i are the values of the set $neighbors_i$ and the array $\ell g_i[neighbors_i]$. The output at each process p_i is the array $length_i[1..n]$. The algorithm has to solve the following set of equations (where the unknown variables are the arrays $length_i[1..n]$):

$$\forall i, k \in \{1, \ldots, n\}: \quad length_i[k] = \min_{j \in neighbors_i} \big(\ell g_i[j] + length_j[k]\big).$$

The meaning of this formula is depicted in Fig. 2.1 for a process p_i such that $neighbors_i = \{j_1, j_2, j_3\}$. Each dotted line from p_{j_x} to p_k, $1 \leq x \leq 3$, represents the shortest path joining p_{j_x} to p_k and its length is $length_{j_x}[k]$. The solution of this set of equations is computed asynchronously and iteratively by the n processes, each process p_i computing successive approximate values of its local array $length_i[1..n]$ until it stabilizes at its final value.

The Algorithm The algorithm is described in Fig. 2.2. At least one process p_i has to receive the external message START() in order to launch the algorithm. It

```
when START() is received do
(1)  for each j ∈ neighbors_i do send UPDATE(length_i) to p_j end for.

when UPDATE(length) is received from p_j do
(2)  updated_i ← false;
(3)  for each k ∈ {1, ..., n} \ {i} do
(4)     if (length_i[k] > ℓg_i[j] + length[k])
(5)        then length_i[k] ← ℓg_i[j] + length[k];
(6)             routing_to_i[k] ← j;
(7)             updated_i ← true
(8)     end if
(9)  end for;
(10) if (updated_i)
(11)    then for each j ∈ neighbors_i do send UPDATE(length_i) to p_j end for
(12) end if.
```

Fig. 2.2 A distributed adaptation of Bellman–Ford's shortest path algorithm (code for p_i)

sends then to each of its neighbors the message UPDATE($length_i$) which describes its current local state as far as the computation of the length of its shortest paths to each other process is concerned.

When a process p_i receive a message UPDATE($length$) from one of its neighbors p_j, it applies the forward/discard strategy introduced in Chap. 1. To that end, p_i first strives to improve its current approximation of its shortest paths to any destination process (lines 3–9). Then, if p_i has discovered shorter paths than the ones it knew before, p_i sends its new current local state to each of its neighbors (lines 10–12). If its local state (captured by the array $length_i[1..n]$) has not been modified, p_i does not send a message to its neighbors.

Termination While there is a finite time after which the arrays $length_i[1..n]$ and $routing\_to_i[1..n]$, $1 \leq i \leq n$, have obtained their final values, no process ever learns when this time has occurred.

Adding Synchronization in Order that Each Process Learns Termination The algorithm described in Fig. 2.3 allows each process p_i not only to compute the shortest paths but also to learn that it knows them (i.e., learn that its local arrays $length_i[1..n]$ and $routing\_to_i[1..n]$ have converged to their final values).

This algorithm is synchronous: the processes execute a sequence of synchronous rounds, and rounds are given for free: they belong to the computation model. During each round r, in addition to local computation, each process sends a message to and receives a message from each of its neighbors. The important synchrony property lies in the fact that a message sent by a process p_i to a neighbor p_j at round r is received and processed by p_j during the very same round r. The progress of a round r to the round $r + 1$ is governed by the underlying system. (A general technique to simulate a synchronous algorithm on top of an asynchronous system will be described in Chap. 9.)

```
when r = 1, 2, ..., D do
begin synchronous round
(1)  for each j ∈ neighbors_i do send UPDATE(length_i) to p_j end for;
(2)  for each j ∈ neighbors_i do receive UPDATE(length_j) from p_j end for;
(3)  for each k ∈ {1, ..., n} \ {i} do
(4)     let length_ik1 = min_{j∈neighbors_i} (ℓg_i[j] + length_j[k]);
(5)     if (length_ik < length_i[k]) then
(6)        length_i[k] ← length_ik;
(7)        routing_to_i[k] ← a neighbor j that realizes the previous minimum
(8)     end if
(9)  end for
end synchronous round.
```

Fig. 2.3 A distributed synchronous shortest path algorithm (code for p_i)

The algorithm considers that the diameter D of the communication graph is known by the processes (let us remember that the diameter is the number of channels separating the two most distant processes). If D is not explicitly known, it can be replaced by an upper bound, namely the value $(n-1)$.

The text of the algorithm is self-explanatory. There is a strong connection between the current round number and the number of channels composing the paths from which p_i learns information. Let us consider a process p_i at the end of a round r.

- When $r < D$, p_i knows the shortest path from itself to any other process p_k, which is composed of at most r channels. Hence, this length to p_k is not necessarily the shortest one.
- Differently, when it terminates round $r = D$, p_i has computed both (a) the shortest lengths from it to all the other processes and (b) the corresponding appropriate routing neighbors.

2.1.2 A Distributed Adaptation of Floyd–Warshall's Shortest Paths Algorithm

Floyd–Warshall's algorithm is a sequential algorithm that computes the shortest paths between any two vertices of a non-directed graph. This section presents an adaptation of this algorithm to the distributed context. This adaptation is due to S. Toueg (1980). As previously, to make the presentation easier, we consider that the graph (communication network) is connected and the length of each edge of the graph (communication channel) is a positive number. (Actually, the algorithm works when edges have negative lengths as long as no cycle has a negative length.)

Floyd–Warshall's Sequential Algorithm Let $LENGTH[1..n, 1..n]$ be a matrix such that, when the algorithm terminates, $LENGTH[i, j]$ represents the length of the shortest path from p_i to p_j. Initially, for any i, $LENGTH[i, i] = 0$, for any pair (i, j)

```
(1)  for pv from 1 to n do
(2)      for i from 1 to n do
(3)          for j from 1 to n do
(4)              if LENGTH[i, pv] + LENGTH[pv, j] < LENGTH[i, j]
(5)                  then LENGTH[i, j] ← LENGTH[i, pv] + LENGTH[pv, j];
(6)                       routing_to_i[j] ← routing_to_i[pv]
(7)              end if
(8)          end for
(9)      end for
(10) end for.
```

Fig. 2.4 Floyd–Warshall's sequential shortest path algorithm

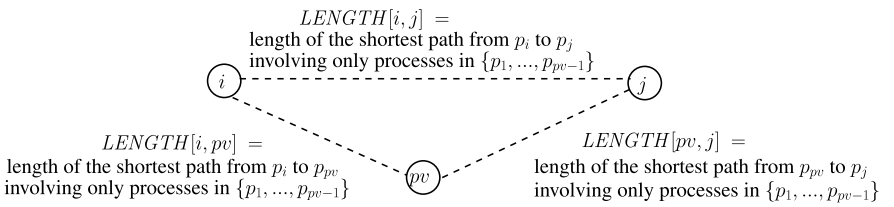

Fig. 2.5 The principle that underlies Floyd–Warshall's shortest paths algorithm

such that $j \in neighbors_i$, $LENGTH[i, j]$ is initialized to the length of the channel from p_i to p_j, and $LENGTH[i, j] = +\infty$ in the other cases. Moreover, for any i, the array $routing\_to_i[1..n]$ is such that $routing\_to_i[i] = i$, $routing\_to_i[j] = j$ for each $j \in neighbors_i$ and $routing\_to_i[j]$ is initially undefined for the other values of j.

The principle of Floyd–Warshall's algorithm is an iterative algorithm based on the following principle. For any process p_i, the algorithm computes first the shortest path from any process p_i to any process p_j that (if any) passes through process p_1. Then, it computes the shortest path from any process p_i to any process p_j among all the paths from p_i to p_j which pass only through processes in the set $\{p_1, p_2\}$. More generally, at the step pv of the iteration, the algorithm computes the shortest path from any process p_i to any process p_j among all the paths from p_i to p_j which are allowed to pass through the set of processes $\{p_1, \ldots, p_{pv}\}$. The text of the algorithm is given in Fig. 2.4. As we can see, the algorithm is made up of three nested **for** loops. The external one defines the processes (namely $p_1, \ldots, p_{pv}$) allowed to appear in current computation of the shortest from any process p_i to any process p_j. The process index pv is usually called the *pivot*.

The important feature of this sequential algorithm lies in the fact that, when computing the shortest path from p_i to p_j involving the communication channels connecting the processes in the set $\{p_1, \ldots, p_{pv}\}$, the variable $LENGTH[i, pv]$ contains the length of the shortest path from p_i to p_{pv} involving only the communication channels connecting the processes in the set $\{p_1, \ldots, p_{pv-1}\}$ (and similarly for the variable $LENGTH[pv, j]$). This is described in Fig. 2.5 when considering the computation of the shortest from p_i to p_j involving the processes $\{p_1, \ldots, p_{pv}\}$ (this constitutes the pvth iteration step of the external loop).

From a Sequential to a Distributed Algorithm As a distributed system has no central memory and communication is by message passing between neighbor processes, two issues have to be resolved to obtain a distributed algorithm. The first concerns the distribution of the data structures; the second the synchronization of processes so that they can correctly compute the shortest paths and the associated routing tables.

The array $LENGTH[1..n, 1..n]$ is split into n vectors such that each process p_i computes and maintains the value of $LENGTH[i, 1..n]$ in its local array $length_i[1..n]$. Moreover, as seen before, each process p_i computes the value of its routing local array $routing\_to_i[1..n]$. On the synchronization side, there are two issues:

- When p_i computes the value of $length_i[j]$ during the iteration step pv, process p_i locally knows the current values of $length_i[j]$ and $length_i[pv]$, but it has to obtain the current value of $length_{pv}[j]$ (see line 4 of Fig. 2.4).
- To obtain from p_{pv} a correct value for $length_{pv}[j]$, the processes must execute simultaneously the same iteration step pv. If a process p_i is executing an iteration step with the pivot value pv while another process p_k is simultaneously executing an iteration step with the pivot value $pv' \neq pv$, the values they obtain, respectively, from p_{pv} for $length_{pv}[j]$ and from $p_{pv'}$ $length_{pv'}[j]$ can be mutually inconsistent if these computations are done without an appropriate synchronization.

The Distributed Algorithm The algorithm is described in Fig. 2.6. The processes execute concurrently a loop where the index pv takes the successive values from 1 to n (line 1). If a process receives a message while it has not yet started executing its local algorithm, it locally starts the local algorithm before processing the message. As the communication graph is connected, it follows that, as soon as at least one process p_i starts its local algorithm, all the processes start theirs.

As indicated just previously, when the processes execute the iteration step pv, the process p_{pv} has to broadcast its local array $length_{pv}[1..n]$ so that each process p_i to try to improve its shortest distance to any process p_j as indicated in Fig. 2.5.

To this end, let us observe that if, at the pvth iteration of the loop, there is path from p_i to p_{pv} involving only processes in the set $\{p_1, \ldots, p_{pv-1}\}$, there is then a favorite neighbor to attain p_{pv}, namely the process whose index has been computed and saved in $routing\_to_i[pv]$. This means that, at the pvth iteration, the set of local variables $routing\_to_x[pv]$ of the processes p_x such that $length_x[pv] \neq +\infty$ define a tree rooted at p_{pv}.

The algorithm executed by the processes, which ensures a correct process coordination, follows from this observation. More precisely, a local algorithm is made up of three parts:

- Part 1: lines 1–6. A process p_i first sends a message to each of its neighbors p_k indicating if p_i is or not one of p_k's children in the tree rooted at p_{pv}. It then waits until it has received such a message from each of its neighbors.

  Then, p_i executes the rest of the code for the pvth iteration only if it has a chance to improve its shortest paths with the help of p_{pv}, i.e., if $length_i[pv] \neq +\infty$.

```
(1)  for pv from 1 to n do
(2)    for each k ∈ neighbors_i do
(3)      if (routing_to_i[pv] = k) then child ← yes else child ← no end if;
(4)      send CHILD(pv, child) to p_k
(5)    end for;
(6)    wait (a message CHILD(pv, −) received from each neighbor);
(7)    if (length_i[pv] ≠ +∞) then
(8)      if (pv ≠ i) then
(9)        wait (message PV_LENGTH (pv, pv_length[1..n]) from p_routing_to_i[pv])
(10)     end if;
(11)     for each k ∈ neighbors_i do
(12)       if (CHILD(pv, yes) received from p_k) then
(13)         if (pv = i) then send PV_LENGTH(pv, length_i[1..n]) to p_k
(14)                     else send PV_LENGTH(pv, pv_length[1..n]) to p_k
(15)         end if
(16)       end if
(17)     end for;
(18)     for j from 1 to n do
(19)       if length_i[pv] + pv_length[j] < length_i[j]
(20)         then length_i[j] ← length_i[pv] + pv_length[j];
(21)              routing_to_i[j] ← routing_to_i[pv]
(22)       end if
(23)     end for
(24)   end if
(25) end for.
```

Fig. 2.6 Distributed Floyd–Warshall's shortest path algorithm

- Part 2: lines 8–17. This part of the algorithm ensures that each process p_i such that $length_i[pv] \neq +\infty$ receives a copy of the array $length_{pv}[1..n]$ so that it can recompute the values of its shortest paths and the associated local routing table (which is done in Part 3).

  The broadcast of $length_{pv}[1..n]$ by p_{pv} is launched at line 13, where this process sends the message PV_LENGTH $(pv, length_{pv})$ to all its children in the tree whose it is the root. When it receives such a message carrying the value pv and the array $pv\_length[1..n]$ (line 9), a process p_i forwards it to its children in the tree rooted at p_{pv} (lines 12 and 14).
- Part 3: lines 18–23. Finally, a process p_i uses the array $pv\_length[1..n]$ it has received in order to improve its shortest paths that pass through the processes $p_1, \ldots, p_{pv}$.

Cost Let e be the number of communication channels. It is easy to see that, during each iteration, (a) at most two messages CHILD() are sent on each channel (one in each direction) and (b) at most $(n-1)$ messages PV_LENGTH() are sent. It follows that the number of messages is upper-bounded by $n(2e+n)$; i.e., the message complexity is $O(n^3)$. As far the size of messages is concerned, a message CHILD() carries a bit, while PV_LENGTH() carries n values whose size depends on the individual lengths associated with the communication channels.

```
(1) for i from 1 to n do
(2)   c ← 1;
(3)   while (COLOR[i] = ⊥) do
(4)     if (⋀_{j∈neighbors_i} COLOR[j] ≠ c) then COLOR[i] ← c else c ← c + 1 end if
(5)   end while
(6) end for.
```

Fig. 2.7 Sequential $(\Delta + 1)$-coloring of the vertices of a graph

Finally, there are n iteration steps, and each has $O(n)$ time complexity. Moreover, in the worst case, the processes starts the algorithm one after the other (a single process starts, which entails the start of another process, etc.). When summing up, it follows that the time complexity is upper-bounded by $O(n^2)$.

2.2 Vertex Coloring and Maximal Independent Set

2.2.1 On Sequential Vertex Coloring

Vertex Coloring An important graph problem, which is encountered when one has to model application-level problems, concerns vertex coloring. It consists in assigning a value (color) to each vertex such that (a) no two vertices which are neighbors have the same color, and (b) the number of colors is "reasonably small". When the number of colors has to be the smallest possible one, the problem is NP-complete.

Let Δ be the maximal degree of a graph (let us remember that, assuming a graph where any two vertices are connected by at most one edge, the degree of a vertex is the number of its neighbors). It is always possible to color the vertices of a graph in $\Delta + 1$ colors. This follows from the following simple reasoning by induction. The assertion is trivially true for any graph with at most Δ vertices. Then, assuming it is true for any graph made up of $n \geq \Delta$ vertices and whose maximal degree is at most Δ, let us add a new vertex to the graph. As (by assumption) the maximal degree of the graph is Δ, it follows that this new vertex has at most Δ neighbors. Hence, this vertex can be colored with the remaining color.

A Simple Sequential Algorithm A simple sequential algorithm that colors vertices in at most $(\Delta + 1)$ colors is described in Fig. 2.7. The array variable $COLOR[1..n]$, which is initialized to $[\bot, \ldots, \bot]$, is such that, when the algorithm terminates, for any i, $COLOR[i]$ will contain the color assigned to process p_i.

The colors are represented by the integers 1 to $(\Delta + 1)$. The algorithm considers sequentially each vertex i (process p_i) and assigns to it the first color not assigned to its neighbors. (This algorithm is sensitive to the order in which the vertices and the colors are considered.)

```
(1)  for each j ∈ neighbors_i do send INIT(color_i[i]) to p_j end for;
(2)  for each j ∈ neighbors_i
(3)        do wait (INIT(col_j) received from p_j); color_i[j] ← col_j
(4)  end for;
(5)  for r_i from (Δ + 2) to m do
       begin asynchronous round
(6)    if (color_i[i] = r_i)
(7)      then c ← smallest color in {1, ..., Δ + 1} such that ∀ j ∈ neighbors_i : color_i[j] ≠ c;
(8)           color_i[i] ← c
(9)    end if;
(10)   for each j ∈ neighbors_i do send COLOR(r_i, color_i[i]) to p_j end for;
(11)   for each j ∈ neighbors_i do
(12)     wait (COLOR(r, col_j) with r = r_i received from p_j);
(13)     color_i[j] ← col_j
(14)   end for
       end asynchronous round
(15) end for.
```

Fig. 2.8 Distributed $(\Delta+1)$-coloring from an initial m-coloring where $n \geq m \geq \Delta+2$

2.2.2 *Distributed* $(\Delta+1)$*-Coloring of Processes*

This section presents a distributed algorithm which colors the processes in at most $(\Delta+1)$ colors in such a way that no two neighbors have the same color. Distributed coloring is encountered in practical problems such as resource allocation or processor scheduling. More generally, distributed coloring algorithms are symmetry breaking algorithms in the sense that they partition the set of processes into subsets (a subset per color) such that no two processes in the same subset are neighbors.

Initial Context of the Distributed Algorithm Such a distributed algorithm is described in Fig. 2.8. This algorithm assumes that the processes are already colored in $m \geq \Delta+1$ colors in such a way that no two neighbors have the same color. Let us observe that, from a computability point of view, this is a "no-cost" assumption (because taking $m = n$ and defining the color of a process p_i as its index i trivially satisfies this initial coloring assumption). Differently, taking $m = \Delta+1$ assumes that the problem is already solved. Hence, the assumption on the value of m is a complexity-related assumption.

Local Variables Each process p_i manages a local variable $color_i[i]$ which initially contains its initial color, and will contain its final color at the end of the algorithm. A process p_i also manages a local variable $color_i[j]$ for each of its neighbors p_j. As the algorithm is asynchronous and round-based, the local variable r_i managed by p_i denotes its current local round number.

Behavior of a Process p_i The processes proceed in consecutive asynchronous rounds and, at each round, each process synchronizes its progress with its neighbors. As the rounds are asynchronous, the round numbers are not given for free by

the computation model. They have to be explicitly managed by the processes themselves. Hence, each process p_i manages a local variable r_i that it increases when it starts a new asynchronous round (line 5).

The first round (lines 1–2) is an initial round during which the processes exchange their initial color in order to fill in their local array $color_i[neighbors_i]$. If the processes know the initial colors of their neighbors, this communication round can be suppressed. The processes then execute $m - (\Delta + 1)$ asynchronous rounds (line 5).

The processes whose initial color belongs to the set of colors $\{1, \ldots, \Delta + 1\}$ keep their color forever. The other processes update their colors in order to obtain a color in $\{1, \ldots, \Delta + 1\}$. To that end, all the processes execute sequentially the rounds $\Delta + 2, \ldots,$ until m, considering that each round number corresponds to a given distinct color. During round r, $\Delta + 2 \leq r \leq m$, each process whose initial color is r looks for a new color in $\{1, \ldots, \Delta + 1\}$ which is not the color of its neighbors and adopts it as its new color (lines 6–8). Then, each process exchanges its color with its neighbors (lines 10–14) before proceeding to the next round. Hence, the round invariant is the following one: When a round r terminates, the processes whose initial colors were in $\{1, \ldots, r\}$ (a) have a color in the set $\{1, \ldots, \Delta + 1\}$, and (b) have different colors if they are neighbors.

Cost The time complexity (counted in number of rounds) is $m - \Delta$ rounds (an initial round plus $m - (\Delta + 1)$ rounds). Each message carries a tag, a color, and possibly a round number which is also a color. As the initial colors are in $\{1, \ldots, m\}$, the message bit complexity is $O(\log_2 m)$.

Finally, during each round, two messages are sent on each channel. The message complexity is consequently $2e(m - \Delta)$, where e denotes the number of channels.

It is easy to see that, the better the initial process coloring (i.e., the smaller the value of m), the more efficient the algorithm.

Theorem 1 *Let $m \geq \Delta + 2$. The algorithm described in Fig. 2.8 is a legal $(\Delta + 1)$-coloring of the processes (where* legal *means that no two neighbors have the same color).*

Proof Let us first observe that the processes whose initial color belongs to $\{1, \ldots, \Delta + 1\}$ never modify their color. Let us assume that, up to round r, the processes whose initial colors were in the set $\{1, \ldots, r\}$ have new colors in the set $\{1, \ldots, \Delta + 1\}$ and any two of them which are neighbors have different colors. Thanks to the initial m-coloring, this is initially true (i.e., for the fictitious round $r = \Delta + 1$).

Let us assume that the previous assertion is true up to some round $r \geq \Delta + 1$. It follows from the algorithm that, during round $r + 1$, only the processes whose current color is $r + 1$ update it. Moreover, each of them updates it (line 7) with a color that (a) belongs to the set $\{1, \ldots, \Delta + 1\}$ and (b) is not a color of its neighbors (we have seen in Sect. 2.2.1 that such a color does exist). Consequently, at the end of round $r + 1$, the processes whose initial colors were in the set $\{1, \ldots, r + 1\}$

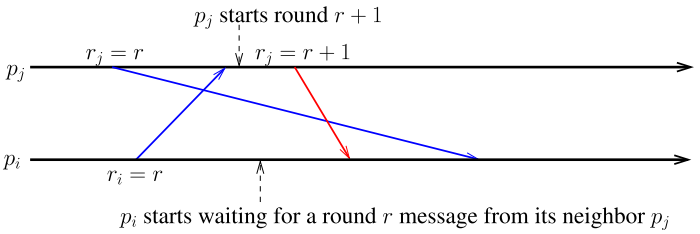

Fig. 2.9 One bit of control information when the channels are not FIFO

have new colors in the set $\{1, \ldots, \Delta + 1\}$ and no two of them have the same new color if they are neighbors. It follows that, as claimed, this property constitutes a round invariant from which we conclude that each process has a final color in the set $\{1, \ldots, \Delta + 1\}$ and no two neighbor processes have the same color. □

Remark on the Behavior of the Communication Channels Let us remember that the only assumption on channels is that they are reliable. No other behavioral assumption is made, hence the channels are implicitly non-FIFO channels.

Let us consider two neighbor processes that execute a round r as depicted in Fig. 2.9. Each of them sends its message COLOR$(r, -)$ to its neighbors (line 10), and waits for a message COLOR() from each of them, carrying the very same round number (line 12).

In the figure, p_j has received the round r message from p_i, proceeded to the next round, and sent the message COLOR$(r + 1, -)$ to p_i while p_i is still waiting for round r message from p_j. Moreover, as the channel is not FIFO, the figure depicts the case where the message COLOR$(r+1, -)$ sent by p_j to p_i arrives before the message COLOR$(r, -)$ it sent previously. As indicated in line 12, the algorithm forces p_i to wait for the message COLOR$(r, -)$ in order to terminate its round r.

As, in each round, each process sends a message to each of its neighbors, a closer analysis of the message exchange pattern shows that the following relation on round numbers is invariant. At any time we have:

$$\forall (i, j): \quad (p_i \text{ and } p_j \text{ are neighbors}) \quad \Rightarrow \quad \big(0 \leq |r_i - r_j| \leq 1\big).$$

It follows that the message COLOR() does not need to carry the value of r but only a bit, namely the parity of r. The algorithm can then be simplified as follows:

- At line 10, each process p_i sends the message COLOR$(r_i \bmod 2, color_i[i])$ to each of its neighbors.
- At line 12, each process p_i waits for a message COLOR$(b, color_i[i])$ from each of its neighbors where $b = (r_i \bmod 2)$.

Finally, it follows from previous discussion that, if the channels are FIFO, the messages COLOR() do not need to carry a control value (neither r, nor its parity bit).

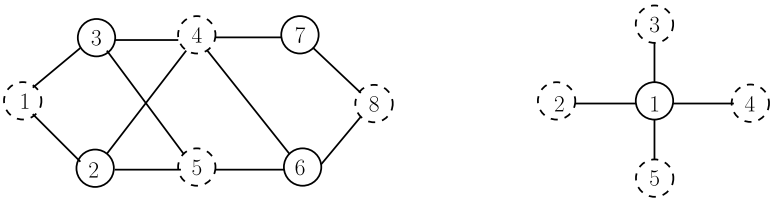

Fig. 2.10 Examples of maximal independent sets

2.2.3 Computing a Maximal Independent Set

Maximal Independent Set: Definition An *independent set* is a subset of the vertices of a graph such that no two of them are neighbors. An independent set M is *maximal* if none of its strict supersets M' (i.e., $M \subset M'$ and $M \neq M'$) is an independent set. A graph can have several maximal independent sets.

The subset of vertices $\{1, 4, 5, 8\}$ of the graph of depicted in the left part of Fig. 2.10 is a maximal independent set. The subsets $\{1, 5, 7\}$ and $\{2, 3, 6, 7\}$ are other examples of maximal independent sets of the same graph. The graph depicted on the right part has two maximal independent sets, the set $\{1\}$ and the set $\{2, 3, 4, 5\}$.

There is a trivial greedy algorithm to compute a maximal independent set in a sequential context. Select a vertex, add it to the independent set, suppress it and its neighbors from the graph, and iterate until there are no more vertices. It follows that the problem of computing a maximal independent set belongs to the time complexity class P (the class of problems that can be solved by an algorithm whose time complexity is polynomial).

A *maximum independent set* is an independent set with maximal cardinality. When considering the graph at the left of Fig. 2.10, the maximal independent sets $\{1, 4, 5, 8\}$ and $\{2, 3, 6, 7\}$ are maximum independent sets. The graph on the right of the figure has a single maximum independent set, namely the set $\{2, 3, 4, 5\}$.

While, from a time complexity point of view, the computation of a maximal independent set is an *easy* problem, the computation of a maximum independent set is a *hard* problem: it belongs to the class of NP-complete problems.

From m-Coloring to a Maximal Independent Set An asynchronous distributed algorithm that computes a maximal independent set is presented in Fig. 2.11. Each process p_i manages a local array $selected_i[j]$, $j \in neighbors_i \cup \{i\}$, initialized to $[false, \ldots, false]$. At the end of the algorithm p_i belongs to the maximal independent set if and only if $selected_i[i]$ is equal to $true$.

This algorithm assumes that there is an initial m-coloring of the processes (as we have just seen, this can be obtained from the algorithm of Fig. 2.8). Hence, the algorithm of Fig. 2.11 is a distributed reduction of the maximal independent set problem to the m-coloring problem. Its underlying principle is based on a simple observation and a greedy strategy. More precisely,

- Simple observation: the processes that have the same color define an independent set, but this set is not necessarily maximal.

```
(1)  for r_i from 1 to m do
       begin asynchronous round
(2)    if (color_i = r_i) then
(3)      if (⋀_{j∈neighbors_i} (¬selected_i[j])) then selected_i[i] ← true end if;
(4)    end if;
(5)    for each j ∈ neighbors_i do send SELECTED(r_i, selected_i[i]) to p_j end for;
(6)    for each j ∈ neighbors_i do
(7)          wait (SELECTED(r, selected_j) with r = r_i received from p_j);
(8)          selected_i[j] ← selected_j
(9)    end for
       end asynchronous round
(10) end for.
```

Fig. 2.11 From m-coloring to a maximal independent set (code for p_i)

- Greedy strategy: as the previous set is not necessarily maximal, the algorithm starts with an initial independent set (defined by some color) and executes a sequence of rounds, each round r corresponding to a color, in which it strives to add to the independent set under construction as much possible processes whose color is r. The corresponding "addition" predicate for a process p_i with color r is that none of its neighbors is already in the set.

As previous algorithms, the algorithm described in Fig. 2.11 simulates a synchronous algorithm. The color of a process p_i is kept in its local variable denoted $color_i$. The messages carry a round number (color) which can be replaced by its parity. The processes execute m asynchronous rounds (a round per color). When it executes round r, if its color is r and none of its neighbors belongs to the set under construction, a process p_i adds itself to the set (line 3). Then, before starting the next round, the processes exchange their membership of the maximal independent set in order to update their local variables $selected_i[j]$. (As we can see, what is important is not the fact that the rounds are executed in the order $1, \ldots, m$, but the fact that the processes execute the rounds in the same predefined order, e.g., $1, m, 2, (m-1), \ldots$.)

The size of the maximal independent set that is computed is very sensitive to the order in which the colors are visited by the algorithm. As an example, let us consider the graph at the right of Fig. 2.10 where the process p_1 is colored a while the other processes are colored b. If $a = 1$ and $b = 2$, the maximal independent set that is built is the set $\{1\}$. If $a = 2$ and $b = 1$, the maximal independent set that is built is the set $\{2, 3, 4, 5\}$.

A Simple Algorithm for Maximal Independent Set This section presents an algorithm, due to M. Luby (1987), that builds a maximal independent set.

This algorithm uses a random function denoted random() which outputs a random value each time it is called (the benefit of using random values is motivated below). For ease of exposition, this algorithm, which is described in Fig. 2.12, is expressed in the synchronous model. Let us remember that the main property of the synchronous model lies in the fact that a message sent in a round is received by its

```
(1)  repeat forever
       begin three synchronous rounds r, r + 1 and r + 2
       beginning of round r
(2)    random_i[i] ← random();
(3)    for each j ∈ com_with_i do send RANDOM(random_i[i]) to p_j end for;
(4)    for each j ∈ com_with_i do
(5)      wait (RANDOM(random_j) received from p_j); random_i[j] ← random_j
(6)    end for;
       end of round r and beginning of round r + 1
(7)    if (∀ j ∈ com_with_i : random_i[j] > random_i[i])
(8)      then for each j ∈ com_with_i do send SELECTED(yes) to p_j end for;
(9)           state_i ← in; return(in)
(10)     else for each j ∈ com_with_i do send SELECTED(no) to p_j end for;
(11)          for each j ∈ com_with_i do wait (SELECTED(−) received from p_j) end for;
              end of round r + 1 and beginning of round r + 2
(12)          if (∃ k ∈ com_with_i : SELECTED(yes) received from p_k)
(13)            then for each j ∈ com_with_i : SELECTED(no) received from p_j
(14)                   do send ELIMINATED(yes) to p_j
(15)                 end for;
(16)                 state_i ← out; return(out)
(17)            else for each j ∈ com_with_i do send ELIMINATED(no) to p_j end for;
(18)                 for each j ∈ com_with_i
(19)                   do wait (ELIMINATED(−) received from p_j)
(20)                 end for;
(21)                 for each j ∈ com_with_i : ELIMINATED(yes) received from p_j
(22)                   do com_with_i ← com_with_i \ {j}
(23)                 end for;
(24)                 if (com_with_i = ∅) then state_i ← in; return(in) end if
(25)          end if
(26)   end if:
       end three synchronous rounds
(27) end repeat.
```

Fig. 2.12 Luby's synchronous random algorithm for a maximal independent set (code for p_i)

destination process in the very same round. (It is easy to extend this algorithm so that it works in the asynchronous model.)

Each process p_i manages the following local variables.

- The local variable $state_i$, whose initial value is arbitrary, is updated only once. It final value (*in* or *out*) indicates whether p_i belongs or not to the maximal independent set that is computed. When, it has updated $state_i$ to its final value, a process p_i executes the statement return() which stops its participation to the algorithm. Let us notice that the processes do not necessarily terminate during the same round.
- The local variable com_with_i, which is initialized to $neighbors_i$, is a set containing the processes with which p_i will continue to communicate during the next round.
- Each local variable $random_i[j]$, where $j \in neighbors_i \cup \{i\}$, represents the local knowledge of p_i about the last random number used by p_j.

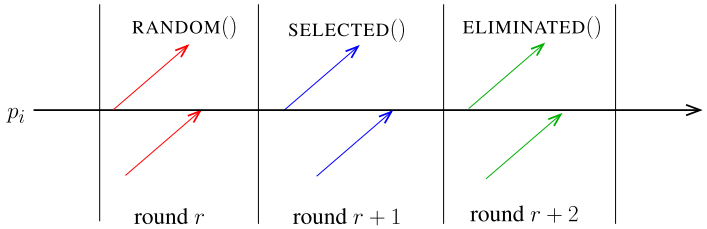

Fig. 2.13 Messages exchanged during three consecutive rounds

As indicated, the processes execute a sequence of synchronous rounds. The code of the algorithm consists in the description of three consecutive rounds, namely the rounds r, $r+1$, and $r+2$, where $r = 1, 4, 7, 10, \ldots$. The messages exchanged during these three consecutive rounds are depicted in Fig. 2.13.

The behavior of the synchronous algorithm during these three consecutive rounds is as follows:

- Round r: lines 2–6.
  Each process p_i invokes first the function random() to obtain a random number (line 2) that it sends to all its neighbors it is still communicating with (line 3). Then, it stores all the random numbers it has received, each coming from a process in com_with_i.
- Round $r+1$: lines 7–11.
  Then, p_i sends the message SELECTED(yes) to its neighbors in com_with_i if its random number is smaller than theirs (line 8). In this case, it progresses to the local state in and stops (line 9).
  Otherwise, its random number is not the smallest. In this case, p_i first sends the message SELECTED(no) to its neighbors in com_with_i (line 10), and then waits for a message from each of these neighbors (line 11).
- Round $r+2$: lines 12–26.
  Finally, if p_i has not entered the maximal independent set under construction, it checks if one of its neighbors in com_with_i has been added to this set (line 12).
  - If one of its neighbors has been added to the independent set, p_i cannot be added to this set in the future. It consequently sends the message ELIMINATED(yes) to its neighbors in com_with_i to inform them that it no longer competes to enter the independent set (line 13). In that case, it also enters the local state out and returns it (line 16).
  - If none of its neighbors in com_with_i has been added to the independent set, p_i sends them the message ELIMINATED(no) to inform them that it is still competing to enter the independent set (line 17). Then, it waits for a message ELIMINATED($-$) from each of them (line 18) and suppresses from the set com_with_i its neighbors that are no longer competing (those are the processes which sent it the message ELIMINATED(yes), lines 21–23).
    Finally, p_i checks if $com\_with_i = \emptyset$. If it is the case, it enters the independent set and returns (line 24). Otherwise, it proceeds to the next round.

The algorithm computes an independent set because when a process is added to the set, all its neighbors stop competing to be in the set (lines 12–15). This set is maximal because when a process enters the independent set, only its neighbors are eliminated from being candidates.

Why to Use Random Numbers Instead of Initial Names or Precomputed Colors As we have seen, the previous algorithm associates a new random number with each process when this process starts a new round triple. The reader can check that the algorithm works if each process uses its identity or a legal color instead of a new random number at each round. Hence, the question: Why use random numbers?

The instance of the algorithm using n distinct identities (or a legal process m-coloring) requires a number of round triples upper bounded by $\lceil n/2 \rceil$ (or $\lceil m/2 \rceil$). This is because, in each round triple, at least one process enters the maximal independent set and at least one process is eliminated. Taking random numbers does not reduce this upper bound (because always taking initial identities corresponds to particular random choices) but reduces it drastically in the average case (the expected number of round triples is then $O(\log_2 n)$).

2.3 Knot and Cycle Detection

Knots and cycles are graph patterns encountered when one has to solve distributed computing problems such as deadlock detection. This section presents an asynchronous distributed algorithm that detects such graph patterns.

2.3.1 Directed Graph, Knot, and Cycle

A directed graph is a graph where every edge is oriented from one vertex to another vertex. A directed path in a directed graph is a sequence of vertices $i_1, i_2, \ldots, i_x$ such that for any y, $1 \leq y < x$, there is an edge from the vertex i_y to the vertex i_{y+1}. A cycle is a directed path such that $i_x = i_1$.

A knot in a directed graph G is a subgraph G' such that (a) any pair of vertices in G' belongs to a cycle and (b) there is no directed path from a vertex in G' to a vertex which is not in G'. Hence, a vertex of a directed graph belongs to a knot if and only if it is reachable from all the vertices that are reachable from it. Intuitively, a knot is a "black hole": once in a knot, there is no way to go outside of it.

An example is given in Fig. 2.14. The directed graph has 11 vertices. The set of vertices $\{7, 10, 11\}$ defines a cycle which is not in a knot (this is because, when traveling on this cycle, it is possible to exit from it). The subgraph restricted to the vertices $\{3, 5, 6, 8, 9\}$ is a knot (after entering this set of vertices, it is impossible to exit from it).

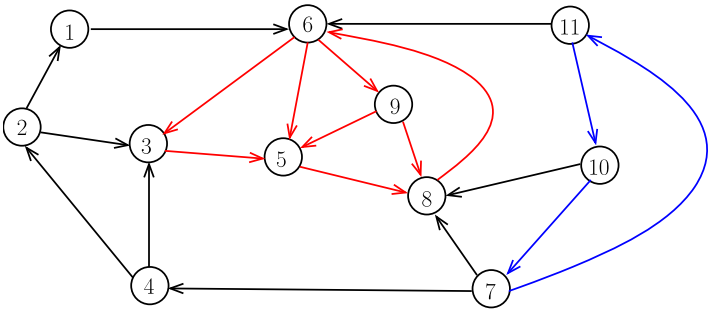

Fig. 2.14 A directed graph with a knot

2.3.2 Communication Graph, Logical Directed Graph, and Reachability

As previously, the underlying communication graph is not directed. Each channel is bidirectional which means that, if two processes are neighbors, either of them can send messages to the other.

It is assumed that a directed graph is defined on the communication graph. Its vertices are the processes, and if p_i and p_j are connected by a communication channel, there is (a) either a logical directed edge from p_i to p_j, or (b) a logical directed edge from p_j to p_i, or (c) two logical directed edges (one in each direction).

If there is a directed edge from p_i to p_j, we say "p_j is an *immediate successor* of p_i" and "p_i is an *immediate predecessor* of p_j". A vertex p_j is said to be *reachable* from a vertex p_i if there is a directed path from p_i to p_j.

From an application point of view, a directed edge corresponds to a dependence relation linking a process p_i to its neighbor p_j (e.g., p_i is waiting for "something" from p_j).

2.3.3 Specification of the Knot Detection Problem

The problem consists in detecting if a given process belongs to a knot of a directed graph. For simplicity, it is assumed that only one process initiates the knot detection. Multiple instantiations can be distinguished by associating with each of them an identification pair made up of a process identity and a sequence number.

The knot detection problem is defined by the following properties, where p_a is the process that initiates the detection:

- Liveness (termination). If p_a starts the knot detection algorithm, it eventually obtains an answer.

- Safety (consistency).
  - If p_a obtains the answer "`knot`", it belongs to a knot. Moreover, it knows the identity of all the processes involved in the knot.
  - If p_a obtains the answer "`no knot`", it does not belong to a knot. Moreover, if it belongs to at least one cycle, p_a knows the identity of all the processes that are involved in a cycle with p_a.

As we can see, the safety property of the knot detection problem states what is a correct result while its liveness property states that eventually a result has to be computed.

2.3.4 Principle of the Knot/Cycle Detection Algorithm

The algorithm that is presented below relies on the construction of a spanning tree enriched with appropriate statements. It is due to D. Manivannan and M. Singhal (2003).

Build a Directed Spanning Tree To determine if it belongs to a knot, the initiator p_a needs to check that every process that is reachable from it is on a cycle which includes it (p_a).

To that end, p_a sends a message GO_DETECT() to its immediate successors in the directed graph and these messages are propagated from immediate successors to immediate successors along directed edges to all the processes that are reachable from p_a. The first time it receives such a message from a process p_j, the receiver process p_i defines p_j as its parent in the directed spanning tree.

Remark The previous message GO_DETECT() and the messages CYCLE_BACK(), SEEN_BACK(), and PARENT_BACK() introduced below are nothing more than particular instances of the messages GO() and BACK() used in the graph traversal algorithms described in Chap. 1.

How to Determine Efficiently that p_a Is on a Cycle If the initiator p_a receives a message GO_DETECT() from a process p_j, it knows that it is on a cycle. The issue is then for p_a to know which are the processes involved in the cycles to which it belongs.

To that end, p_a sends a message CYCLE_BACK() to p_j and, more generally when a process p_i knows that it is on a cycle including p_a, it will send a message CYCLE_BACK() to each process from which it receives a message GO_DETECT() thereafter. Hence, these processes will learn that they are on a cycle including the initiator p_a.

But it is possible that, after it has received a first message GO_DETECT(), a process p_i receives more GO_DETECT() messages from other immediate predecessors (let p_k be one of them, see Fig. 2.15). If this message exchange pattern occurs, p_i

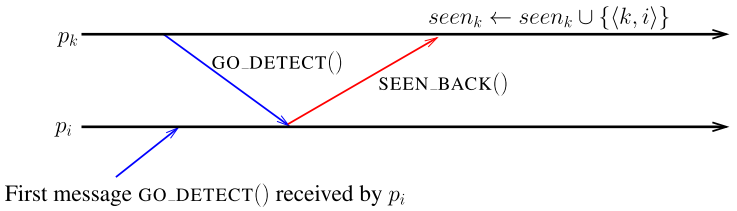

Fig. 2.15 Possible message pattern during a knot detection

sends back to p_k the message SEEN_BACK(), and when p_k receives this message it includes the ordered pair $\langle k, i\rangle$ in a local set denoted $seen_k$. (Basically, the message SEEN_BACK() informs its receiver that its sender has already received a message GO_DETECT().) In that way, if later p_i is found to be on a cycle including p_a, it can be concluded from the pair $\langle k, i\rangle \in seen_k$ that p_k is also on a cycle including p_a (this is because, due to the messages GO_DETECT(), there is a directed path from p_a to p_k and p_i, and due to the cycle involving p_a and p_i, there is a directed path from p_i to p_a).

Finally, as in graph traversal algorithms, when it has received an acknowledgment from each of its immediate successors, a process p_i sends a message PARENT_BACK() to its parent in the spanning tree. Such a message contains (a) the processes that, due to the messages CYCLE_BACK() received by p_i from immediate successors, are known by p_i to be on a cycle including p_a, and (b) the ordered pairs $\langle i, \ell\rangle$ stored in $seen_i$ as a result of the acknowledgment messages SEEN_BACK() and PARENT_BACK() it has received from its immediate successors in the logical directed graph. This information, which will be propagated in the tree to p_a, will allow p_a to determine if it is in a knot or a cycle.

2.3.5 Local Variables

Local Variable at the Initiator p_a Only The local variable $candidates_a$, which appears only at the initiator, is a set (initially empty) of process identities. If p_a is in a knot, $candidates_a$ will contain the identities of all the processes that are in the knot including p_a, when the algorithm terminates. If p_a is not in a knot, $candidates_a$ will contain all the processes that are in a cycle including p_a (if any). If $candidates_a = \emptyset$ when the algorithm terminates, p_a belongs to neither a knot, nor a cycle.

Local Variables at Each Process p_i Each (initiator or not) process p_i manages the following four local variables.

- The local variable $parent_i$ is initialized to $\perp$. If p_i is the initiator we will have $parent_i = i$ when it starts the detection algorithm. If p_i is not the initiator, $parent_i$ will contain the identity of the process from which the first message GO_DETECT() was received by p_i. When all the processes reachable from p_a

have received a message GO_DETECT(), these local variables define a directed spanning tree rooted at p_a which will be used to transmit information back to this process.

- The local variable $waiting\_from_i$ is a set of process identities. It is initialized to set of the immediate successors of p_i in the logical directed graph.
- The local variable in_cycle_i is a set (initially empty) of process identities. It will contain processes that are on a cycle including p_i.
- The local variable $seen_i$ is a set (initially empty) of ordered pairs of process identities. As we have seen, $\langle k, j \rangle \in seen_i$ means that there is a directed path from p_a to p_k and a directed edge from p_k to p_j in the directed graph. It also means that both p_k and p_j have received a message GO_DETECT() and, when p_j received the message GO_DETECT() from p_k, it did not know whether it belongs to a cycle including p_a (see Fig. 2.15).

2.3.6 Behavior of a Process

The knot detection algorithm is described in Fig. 2.16.

Launching the Algorithm The only process p_i that receives the external message START() discovers that it is the initiator, i.e., p_i is p_a. If it has no outgoing edges – predicate ($waiting\_from_i \neq \emptyset$) at line 1 –, p_i returns the pair (`no knot`,$\emptyset$), which indicates that p_i belongs neither to a cycle, nor to a knot (line 4). Otherwise, it sends the message GO_DETECT() to all its immediate successors in the directed graph (line 3).

Reception of a Message GO_DETECT() When a process p_i receives the message GO_DETECT() from p_j, it sends back to p_j the message CYCLE_BACK() if it is the initiator, i.e., if $p_i = p_a$ (line 7). If it is not the initiator and this message is the first it receives, it first defines p_j as its parent in the spanning tree (line 9). Then, if $waiting\_from_i \neq \emptyset$ (line 10), p_i propagates the detection to its immediate successors in the directed graph (line 11). If $waiting\_from_i = \emptyset$, p_i has no successor in the directed graph. It then returns the message PARENT_BACK($seen_i$, in_cycle_i) to its parent (both $seen_i$ and in_cycle_i are then equal to their initial value, i.e., $\emptyset$; $seen_i = \emptyset$ means that p_i has not seen another detection message, while $in\_cycle_i = \emptyset$ means that p_i is not involved in a cycle including the initiator).

If p_i is already in the detection tree, it sends back to p_j the message SEEN_BACK() or CYCLE_BACK() according to whether the local set in_cycle_i is empty or not (line 14–15). Hence, if $in\_cycle_i \neq \emptyset$, p_i is on a cycle including p_a and p_j will consequently learn that it is also on a cycle including p_a.

Reception of a Message XXX_BACK() When a process p_i receives a message XXX_BACK() (where XXX stands for SEEN, CYCLE, or PARENT), it first suppresses its sender p_j from $waiting\_from_i$.

```
when START() is received do
(1)  if (waiting_from_i ≠ ∅)
(2)    then parent_i ← i;
(3)         for each j ∈ waiting_from_i do send GO_DETECT() to p_j end for
(4)    else return(no knot, ∅)
(5)  end if.

when GO_DETECT() is received from p_j do
(6)  if (parent_i = i)
(7)    then send CYCLE_BACK() to p_j
(8)    else if (parent_i = ⊥)
(9)           then parent_i ← j;
(10)               if (waiting_from_i ≠ ∅)
(11)                 then for each k ∈ waiting_from_i do send GO_DETECT() to p_k end for
(12)                 else send PARENT_BACK(seen_i, in_cycle_i) to p_parent_i
(13)               end if
(14)          else if (in_cycle_i ≠ ∅) then send CYCLE_BACK() to p_j
(15)                                   else send SEEN_BACK() to p_j
(16)               end if
(17)        end if
(18) end if.

when SEEN_BACK() is received from p_j do
(19) waiting_from_i ← waiting_from_i \ {j}; seen_i ← seen_i ∪ {⟨i, j⟩}; check_waiting_from().

when CYCLE_BACK() is received from p_j do
(20) waiting_from_i ← waiting_from_i \ {j}; in_cycle_i ← in_cycle_i ∪ {j};
(21) check_waiting_from().

when PARENT_BACK(seen, in_cycle) is received from p_j do
(22) waiting_from_i ← waiting_from_i \ {j}; seen_i ← seen_i ∪ seen;
(23) if (in_cycle = ∅)
(24)          then seen_i ← seen_i ∪ {⟨i, j⟩}
(25)          else in_cycle_i ← in_cycle_i ∪ in_cycle
(26) end if;
(27) check_waiting_from().

internal operation check_waiting_from() is
(28) if (waiting_from_i = ∅) then
(29)   if (parent_i = i)
(30)     then for each k ∈ in_cycle_i do
(31)               in_cycle_i ← in_cycle_i \ {k}; candidates_i ← candidates_i ∪ {k};
(32)               for each x ∈ {1, ..., n} do
(33)                 if (⟨x, k⟩ ∈ seen_i)
(34)                       then in_cycle_i ← in_cycle_i ∪ {x};
(35)                            seen_i ← seen_i \ {⟨x, k⟩}
(36)                 end if
(37)               end for
(38)          end for;
(39)          if (seen_i = ∅) then res ← knot else res ← no knot end if;
(40)          return(res, candidates_i)
(41)     else if (in_cycle_i ≠ ∅) then in_cycle_i ← in_cycle_i ∪ {i} end if;
(42)          send PARENT_BACK(seen_i, in_cycle_i) to p_parent_i; return()
(43)   end if
(44) end if.
```

Fig. 2.16 Asynchronous knot detection (code of p_i)

As we have seen, a message SEEN_BACK() informs its receiver p_i that its sender p_j has already been visited by the detection algorithm (see Fig. 2.15). Hence, p_i adds the ordered pair $\langle i, j \rangle$ to $seen_i$ (line 19). Therefore, if later p_j is found to be on a cycle involving the initiator, the initiator will be able to conclude from $seen_i$ that p_i is also on a cycle involving p_a. The receiver p_i then invokes the internal operation check_waiting_from().

If the message received by p_i from p_j is CYCLE_BACK(), p_i adds j to in_cycle_i (line 20) before invoking check_waiting_from(). This is because there is a path from the initiator to p_i and a path from p_j to the initiator, hence p_i and p_j belong to a same cycle including p_a.

If the message received by p_i from p_j is PARENT_BACK($seen, in\_cycle$), p_i adds the ordered pairs contained in $seen$ sent by its child p_j to its set $seen_i$ (line 22). Moreover, if in_cycle is not empty, p_i merges it with in_cycle_i (line 25). Otherwise p_i adds the ordered pair $\langle i, j \rangle$ to $seen_i$ (line 24). In this way, the information allowing p_a to know (a) if it is in a knot or (b) if it is only in a cycle involving p_i will be propagated from p_i first to its parent, and then propagated from its parent until p_a. Finally, p_i invokes check_waiting_from().

The Internal Operation check_waiting_from() As just seen, this operation is invoked each time p_i receives a message XXX_BACK(). Its body is executed only if p_i has received a message XXX_BACK() from each of its immediate successors (line 28). There are two cases.

If p_i is not the initiator, it first adds itself to in_cycle_i if this set is not empty (line 41). This is because, if $in\_cycle_i \neq \emptyset$, p_i knows that it is on a cycle involving the initiator (lines 20 and 25). Then, p_i sends to its parent (whose identity has been saved in $parent_i$ at line 9) the information it knows on cycles involving the initiator. This information has been incrementally stored in its local variables $seen_i$ and in_cycle_i at lines 19–27. Finally, p_i invokes return(), which terminates its participation (line 42).

If p_i is the initiator p_a, it executes the statements of lines 30–39. First p_i cleans its local variables $seen_i$ and in_cycle_i (lines 30–38). For each $k \in in\_cycle_i$, p_i first moves k from in_cycle_i to $candidates_i$. This is because, if p_i is in a knot, so are all the processes which are on a cycle including p_i. Then, for each x, if the ordered pair $\langle x, k \rangle \in seen_i$, p_i suppresses it from $seen_i$ and adds p_x to in_cycle_i. This is because, after p_a has received a message XXX_BACK() from each of its immediate successors, we have for each process p_k reachable from p_a either $k \in in\_cycle_a$ or $\langle x, k \rangle \in seen_a$ for some p_x reachable from p_a. Hence, if $k \in in\_cycle_a$ and $\langle x, k \rangle \in seen_a$, then p_x is also in a cycle with p_a.

Therefore, after the execution of lines 30–38, $candidates_a$ contains the identities of all the processes reachable from p_a which are on a cycle with p_a. It follows that, if $seen_a$ becomes empty, all the processes reachable from p_a are on a cycle with p_a. The statement of line 39 is a direct consequence of this observation. If $seen_a = \emptyset$, p_a belongs to a knot made up of the processes which belong to the set $candidates_i$. If $seen_a \neq \emptyset$, $candidates_a$ contains all the processes that are involved in a cycle including p_a (hence if $candidates_a = \emptyset$, p_i is involved neither in a knot, nor in a cycle).

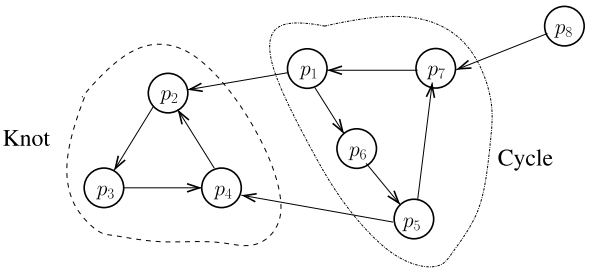

Fig. 2.17 Knot/cycle detection: example

An Example Let us consider the directed graph depicted in Fig. 2.17. This graph has a knot composed of the processes p_2, p_3, and p_4, a cycle involving the processes p_1, p_6, p_5 and p_7, plus another process p_8. If the initiator process p_a belongs to the knot, p_a will discover that it is in a knot, and we will have $candidates_a = \{2, 3, 5\}$ and $seen_a = \emptyset$ when the algorithm terminates. If the initiator process p_a belongs to the cycle on the right of the figure (e.g., p_a is p_1), we will have $candidates_a = \{1, 6, 5, 7\}$ and $seen_a = \{\langle 4, 2\rangle, \langle 3, 4\rangle, \langle 2, 3\rangle, \langle 1, 2\rangle, \langle 5, 4\rangle\}$ when the algorithm terminates (assuming that the messages GO_DETECT() propagate first along the process chain (p_1, p_2, p_3, p_4), and only then from p_5 to p_4).

Cost of the Algorithm As in a graph traversal algorithm, each edge of the directed graph is traversed at most once by a message GO_DETECT() and a message SEEN_BACK(), CYCLE_BACK() or PARENT_BACK() is sent in the opposite direction. It follows that the number of message used by the algorithm is upper bounded by $2e$, where e is the number of edges of the logical directed graph.

Let D_T be the depth of the spanning tree rooted at p_a that is built. It is easy to see that the time complexity is $2(D_T + 1)$ (D_T time units for the messages GO_DETECT() to go from the root p_a to the leaves, D_T time units for the messages XXX_BACK() to go back in the other direction and 2 more time units for the leaves to propagate the message GO_DETECT() to their immediate successors and obtain their acknowledgment messages XXX_BACK()).

2.4 Summary

Considering a distributed system as a graph whose vertices are the processes and edges are the communication channels, this chapter has presented several distributed graph algorithms. "Distributed" means here each process cooperates with its neighbors to solve a problem but never learns the whole graph structure it is part of.

The problems that have been addressed concern the computation of shortest paths, the coloring of the vertices of a graph in $\Delta + 1$ colors (where Δ is the maximal degree of the vertices), the computation of a maximal independent set, and the detection of knots and cycles.

As the reader has seen, the algorithmic techniques used to solve graph problems in a distributed context are different from their sequential counterparts.

2.5 Bibliographic Notes

- Graph notions and sequential graph algorithms are described in many textbooks, e.g., [122, 158]. Advanced results on graph theory can be found in [164]. Time complexity results of numerous graph problems are presented in [148].
- Distributed graph algorithms and associated time and message complexity analyses can be found in [219, 292].
- As indicated by its name, the sequential shortest path algorithm presented in Sect. 2.1.1 is due to R.L. Ford. It is based on Bellman's dynamic programming principle [44]. Similarly, the sequential shortest path algorithm presented in Sect. 2.1.2 is due to R.W. Floyd and R. Warshall who introduced independently similar algorithms in [128] and [384], respectively. The adaptation of Floyd–Warshall's shortest path algorithm is due to S. Toueg [373].

  Other distributed shortest path algorithms can be found in [77, 203].
- The random algorithm presented in Sect. 2.2.3, which computes a maximal independent set, is due to M. Luby [240]. The reader will find in this paper a proof that the expected number of rounds is $O(\log_2 n)$. Another complexity analysis of $(\Delta + 1)$-coloring is presented in [201].
- The knot detection algorithm described in Fig. 2.3.4 is due to D. Manivannan and M. Singhal [248] (this paper contains a correctness proof of the algorithm). Other asynchronous distributed knot detection algorithms can be found in [59, 96, 264].
- Distributed algorithms for finding centers and medians in networks can be found in [210].
- Deterministic distributed vertex coloring in polylogarithmic time, suited to synchronous systems, is addressed in [43].

2.6 Exercises and Problems

1. Adapt the algorithms described in Sect. 2.1.1 to the case where the communication channels are unidirectional in the sense that a channel transmits messages in one direction only.
2. Execute Luby's maximal independent set described in Fig. 2.12 on both graphs described in Fig. 2.10 with various values output by the function random().
3. Let us consider the instances of Luby's algorithm where, for each process, the random numbers are statically replaced by its initial identity or its color (where no two neighbor processes have the same color).

   Compare these two instances. Do they always have the same time complexity?
4. Adapt Luby's synchronous maximal independent set algorithm to an asynchronous message-passing system.
5. Considering the directed graph depicted in Fig. 2.14, execute the knot detection algorithm described in Sect. 2.3.4 (a) when p_5 launches the algorithm, (b) when p_{10} launches the algorithm, and (c) when p_4 launches the algorithm.

http://www.springer.com/978-3-642-38122-5

Distributed Algorithms for Message-Passing Systems

Raynal, M.

2013, XXXI, 500 p., Hardcover
ISBN: 978-3-642-38122-5