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 $\Delta$ 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 $\text{neighbors}_i$; 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
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 $\text{neighbors}_i$). Interestingly, it will never learn more on the structure of this graph. From a local state point of view, each process $pi$ manages the following variables.

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

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

$$\forall i, k \in \{1, \ldots, n\} : \quad \text{length}_i[k] = \min_{j \in \text{neighbors}_i} (\ell g_i[j] + \text{length}_j[k]).$$

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

**The Algorithm** The algorithm is described in Fig. 2.2. At least one process $pi$ has to receive the external message $\text{START}()$ in order to launch the algorithm. It
when START() is received do
    (1) for each $j \in \text{neighbors}_i$ do send UPDATE($\text{length}_i$) to $p_j$ end for.

when UPDATE($\text{length}$) is received from $p_j$ do
    (2) updated$_i \leftarrow$ false;
    (3) for each $k \in \{1, \ldots, n\} \setminus \{i\}$ do
        (4) if ($\text{length}_i[k] > \ell_{g_i}[j] + \text{length}[k]$)
            (5) then $\text{length}_i[k] \leftarrow \ell_{g_i}[j] + \text{length}[k]$;
                (6) $\text{routing}_\text{to}_i[k] \leftarrow j$;
                (7) updated$_i \leftarrow$ true
        end if
    end for;
    (8) if (updated$_i$)
        (9) then for each $j \in \text{neighbors}_i$ do send UPDATE($\text{length}_i$) to $p_j$ end for
        (10) end if
    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($\text{length}_i$) 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($\text{length}$) 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 $\text{length}_i[1..n]$) 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 $\text{length}_i[1..n]$ and $\text{routing}_\text{to}_i[1..n]$, $1 \leq i \leq n$, 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 $\text{length}_i[1..n]$ and $\text{routing}_\text{to}_i[1..n]$ 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 \( r = 1, 2, \ldots, D \) do

begin synchronous round

(1) for each \( j \in \text{neighbors}_i \) do send \( \text{UPDATE}(\ell g_i[j]) \) to \( p_j \) end for;

(2) for each \( j \in \text{neighbors}_i \) do receive \( \text{UPDATE}(\ell g_j) \) from \( p_j \) end for;

(3) for each \( k \in \{1, \ldots, n\} \setminus \{i\} \) do

(4) let \( \ell g_{ik_1} \) = \( \min_{j \in \text{neighbors}_i} (\ell g_i[j] + \ell g_j[k]) \);

(5) if \( \ell g_{ik_1} < \ell g_i[k] \) then

(6) \( \ell g_i[k] \leftarrow \ell g_{ik_1} \);

(7) routing_to_i[k] \leftarrow \text{a neighbor } j \text{ 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 \( \text{LENGTH}[1..n, 1..n] \) be a matrix such that, when the algorithm terminates, \( \text{LENGTH}[i, j] \) represents the length of the shortest path from \( p_i \) to \( p_j \). Initially, for any \( i \), \( \text{LENGTH}[i, i] = 0 \), for any pair \((i, j)\)
such that \( j \in \text{neighbors}_i \), \( \text{LENGTH}[i, j] \) is initialized to the length of the channel from \( p_i \) to \( p_j \), and \( \text{LENGTH}[i, j] = +\infty \) in the other cases. Moreover, for any \( i \), the array \( \text{routing}_\text{to}_i[1..n] \) is such that \( \text{routing}_\text{to}_i[i] = i \), \( \text{routing}_\text{to}_i[j] = j \) for each \( j \in \text{neighbors}_i \) and \( \text{routing}_\text{to}_i[j] \) 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 \( \text{LENGTH}[i, pv] \) 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 \( \text{LENGTH}[pv, j] \)). 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 \( pv \)th 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 \( \text{LENGTH}[1..n, 1..n] \) is split into \( n \) vectors such that each process \( p_i \) computes and maintains the value of \( \text{LENGTH}[i, 1..n] \) in its local array \( \text{length}_i[1..n] \). Moreover, as seen before, each process \( p_i \) computes the value of its routing local array \( \text{routing}_{to_i}[1..n] \). On the synchronization side, there are two issues:

- When \( p_i \) computes the value of \( \text{length}_i[j] \) during the iteration step \( p_v \), process \( p_i \) locally knows the current values of \( \text{length}_i[j] \) and \( \text{length}_i[pv] \), but it has to obtain the current value of \( \text{length}_{pv}[j] \) (see line 4 of Fig. 2.4).
- To obtain from \( p_{pv} \) a correct value for \( \text{length}_{pv}[j] \), the processes must execute simultaneously the same iteration step \( p_v \). If a process \( p_j \) is executing an iteration step with the pivot value \( p_v \) while another process \( p_k \) is simultaneously executing an iteration step with the pivot value \( p_v' \neq p_v \), the values they obtain, respectively, from \( p_{pv} \) for \( \text{length}_{pv}[j] \) and from \( p_{pv'} \text{length}_{pv'}[j] \) 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 \( p_v \) 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 \( p_v \), the process \( p_{pv} \) has to broadcast its local array \( \text{length}_{pv}[1..n] \) 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 \( p_v \)th 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 \( \text{routing}_{to_i}[pv] \). This means that, at the \( p_v \)th iteration, the set of local variables \( \text{routing}_{to_x}[pv] \) of the processes \( p_x \) such that \( \text{length}_x[pv] \neq +\infty \) 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 \( p_v \)th iteration only if it has a chance to improve its shortest paths with the help of \( p_{pv} \), i.e., if \( \text{length}_i[pv] \neq +\infty \).
for \( pv \) from 1 to \( n \) do
  for each \( k \in \text{neighbors}_i \) do
    if \((\text{routing}_i[pv] = k)\) then child \(\leftarrow \) yes else child \(\leftarrow \) no end if;
    send \(\text{CHILD}(pv, \text{child})\) to \( p_k \)
  end for;
  wait (a message \(\text{CHILD}(pv, \_\) received from each neighbor);
  if \((\text{length}[pv] \neq +\infty)\) then
    send \(\text{CHILD}(pv, \text{child})\) to \( p_k \)
  end if;
  wait (message \(\text{PV}_\text{LENGTH}(pv, \text{length}[1..n])\) from \( p_{\text{routing}_i[pv]} \))
  for each \( k \in \text{neighbors}_i \) do
    if \((\text{CHILD}(pv, \text{yes})\) received from \( p_k \)) then
      if \((pv = i)\) then send \(\text{PV}_\text{LENGTH}(pv, \text{length}[1..n])\) to \( p_k \)
      else send \(\text{PV}_\text{LENGTH}(pv, \text{length}[1..n])\) to \( p_k \)
    end if
  end for;
for \( j \) from 1 to \( n \) do
  if \(\text{length}[pv] + \text{length}[j] < \text{length}[j]\)
    then \(\text{length}[j] \leftarrow \text{length}[pv] + \text{length}[j]\);
    \(\text{routing}_i[j] \leftarrow \text{routing}_i[pv]\)
  end if
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 \(\text{length}[pv] \neq +\infty\) receives a copy of the array \(\text{length}[pv][1..n]\) 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 \(\text{length}[pv][1..n]\) by \( p_{pv} \) is launched at line 13, where this process sends the message \(\text{PV}_\text{LENGTH}(pv, \text{length}[1..n])\) 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 \(\text{length}[1..n]\) (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 \(\text{length}[1..n]\) 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 \(\text{CHILD}()\) are sent on each channel (one in each direction) and (b) at most \((n - 1)\) messages \(\text{PV}_\text{LENGTH}()\) 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 \(\text{CHILD}()\) carries a bit, while \(\text{PV}_\text{LENGTH}()\) carries \( n \) values whose size depends on the individual lengths associated with the communication channels.
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 $\Delta$ 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 $\Delta$ vertices. Then, assuming it is true for any graph made up of $n \geq \Delta$ vertices and whose maximal degree is at most $\Delta$, let us add a new vertex to the graph. As (by assumption) the maximal degree of the graph is $\Delta$, it follows that this new vertex has at most $\Delta$ 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 $\text{COLOR}[1..n]$, which is initialized to $[\perp, \ldots, \perp]$, is such that, when the algorithm terminates, for any $i$, $\text{COLOR}[i]$ 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.)
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 $\text{color}_i[i]$ 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 $\text{color}_i[j]$ 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 $\{1, \ldots, r + 1\}$
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 \( \text{COLOR}(r, -) \) to its neighbors (line 10), and waits for a message \( \text{COLOR}() \) 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 \( \text{COLOR}(r + 1, -) \) 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 \( \text{COLOR}(r + 1, -) \) sent by \( p_j \) to \( p_i \) arrives before the message \( \text{COLOR}(r, -) \) it sent previously. As indicated in line 12, the algorithm forces \( p_i \) to wait for the message \( \text{COLOR}(r, -) \) 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:

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

It follows that the message \( \text{COLOR}() \) 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 \( \text{COLOR}(r_i \mod 2, color_i[i]) \) to each of its neighbors.
- At line 12, each process \( p_i \) waits for a message \( \text{COLOR}(b, color_i[i]) \) from each of its neighbors where \( b = (r_i \mod 2) \).

Finally, it follows from previous discussion that, if the channels are FIFO, the messages \( \text{COLOR}() \) do not need to carry a control value (neither \( r \), nor its parity bit).
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 \( \text{selected}_i[j], j \in \text{neighbors}_i \cup \{i\} \), initialized to \( \text{false}, \ldots, \text{false} \). At the end of the algorithm \( p_i \) belongs to the maximal independent set if and only if \( \text{selected}_i[i] \) is equal to \( \text{true} \).

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.
2.2 Vertex Coloring and Maximal Independent Set

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 \( \text{color}_i \). 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 \( \text{selected}_i[j] \). (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 \( \text{random}() \) 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
repeat forever
begin three synchronous rounds $r$, $r+1$ and $r+2$
beginning of round $r$

random$_i$[i] $\leftarrow$ random();

for each $j \in \text{com}_i$ do send random(random$_i$[i]) to $p_j$ end for;

for each $j \in \text{com}_i$ do

wait (random(random$_j$) received from $p_j$); random$_i$[j] $\leftarrow$ random$_j$
end for;

end of round $r$ and beginning of round $r+1$

if ($\forall j \in \text{com}_i : \text{random}_i[j] > \text{random}_i[i]$)

then for each $j \in \text{com}_i$ do send SELECTED(yes) to $p_j$ end for;

state$_i$ $\leftarrow$ in; return(in)
else for each $j \in \text{com}_i$ do send SELECTED(no) to $p_j$ end for;

end of round $r+1$ and beginning of round $r+2$

if ($\exists k \in \text{com}_i : \text{SELECTED}(\text{yes})$ received from $p_k$)

then for each $j \in \text{com}_i$ : SELECTED(no) received from $p_j$

do send ELIMINATED(yes) to $p_j$

end for;

state$_i$ $\leftarrow$ out; return(out)
else for each $j \in \text{com}_i$ do send ELIMINATED(no) to $p_j$ end for;

for each $j \in \text{com}_i$

do wait (ELIMINATED(no) received from $p_j$)

end for;

for each $j \in \text{com}_i$ : ELIMINATED(yes) received from $p_j$

do com$_i$ $\leftarrow$ com$_i$ \ {j}

end for;

if (com$_i$ = $\emptyset$) then state$_i$ $\leftarrow$ in; return(in) end if

end if

end if

end if

end three synchronous rounds
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 $\text{state}_i$, 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 $\text{state}_i$ 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 $\text{com}_i$, which is initialized to $\text{neighbors}_i$, 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 $j \in \text{neighbors}_i \cup \{i\}$, represents the local knowledge of $p_i$ about the last random number used by $p_j$. 


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 $\text{com}_i$.

- **Round $r + 1$:** lines 7–11.
  
  Then, $p_i$ sends the message $\text{SELECTED}(y \in s)$ to its neighbors in $\text{com}_i$ if its random number is smaller than theirs (line 8). In this case, it progresses to the local state $\text{in}$ and stops (line 9).

  Otherwise, its random number is not the smallest. In this case, $p_i$ first sends the message $\text{SELECTED}(\text{no})$ to its neighbors in $\text{com}_i$ (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 $\text{com}_i$ 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 $\text{ELIMINATED}(y \in s)$ to its neighbors in $\text{com}_i$ to inform them that it no longer competes to enter the independent set (line 13). In that case, it also enters the local state $\text{out}$ and returns (line 16).

  - If none of its neighbors in $\text{com}_i$ has been added to the independent set, $p_i$ sends them the message $\text{ELIMINATED}(\text{no})$ to inform them that it is still competing to enter the independent set (line 17). Then, it waits for a message $\text{ELIMINATED}(\text{no})$ from each of them (line 18) and suppresses from the set $\text{com}_i$ its neighbors that are no longer competing (those are the processes which sent it the message $\text{ELIMINATED}(y \in s)$, lines 21–23).

  Finally, $p_i$ checks if $\text{com}_i = \emptyset$. 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).
2.3 Knot and Cycle Detection

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 $⟨k, i⟩$ 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 $⟨k, i⟩ ∈ seen_k$ 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 $⟨i, ℓ⟩$ 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 $candidates_a = ∅$ 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 $⊥$. 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 \( \text{waiting}_i \) 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 \( \text{in}_i \) is a set (initially empty) of process identities. It will contain processes that are on a cycle including \( p_i \).
- The local variable \( \text{seen}_i \) is a set (initially empty) of ordered pairs of process identities. As we have seen, \( \langle k, j \rangle \in \text{seen}_i \) 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., \( p_i = p_a \). If it has no outgoing edges – predicate \( \text{waiting}_i \neq \emptyset \) at line 1 –, \( p_i \) returns the pair \( (\text{noknot}, \emptyset) \), 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**  
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., \( p_i = p_a \) (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 \( \text{waiting}_i \neq \emptyset \) (line 10), \( p_i \) propagates the detection to its immediate successors in the directed graph (line 11). If \( \text{waiting}_i = \emptyset \), \( p_i \) has no successor in the directed graph. It then returns the message PARENT_BACK\( (\text{seen}_i, \text{in}_i) \) to its parent (both \( \text{seen}_i \) and \( \text{in}_i \) are then equal to their initial value, i.e., \( \emptyset; \text{seen}_i = \emptyset \) means that \( p_i \) has not seen another detection message, while \( \text{in}_i = \emptyset \) 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 \( \text{in}_i \) is empty or not (line 14–15). Hence, if \( \text{in}_i \neq \emptyset \), \( p_i \) 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**  
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 \( \text{waiting}_i \).
when \texttt{START()} is received do \\
(1) \textbf{if} (waiting\_from\_i \neq \emptyset) \Rightarrow \textbf{then} parent\_i \leftarrow i; \\
(2) \textbf{for each} \ j \in \text{waiting\_from\_i} \textbf{do} \text{GO\_DETECT()} \rightarrow \text{p}_j \textbf{end for} \\
(3) \textbf{else} \text{return(no knot, } \emptyset) \\
(4) \textbf{end if}. \\

when \texttt{GO\_DETECT()} is received from \text{p}_j \textbf{do} \\
(5) \textbf{if} (\text{parent}_i = i) \Rightarrow \textbf{then send CYCLE\_BACK()} \rightarrow \text{p}_j \\
(6) \textbf{else if} (\text{parent}_i = \perp) \Rightarrow \textbf{then} parent\_i \leftarrow j; \\
(7) \textbf{if} (\text{waiting\_from\_i} \neq \emptyset) \Rightarrow \textbf{then for each} \ k \in \text{waiting\_from\_i} \textbf{do} \text{GO\_DETECT()} \rightarrow \text{p}_k \textbf{end for} \\
(8) \textbf{else} \text{send PARENT\_BACK(seen\_i, in\_cycle\_i)} \rightarrow \text{p}_{\text{parent\_i}} \\
(9) \textbf{end if}. \\

when \texttt{SEEN\_BACK()} is received from \text{p}_j \textbf{do} \\
(10) \text{waiting\_from\_i} \leftarrow \text{waiting\_from\_i} \setminus \{j\}; \text{seen\_i} \leftarrow \text{seen\_i} \cup \{(i, j)\}; \text{check\_waiting\_from()} \\

when \texttt{CYCLE\_BACK()} is received from \text{p}_j \textbf{do} \\
(11) \text{waiting\_from\_i} \leftarrow \text{waiting\_from\_i} \setminus \{j\}; \text{in\_cycle\_i} \leftarrow \text{in\_cycle\_i} \cup \{j\}; \\
(12) \text{check\_waiting\_from()} \\

when \texttt{PARENT\_BACK(seen, in\_cycle)} is received from \text{p}_j \textbf{do} \\
(13) \text{waiting\_from\_i} \leftarrow \text{waiting\_from\_i} \setminus \{j\}; \text{seen\_i} \leftarrow \text{seen\_i} \cup \{\text{seen}\}; \\
(14) \textbf{if} (\text{in\_cycle} = \emptyset) \Rightarrow \textbf{then seen\_i} \leftarrow \text{seen\_i} \cup \{(i, j)\} \\
(15) \textbf{else} \text{in\_cycle\_i} \leftarrow \text{in\_cycle\_i} \cup \text{in\_cycle} \\
(16) \textbf{end if}. \\
(17) \text{check\_waiting\_from()} \\

\textbf{internal operation} \text{check\_waiting\_from()} \textbf{is} \\
(18) \textbf{if} (\text{waiting\_from\_i} = \emptyset) \Rightarrow \textbf{then} \\
(19) \textbf{if} (\text{parent\_i} = i) \Rightarrow \textbf{then for each} \ k \in \text{in\_cycle\_i} \textbf{do} \\
(20) \text{in\_cycle\_i} \leftarrow \text{in\_cycle\_i} \setminus \{k\}; \text{candidates\_i} \leftarrow \text{candidates\_i} \cup \{k\}; \\
(21) \textbf{for each} \ x \in \{1, \ldots, n\} \textbf{do} \\
(22) \textbf{if} ((x, k) \in \text{seen\_i}) \Rightarrow \textbf{then in\_cycle\_i} \leftarrow \text{in\_cycle\_i} \cup \{x\}; \\
(23) \text{seen\_i} \leftarrow \text{seen\_i} \setminus \{(x, k)\} \\
(24) \textbf{end if}. \\
(25) \textbf{end for} \\
(26) \text{end if}. \\
(27) \text{end for}; \\
(28) \textbf{if} (\text{seen\_i} = \emptyset) \Rightarrow \text{res} \leftarrow \text{knot} \text{ else } \text{res} \leftarrow \text{no knot} \textbf{end if}; \\
(29) \text{return(res, candidates\_i) } \\
(30) \textbf{else if} (\text{in\_cycle\_i} \neq \emptyset) \Rightarrow \textbf{then in\_cycle\_i} \leftarrow \text{in\_cycle\_i} \cup \{i\} \textbf{end if}; \\
(31) \text{send PARENT\_BACK(seen\_i, in\_cycle\_i)} \rightarrow \text{p}_{\text{parent\_i}} \textbf{; return()}; \\
(32) \textbf{end if} \\
(33) \textbf{end if} \\
(34) \textbf{end if} \\
(35) \textbf{end if} \\
(36) \textbf{end for} \\
(37) \textbf{end for}; \\
(38) \textbf{if} (\text{seen\_i} = \emptyset) \Rightarrow \text{res} \leftarrow \text{knot} \text{ else } \text{res} \leftarrow \text{no knot} \textbf{end if}; \\
(39) \text{return(res, candidates\_i) } \\
(40) \textbf{else if} (\text{in\_cycle\_i} \neq \emptyset) \Rightarrow \textbf{then in\_cycle\_i} \leftarrow \text{in\_cycle\_i} \cup \{i\} \textbf{end if}; \\
(41) \text{send PARENT\_BACK(seen\_i, in\_cycle\_i)} \rightarrow \text{p}_{\text{parent\_i}} \textbf{; return()}; \\
(42) \textbf{end if} \\
(43) \textbf{end if} \\
(44) \textbf{end if}. \\

\textbf{Fig. 2.16} \text{ Asynchronous knot detection (code of } \text{p}_i)
As we have seen, a message \textsc{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 \textit{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 \textit{seen}_i that \(p_i\) is also on a cycle involving \(p_a\). The receiver \(p_i\) then invokes the internal operation \textsc{check\_waiting\_from}().

If the message received by \(p_i\) from \(p_j\) is \textsc{cycle\_back}(), \(p_i\) adds \(j\) to \textit{in\_cycle}_i (line 20) before invoking \textsc{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 \textsc{parent\_back}(\textit{seen}, \textit{in\_cycle}), \(p_i\) adds the ordered pairs contained in \textit{seen} sent by its child \(p_j\) to its set \textit{seen}_i (line 22). Moreover, if \textit{in\_cycle} is not empty, \(p_i\) merges it with \textit{in\_cycle}_i (line 25). Otherwise \(p_i\) adds the ordered pair \(\langle i, j \rangle\) to \textit{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 \textsc{check\_waiting\_from}().

The Internal Operation \textsc{check\_waiting\_from}() As just seen, this operation is invoked each time \(p_i\) receives a message \textsc{xxx\_back}(). Its body is executed only if \(p_i\) has received a message \textsc{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 \textit{in\_cycle}_i if this set is not empty (line 41). This is because, if \textit{in\_cycle}_i ≠ \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 \textit{parent}_i at line 9) the information it knows on cycles involving the initiator. This information has been incrementally stored in its local variables \textit{seen}_i and \textit{in\_cycle}_i at lines 19–27. Finally, \(p_i\) invokes \textsc{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 \textit{seen}_i and \textit{in\_cycle}_i (lines 30–38). For each \(k \in \textit{in\_cycle}_i\), \(p_i\) first moves \(k\) from \textit{in\_cycle}_i to \textit{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 \(\langle x, k \rangle\) ∈ \textit{seen}_i, \(p_i\) suppresses it from \textit{seen}_i and adds \(x\) to \textit{in\_cycle}_i. This is because, after \(p_a\) has received a message \textsc{xxx\_back}() from each of its immediate successors, we have for each process \(p_k\) reachable from \(p_a\) either \(k \in \textit{in\_cycle}_a\) or \(\langle x, k \rangle\) ∈ \textit{seen}_a for some \(p_x\) reachable from \(p_a\). Hence, if \(k \in \textit{in\_cycle}_a\) and \(\langle x, k \rangle\) ∈ \textit{seen}_a, then \(p_x\) is also in a cycle with \(p_a\).

Therefore, after the execution of lines 30–38, \textit{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 \textit{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 \(\textit{seen}_a = \emptyset\), \(p_a\) belongs to a knot made up of the processes which belong to the set \textit{candidates}_i. If \(\textit{seen}_a \neq \emptyset\), \textit{candidates}_a contains all the processes that are involved in a cycle including \(p_a\) (hence if \textit{candidates}_a = \emptyset, \(p_i\) is involved neither in a knot, nor in a cycle).
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 $\text{candidates}_a = \{2, 3, 5\}$ and $\text{seen}_a = \emptyset$ 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 $\text{candidates}_a = \{1, 6, 5, 7\}$ and $\text{seen}_a = \{\langle 4, 2 \rangle, \langle 3, 4 \rangle, \langle 2, 3 \rangle, \langle 1, 2 \rangle, \langle 5, 4 \rangle\}$ 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 $\Delta$ 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 $\text{random}()$.

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.
Distributed Algorithms for Message-Passing Systems
Raynal, M.
2013, XXXI, 500 p., Hardcover
ISBN: 978-3-642-38122-5