entity *sequentially* (i.e., one at the time). Once a node is reached by the token, it is marked as "visited." Depending on the traversal strategy employed, we will have different traversal protocols.

### 2.3.1 Depth-First Traversal

A well known strategy is the *depth-first traversal* of a graph. According to this strategy, the graph is visited (i.e., the token is forwarded) trying to go forward as long as possible; if it is forwarded to an already visited node, it is sent back to the sender, and that link is marked as a *back-edge*; if the token can no longer be forwarded (it is at a node where all its neighbors have been visited), the algorithm will "backtrack" until it finds an unvisited node where the token can be forwarded to.

The distributed implementation of depth-first traversal is straightforward.

1. When first visited, an entity remembers who sent the token, creates a list of all its still unvisited neighbors, forwards the token to one of them (removing it from the list), and waits for its reply returning the token.
2. When the neighbor receives the token, it will return the token immediately if it had been visited already by somebody else, notifying that the link is a back-edge; otherwise, it will first forward the token to each of its unvisited neighbors sequentially, and then reply returning the token.
3. Upon the reception of the reply, the entity forwards the token to another unvisited neighbor.
4. Should there be no more unvisited neighbors, the entity can no longer forward the token; it will then send the reply, returning the token to the node from which it first received it.

**NOTE.** When the neighbor in step (2) determines that a link is a back-edge , it knows that the sender of the token is already visited; thus, it will remove it from the list of unvisited neighbors.

We will use three types of messages: "T" to forward the token in the traversal, "Backedge" to notify the detection of a back-edge, and "Return" to return the token upon local termination.

Protocol *DF_Traversal* is shown in Figure 2.6, where the operation of extracting an element from a set $B$ and assigning it to variable $a$ is denoted by $a \Leftarrow B$. Let us examine its costs.

Focus on a link $(x,y) \in E$. What messages can be sent on it? Suppose $x$ sends T to $y$; then $y$ will only send to $x$ either Return (if it was *idle* when the T arrived) or Backedge (otherwise). In other words, on each link there will be exactly two messages transmitted. Since the traversal is sequential, $\mathbf{T}[DF\_Traversal] = \mathbf{M}[DF\_Traversal]$; hence

$$\mathbf{T}[DF\_Traversal] = \mathbf{M}[DF\_Traversal] = 2m. \tag{2.6}$$

**PROTOCOL DF_Traversal**.

- Status: $\mathcal{S}$ = {INITIATOR,IDLE,VISITED,DONE};
  $\mathcal{S}_{\text{INIT}}$ = {INITIATOR,IDLE}; $\mathcal{S}_{\text{TERM}}$ = {DONE}.
- Restrictions: **R** ;UI.

```
INITIATOR
          Spontaneously
          begin
              Unvisited:= N(x);
              initiator:= true;
              VISIT;
          end

IDLE
          Receiving (T)
          begin
              entry: = sender;
              Unvisited: = N(x) − {sender};
              initiator: = false;
              VISIT;
          end

VISITED
          Receiving (T)
          begin
              Unvisited: = Unvisited −{sender};
              send (Backedge) to {sender};
          end

          Receiving (Return)
          begin
              VISIT;
          end

          Receiving (Backedge)
          begin
              VISIT;
          end

Procedure VISIT
begin
     if Unvisited ≠ ∅ then
       next ⇐ Unvisited;
       send (T) to next;
       become VISITED
     else
       if not (initiator) then send (Return) to entry; endif
       become DONE;
     endif
end
```

**FIGURE 2.6:** DF_Traversal

To determine how efficient is the protocol, we are going to determine what is the complexity of the problem.

Using exactly the same technique we employed in the proof of Theorem 2.1.1, we have (Exercise 2.9.11):

**Theorem 2.3.1** $\mathcal{M}(\textbf{DFT/R}) \geq m$.

Therefore, the $2m$ message cost of protocol *DF_Traversal* is indeed excellent, and the protocol is *message optimal*.

**Property 2.3.1** *The message complexity of depth-first traversal under* $\mathbf{R}$ *is* $\Theta(m)$.

The time requirements of a depth-first traversal are quite different from those of a broadcast. In fact, since each node must be visited sequentially, starting from the sole initiator, the time complexity is at least the number of nodes:

**Theorem 2.3.2** $\mathcal{T}(\mathbf{DFT/R}) \geq n - 1$.

The time complexity of protocol *DF_Traversal* is dreadful. In fact, the upper bound $2m$ could be several order of magnitude larger than the lower bound $n - 1$. For example, in a complete graph, $2m = n^2 - n$. Some significant improvements in the time complexity can, however, be made by going into a finer granularity. We will discuss this topic in greater details next.

### 2.3.2 Hacking ($\star$)

Let us examine protocol *Protocol DF_Traversal* to see if it can be improved, especially its time cost.

**IMPORTANT.** When measuring ideal time, we consider only *synchronous* executions; however, when measuring messages and establishing correctness we must consider *every* possible schedule of events, especially the nonsynchronous executions.

***Basic Hacking*** The protocol we have constructed is totally sequential: in a synchronous execution, at each time unit only one message will be sent, and every message requires one unit of time. So, to improve the time complexity, we need to (1) reduce the number of messages and/or (2) introduce some concurrency.

By definition of traversal, each entity must receive the token (message T) at least once. In the execution of our protocol, however, some entities receive it more than once; those links from which these other T messages arrive are precisely the back-edges.

**Question.** Can we avoid sending T messages on back-edges?

To answer this question we must understand why T messages are sent on back-edges. When an entity $x$ sends a T message to $y$, it does not know whether the link is a back-edge or not; that is, whether $y$ has already been visited by somebody else or not. If $x$ knew which of its neighbors are already visited, it would not send a T message to them, there would be no need for Backedge messages from them, and we would be saving messages and time. Let us examine how to achieve such a condition.