Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise C-6.17

Essentially, we are performing a DFS-like traversal of the graph. The idea is to keep visiting nodes using edges that have never been traversed (in this direction) in such a way so that we are able to finish the traversal exactly at the node where we started from. To accomplish this, we keep track of the edge that we used to reach a node for the very first time. We use this edge to leave the node when we have traversed all its other edges.

The algorithm can be described as follows:

> Start from any vertex, say s, of the graph. Traverse any incident edge of s and visit a new node. On a node that you have just reached, if it has not ever been visited before, label it as VISITED. Mark the "entrance" edge of any node that is labeled as VISITED (in fact, remember the "entrance" neighbor). While being on a node u, traverse any incident edge other than the "entrance" one, that you have not ever traversed from node u, to visit a neighboring node. If, while being at a node u, you can not traverse any edge (except the "entrance" edge) that you have not traversed before, return to the "entry" neighbor; if $u = s$, there is no "entry" neighbor, so terminate.

First, we show that the algorithm is correct. Each edge that is not an "entrance" edge is traversed twice: the first time is when a node is named as VISITED and the second when we leave the node and never return back. Each other node is clearly traversed twice. Note that we finish the traversal at the node that we started it.

Any node keeps a label (boolean variable) that denotes if it has ever been visited or not. Also, a node keeps a reference to its "entrance" node. Finally, a node u needs to keep track of the incident edges that it has traversed (from this node to a neighboring node). This can be done by storing the next edge that need be traversed, assuming that an ordering has been defined over the neighboring nodes.

We give the pseudo-code. Each node stores three extra variables: a boolean flag $\mathsf{VISITED}(v)$, a node reference $\mathsf{ENTRANCE}(v)$ and an iterator $\mathsf{EDGE\_IT}(v)$

over incident edges. For all these extra variables we assume that there is a mechanism for setting and getting the corresponding values (in constant time). Method NextNeighbor(G,v) returns that neighbor u of v that will be next visited (that is, the corresponding edge has not been traversed from v to u and u is not the "entrance" node of v) or NULL if no such node can be visited.

```
Algorithm Traverse(G):
  v ← G.aVertex()
  VISITED(v) ← true
  ENTRANCE(v) ← NULL
  u ← NextNeighbor(G,v)
  DONEv ← false
  while ¬ DONE do
    if ¬ VISITED(u) then
      VISITED(u) ← true
      ENTRANCE(u) ← e
    w ← NextNeighbor(G,u)
    if w = NULL then
      if u = v then DONE ← true
      else
          Report(u,ENTRANCE(u))
          u ← ENTRANCE(u)
    else
      Report(u,w)
      u ← w

Algorithm NextNeighbor(G,v):
  if EDGE_IT(v).hasNext() then
    o ←EDGE_IT(v).nextObject()
    u ← G.opposite(v,o)
    if u = ENTRANCE(v) then
      return NextNeighbor(v)
    else return u
  else return NULL
```

The running time of the algorithm is $O(n+m)$. Each edge is clearly "traversed" (reported) twice and at each node, the decision about what is the next node to be visited, is made in constant time.