Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise C-6.19

First, even if it is not asked, we show that a connected directed graph has an Euler tour, if and only if, each vertex has the same in-degree and out-degree (degree property from now on).

If the graph has an Euler tour, then by following that tour, it is always able to leave any vertex that we visit. Thus, the in-degree of each vertex equals its out-degree.

Assume now that each vertex has the same in-degree and out-degree. we can construct an Euler tour using the following precedure. Start at any node and keep traversing edges until you reach the same vertex (the graph is connected and by the degree property we are always able to return at this node). If this cycle (not necessary simple) is a tour (contains all edges) we are done. Otherwise, delete the traversed edges from the graph and starting by any vertex of the cycle that has non-zero degree, find (by traversing edges) another cycle (observe that the degree property holds even after the edges' removal). Remove the traversed edges and continue until no edges are left in the graph. All the cycles that where discovered can be conbined to give us an Euler tour: just keep track of the starting vertex of each cycle and insert this cycle into the previous cycle.

The last procedure corresponds to a well defined linear time algorithm for finding an Euler tour. However, we give another algorithm that, using a DFS traversal over the graph, constructs an Euler tour in a systematic way.

The algorithm is very similar to the one that traverses a connected undirected graph such that each edge is traversed exactly twice (once for each direction) (see problem 2 of homework 7a - Collaborative). there by marking the entrance edge we had been able to succesfully (that is, so that no edges were left untouched) leave a vertex and never visit it again.

This, however, can not be done here because an edge has only one direction. We, instead, "preprocess" the graph, by performing a DFS traversal in the graph. however, traversing edges in the opposite direction (that is, we have a DFS traversal where each edge is trvaersed in its opposite direction (backwards)). We only label the discovering edges in this DFS. Thus, when the DFS is over, each vertex will have a unique outgoing labeled edge (the one connecting this vertex with

its perent in the discovering tree of DFS). We then simulate the algorithm for homework 7a, problem 2: starting from any node, we perform a "blind" traversal, where we do not cross any label edge unless we are forced (there are no other way to leave the current vertex).

We give the pseudo-code that describe the algorithm. Instead of walking blindly, we assume that each vertex keep an iterator OUT_EDGE_IT of the outgoing incident edges and in this way edges are traversed in a systematic way (and thus, we do not have to label them). Also, each vertex stores an edge reference LAST_EDGE; it is a reference to an edge (the one labeled by the DFS traversal) that must be traversed only if no other option is available. Finally, instead of performing a DFS on graph $\overrightarrow{G}$ traversing edges backwards, we can perform a DFS at graph $\overrightarrow{G'}$, which is graph $\overrightarrow{G}$ having each edge with a reversed direction. We then reverse all edges (to switch back to the graph $\overrightarrow{G}$) but keep the labeling.

Algorithm EulerTour(G):
  for all vertices v **do**
    OUT_EDGE_IT(v) $\leftarrow$ G.outIncidentEdges(v)
    LAST_EDGE(v) $\leftarrow$ NULL
  let $\overrightarrow{G'}$ be the graph resulting by reversing
  the direction of each edge of $\overrightarrow{G}$
  perform DFS to graph $\overrightarrow{G'}$ and for each vertex v store the
  corresponding unique outgoing discovering edge as LAST_EDGE(v)
  $s \leftarrow G$.aVertex()
  NextVertex(G,s)

Algorithm NextEdge(G,v):
  if OUT_EDGE_IT(v).hasNext() **then**
    $e \leftarrow$ OUT_EDGE_IT(v).nextObject()
    if $e =$LAST_EDGE(v) **then**
      return NextEdge(G,v)
    else return e
  else return LAST_EDGE(v)

Algorithm NextVertex(G,v):
  $e \leftarrow$ NextEdge(G,v)
  if $e \neq$ NULL **then**
    $u = G$.opposite(v,e)
    Report(e)

NextVertex(G,u)

The time complexity is clearly $O(n+m)$: the edge labeling takes linear time (using DFS in graph $\overrightarrow{G'}$) and each edge is traversed exactly once and the decision about which edge to traverse next is made in constant time.