Implementation of a simple graph with an adjacency list

Variables

vertices: set of vertices
edges: set of edges
For each vertex, we keep track of the element associated with the vertex and the degree, the in-degree and the out-degree of the vertex and the sets of incoming edges I_{in}, outgoing edges I_{out} and incident undirected edges I_{un}, that is, a 7-tuple [*element*, *degree*, *in-degree*, *out-degree*, I_{in}, I_{out}, I_{un}]. For each edge, we keep track of the element associated with the edge, whether the edge is directed, the end vertices of the edge and pointers to the edge in the sets I_{in}, I_{out} and I_{un} the edge is part of, that is, a 5-tuple [*element*, *directed?*, $vertex_1$, $vertex_2$, *pointers*] where $vertex_1$ is the origin of the edge and $vertex_2$ is the destination if the edge is directed.
invariant: *vertices* is the set of vertices of the graph and *edges* is the set of edges of the graph

Initialization

vertices $\leftarrow \emptyset$
edges $\leftarrow \emptyset$

Algorithms

adjacentVertices(*vertex*):
  input: vertex the adjacent vertices of which are returned
  output: collection of vertices adjacent to *vertex*
col $\leftarrow$ empty collection
for each *edge* in $I_{\text{in}} \cup I_{\text{out}} \cup I_{\text{un}}$ of *vertex* **do**
    add other end vertex of *edge* to *col*
return *col*

incidentEdges(*vertex*):
  input: vertex whose incident edges are returned
  output: collection of edges incident on *vertex*
col $\leftarrow$ empty collection
for each *edge* in $I_{\text{in}} \cup I_{\text{out}} \cup I_{\text{un}}$ of *vertex* **do**
    add *edge* to *col*
return *col*

areAdjacent(*first*, *second*):
  input: vertices
  output: *first* and *second* are adjacent?
found $\leftarrow$ false
if degree of *first* $<$ degree of *second* **then**
    for each *edge* in $I_{\text{in}} \cup I_{\text{out}} \cup I_{\text{un}}$ of *first* **while** not *found* **do**
        found $\leftarrow$ *found* or (*second* is the other end vertex of *edge*)
else
    for each *edge* in $I_{\text{in}} \cup I_{\text{out}} \cup I_{\text{un}}$ of *second* **while** not *found* **do**
        found $\leftarrow$ *found* or (*first* is the other end vertex of *edge*)
return *found*

inIncidentEdges(*vertex*):
  input: vertex
  output: collection of incoming edges of *vertex*
col $\leftarrow$ empty collection
for each edge *edge* in I_{in} of *vertex* **do**

    add *edge* to *col*
return *col*

outIncidentEdges(*vertex*):
  input: vertex
  output: collection of outgoing edges of *vertex*
col ← empty collection
for each edge *edge* in I_{out} of *vertex* **do**
    add *edge* to *col*
return *col*

inAdjacentVertices(*vertex*):
  input: vertex
  output: collection of vertices adjacent to *vertex* along incoming edges
col ← empty collection
for each edge *edge* in I_{in} of *vertex* **do**
    add origin of *edge* to *col*
return *col*

outAdjacentVertices(*vertex*):
  input: vertex
  output: collection of vertices adjacent to *vertex* along outgoing edges
col ← empty collection
for each edge *edge* in I_{out} of *vertex* **do**
    add destination of *edge* to *col*
return *col*

insertEdge(*first*, *second*, *element*):
  input: vertices and element
  output: undirected edge with end vertices *first* and *second* and element *element*
  precondition: there is no edge between *first* and *second*, *first* and *second* are different
  postcondition: undirected edge with end vertices *first* and *second* and element *element* has been added to the graph
edge ← undirected edge with end vertices *first* and *second* and element *element*
add *edge* to *edges*
add *edge* to I_{un} of *first* and *second*
add to *edge* pointers to I_{un} of *first* and *second*
degree of *first* ← degree of *first* + 1
degree of *second* ← degree of *second* + 1
return *edge*

insertDirectedEdge(*first*, *second*, *element*):
  input: vertices and element
  output: directed edge from *first* to *second* with element *element*
  precondition: there is no undirected edge between *first* and *second*, there is no directed edge from *first* to *second*, *first* and *second* are different
  postcondition: directed edge from *first* to *second* with element *element* has been added to the graph
edge ← directed edge from *first* to *second* with element *element*
add *edge* to *edges*
add *edge* to I_{out} of *first* and I_{in} of *second*
add to *edge* pointers to I_{out} of *first* and I_{in} of *second*
degree of *first* ← degree of *first* + 1
degree of *second* ← degree of *second* + 1
outdegree of *first* ← outdegree of *first* + 1
indegree of *second* ← indegree of *second* + 1
return *edge*

insertVertex(*element*):
  input: element
  output: vertex with element *element*
  postcondition: vertex with element *element* has been added to graph
vertex ← vertex with element *element* and degree, indegree and outdegree all 0, and I_{in}, I_{out} and I_{un} all empty
add *vertex* to *vertices*
return *vertex*

removeVertex(*vertex*):
  input: vertex to be removed
  postcondition: *vertex* and edges incident on *vertex* have been removed from graph
for each edge *edge* in I_{in} of *vertex* **do**
    removeEdge(*edge*)
for each edge *edge* in I_{out} of *vertex* **do**
    removeEdge(*edge*)
for each edge *edge* in I_{un} of *vertex* **do**
    removeEdge(*edge*)
remove *vertex* from *vertices*

removeEdge(*edge*):
  input: edge to be removed
  postcondition: *edge* has been removed from graph
(*origin*, *destination*) ← end vertices of *edge*
degree of *origin* ← degree of *origin* − 1
degree of *destination* ← degree of *destination* − 1
if *edge* is directed **then**
    outdegree of *origin* ← outdegree of *origin* − 1
    indegree of *destination* ← indegree of *destination* − 1
for each edge in an incident set *I* *edge* points to **do**
    remove *edge* from *I*
remove *edge* from *edges*

makeUndirected(*edge*):
  input: edge
  postcondition: *edge* is undirected
set *edge* to be undirected
(*origin*, *destination*) ← end vertices of *edge*
left as an exercise

reverseDirection(*edge*):
  input: edge
  precondition: *edge* is directed
  postcondition: direction of *edge* has been reversed
left as an exercise

setDirectionFrom(*edge*, *vertex*):
  input: edge and vertex
  precondition: *vertex* is an end vertex of *edge* and *edge* is directed
  postcondition: *edge* has been directed away from *vertex*
left as an exercise

setDirectionTo(*edge*, *vertex*):
  input: edge and vertex
  precondition: *vertex* is an end vertex of *edge* and *edge* is directed
  postcondition: *edge* has been directed to *vertex*
left as an exercise

All other algorithms are the same as in the implementation by means of an edge list.