ESPECIFICACIONES DE GRAFOS

ESTRUCTURAS DE DATOS

GRADO EN INFORMÁTICA

UNIVERSIDAD DE CÓRDOBA

Índice de contenidos

Actualizado a 03/05/18 a las 14:15:04

ADT Vertex[G]

- **Observers**:
  - G getData() // gets the data.
  - int getLabel() // gets the vertex label.
    - post-c: the label is unique for this vertex in the graph.
- **Modifiers**:
  - setData(d:G) // set the data.

ADT Edge[G]

- **Observers**:
  - G getData() // gets edge's data.
  - bool has(u:Vertex) // Is vertex u an end of this edge.
  - Vertex other(u:Vertex) // the vertex other than u.
    - pre-c: has(u).
    - post-c: has(retVal) and other(retVal) = u
  - Vertex first() //get the start vertex in directed graphs or one of the ends in undirected graphs.
    - post-c: other(retVal) = second()
  - Vertex second() //get the last vertex in directed graphs or one of the ends in undirected graphs.
    - post-c: other(second()) = first()
- **Modifiers**:
  - setData(d:G) // set the edge's data.
    - post-c: getData() = d

ADT Graph[V,E]

- **Creators**:
  - makeDirected() //create a directed graph.
  - makeUndirected() //create an undirected graph.
- **Observers**:
  - Bool isEmpty() //Is the graph empty?
  - Bool isDirected() //Is the graph a directed one?.
  - Bool adjacent(u,v:Vertex)// Is there any edge linking u,v?
    - pre-c: u,v are graph's vertexes.
  - Bool hasCurrVertex() // true if the cursor points to a vertex.
  - Vertex currVertex() //gets current vertex.
    - pre-c: hasCurrVertex()
  - Bool hasCurrEdge() // true if the cursor points to an edge.
  - Edge currEdge() //gets current edge.
    - pre-c: hasCurrEdge()
- **Modifiers**:
  - addVertex(d:N) //create a new vertex.
    - post-c: hasCurrVertex() and currVertex().getData()=d.
  - addEdge(u,v:Vertex, d:E) //create new edge to link u,v.
    - pre-c: u,v are graph's vertexes.
    - post-c hasCurrEdge() and currEdge().has(v) and currEdge.other(v)=u and currEdge().getData()=e.
    - post-c: isDirected() implies currEdge().first()=u and currEdge().second()=v

- removeVertex() //remove current vertex and all its edges.
  - pre-c: hasCurrVertex().
- removeEdge() //remove current edge.
  - pre-c: hasCurrEdge().

- **Modifiers**: //Cursor movement.
  - findFirstVertex(d:N) //search first vertex using data.
    - post-c: hasCurrVertex() implies currVertex().getData()=d
  - findNextVertex(d:N) //search next vertex using data.
    - post-c: hasCurrVertex() implies currVertex().getData()=d
  - findFirstEdge(d:N) //search first edge from current using data.
    - prec-c: hasCurrVertex()
    - post-c: hasCurrEdge() implies currEdge().getData()=d
  - findNextEdge(d:N) //search next edge using data.
    - prec-c: hasCurrVertex()
    - post-c: hasCurrEdge() implies currEdge().getData()=d
  - goToVertex(v:Vertex) //move cursor to a vertex.
    - pre-c: v is a graph's vertex.
    - post-c: currVertex().getData()=v.getdata()
  - goToEdge(u,v:Vertex) //move cursor to an edge.
    - pre-c: u,v, are graph's vertexes.
    - post-c: hasCurrEdge() implies currVertex()=u and currEdge().first()=u and currEdge().second()=v
  - goToFirstVertex() //move cursor to the first Vertex.
    - post-c: isEmpty() implies not hasCurrVertex().
  - nextVertex() //move cursor to next vertex.
    - pre-c: hasCurrVertex()
  - goToFirstEdge()
    - pre-c: hasCurrVertex().
    - post-c: hasCurrEdge() and isDirected() implies currVertex()==currEdge().first()
  - nextEdge()
    - pre-c: hasCurrEdge().
    - post-c: hasCurrEdge() and isDirected() implies currVertex()==currEdge().first()
    - post-c: hasCurrEdge() and not isDirected() implies currEdge().has(currVertex())

ALGORITHM Warshall(A,N): //Time Analysis O(N3)

```
Begin
P ← A
For k ← 1 to N Do
  For i ← 1 to N Do
    For j ← 1 to N Do
      If P[i,j]=0 Then
        P[i,j]←P[i,k]*P[k,j]
      End-If
    End-For
  End-For
End-For
End.
```

ALGORITHM Dijkstra(IN: W[N,N], start; OUT: D[N],P[N])

```
LOCAL: S[N],i,j,x,minD
```

```
Begin:
   S[start] <- 1
   For i From 1  To N Do
      D[i] = W[start,i]
      P[i] = start
   End-For.
   For i From 1 To N-1 Do
      minD <- inf
      For j From 1 To N Do
         If S[j]=0 And D[J]<=minD Then
            minD <- D[j]
            x <- j
         End-If
      End-For
      S[x] <- 1
      For j From 1 To N Do
         If S[j]=0 And D[j] > D[x]+W[x,j] Then
            D[j]=D[x]+W[x,j]
            P[j]=x
         End-If
      End-For
   End-For
End.
```

ALGORITHM Floyd(IN: W[N,N]; OUT: D[N,N],I[N,N])

```
Begin
D ← W
I ← 0
For k From 1 To N Do
  For i From 1 To N Do
    For j From 1 To N Do
      If D[i,k]+D[k,j]<D[i,j] Then
        D[i,j] ← D[i,k]+D[k,j]
        I[i,j] ← K
      End-If
    End-For
  End-For
End-For
End.
```

ALGORITHM depthFirst(INOUT: g:Graph, f:Functor)

```
Local:
  u: Vertex
Begin
  resetNodes(G) //unset visited flags.
  g.gotoFirstVertex()
  While g.hasCurrVertex() Do
    u <- g.currVertex()
      If not u.isVisited() Then
      dfScan(g, f)
      g.gotoVertex(u)//restore cursor
      End-If
      g.nextVertex()
  End-While
End.
```

ALGORITHM dfScan(INOUT: g:Graph; f:Functor) //Versión recursiva

```
Local:
  u,v:Vertex
Begin
```

```
  u <- g.currVertex()
  u.setVisited()
  f(u) //process the node.
  g.gotoFirstEdge()
  While g.hasCurrEdge() Do
    v <- g.currEdge().other(u)
    If not v.isVisited() Then
        g.gotoVertex(v)
      dfScan(g, f)
        g.gotoEdge(u, v)
    End-If
      g.nextEdge()
  End-While
End.
```

ALGORITHM dfScan(INOUT g:Graph, f:Functor) //Versión iterativa

```
Local:
   s : Stack[Vertex]
   u,v : Vertex
Begin
  s.push(g.currVertex())
  While not s.empty() Do
    u ← s.top()
    s.pop()
    If not u.isVisited() Then
      u.setVisited()
      f(u)
      g.gotoVertex(u)
      g.gotoFirstEdge()
      While g.hasCurrEdge() Do
        v <- g.currEdge().other(u)
        If not v.isVisited() Then
          s.push(v)
        End-If
        g.nextEdge()
      End-While
    End-If
  End-While
End.
```

ALGORITHM breadFirst(INOUT: g:Graph, f:Functor)

```
Local:
  u: Vertex
Begin
  resetNodes(G) //unset visited flags
  g.gotoFirstVertex()
  While g.hasCurrVertex() Do
    u <- g.currVertex()
      If not u.isVisited() Then
      bfScan(g, f)
      g.gotoVertex(u)//restore cursor
      End-If
      g.nextVertex()
  End-While
End.
```

ALGORITHM bfScan(INOUT: g:Graph, f:Functor)

```
Local:
  q: Queue[Vertex]
  u,v: Vertex
```

```
Begin
  u <- g.currVertex()
  f(u)
  u.setVisited()
  q.insert(u)
  While not q.isEmpty() Do
    u <- q.front()
    q.remove()
    g.gotoVertex(u)
    g.gotoFirstEdge()
    While g.hasCurrEdge() Do
      v <- g.currEdge().other(u)
      If not v.isVisited() Then
        f(v)
        v.setVisited()
        q.insert(v)
      End-If
      g.nextEdge()
    End-While
  End-While
End.
```

ALGORITHM TopologicalSorting(INOUT g:Graph; f:Funct)

```
Local:
  V: Vertex
Begin
  resetNodes(g)
  g.gotoFirstVertex()
  While g.hasCurrVertex() Do
    V ← g.currVertex()
      If not v.isVisited() Then
      tsScan(g, f)
      g.gotoVertex(v) //restore cursor
    End-If
    g.nextVertex()
  End-While
End.
```

ALGORITHM tsScan(INOUT g:Graph, f:Funct)

```
Local:
  u,v: Graph
Begin
  u ← g.currVertex()
  u.setVisited()
  g.gotoFirstEdge()
  While g.hasCurrEdge()Do
    v ← g.currEdge().other(u)
    If not v.isVisited() then
      g.gotoVertex(v)
      tsScan(g, f)
      g.gotoEdge(u,v)
    End-If
    g.nextEdge()
  End-While
  f(u)
End.

Functor::operator() (IN n:Vertex)
Begin
   list.insertFront(n)
```

```
End.
```