Algorithm **findValue(k)**

item ← findItem(k)

return item.value()

**Linear Probing** **findItem( k )**

i ← h ( k ) p ← 0

while p < N do

x ← ( i + p ) mod N

item ← A [x ]

if item = ∅ then

return NO\_SUCH\_KEY

else if item.key () = k then

return item

else p ← p + 1

return NO\_SUCH\_KEY

**Quadratic Probing findItem(k)**

i ← h(k) p ← 0

while p < N do

x← (i + **p2** ) mod N

item ← A[x]

if item = ∅ then

return NO\_SUCH\_KEY

else if item.key () = k then

return item

else p ← p + 1

return NO\_SUCH\_KEY

**Double Hashing findItem(k)**

i ← h(k) p ← 0

while p < N do

x ← (i + p\*d(k)) mod N

item ← A[x]

if item = ∅ then

return NO\_SUCH\_KEY

else if item.key () = k then

return item

else p ← p + 1

return NO\_SUCH\_KEY

**\*\*\*\*\*\*\*\*\*\*\*\* AVL TREE \*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Algorithm findValue(k, v)**

if T.isExternal (v)

return NO\_SUCH\_KEY

if k = v.key() then

return v.value()

else if k < key(v) then

return findValue(k, T.leftChild(v))

else { k > v.key() }

return findValue(k, T.rightChild(v))

**Algorithm rotateLeft(T, y) || OPPO**

Input Binary Tree T and node y in T

Output a left rotation around node y

if T.isRoot(y) throw InvalidLeftRotation

p ← T.parent(y)

gp ← T.parent(p)

T.setRightChild(p, T.leftChild(y)) **|| OP**

if T.isInternal(T.leftChild(y)) then

T.setParent(T.leftChild(y), p)

T.setLeftChild(y, p) **|| OPPOSITE**

T.setParent(p, y)

if T.isRoot(p)

then T.setRoot(y)

else if T.rightChild(gp) = p

then T.setRightChild(gp, y)

else T.setLeftChild(gp, y)

T.setParent(y, gp)

\*\*\*\*\*\*\*\*\*\* **2-4 TREE** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Algorithm QuickSelect(S, lo, hi , k )**

Input Unsorted ArrayList S and k

Output the k-th smallest element in S

p ← **inPlacePartition**(S, lo, hi)

j ← p – lo + 1 if j = k then

return S.**elemAtRank**(p)

else if j > k then

return **QuickSelect**(S, lo, p-1, k)

else return **QuickSelect**(S, p+1, hi, k-j)

**Algorithm inPlacePartition(S, lo, hi)**

Input Sequence S and ranks lo and hi, 0 ≤ lo ≤ hi < S.size()

Output the pivot is now stored at its sorted rank

p ← a random integer between lo and hi swapElements(S, lo, p)

pivot ← S.elemAtRank(lo)

j ← lo + 1

k ← hi

while j < k do

while k > j ∧ S.elemAtRank( k ) > pivot do

k ← k – 1

while j < k ∧ S.elemAtRank( j ) < pivot do

j ← j + 1

if j < k then swapElements(S, j, k) swapElements(S, lo, k)

{move pivot to sorted rank} return k

\*\*\*\*\*\* **RED BLACK TREE** \*\*\*\*\*\*\*\*\*\*

**Algorithm insertItem(k, o)**

1. Search for **key k** to locate the insertion node z using **findPosition**

2. Add the new **item (k, o) at node z** and color z red

3. while **isDoubleRed(z)**

if **isBlack**(sibling(parent(z)))

gpz ← parent(parent(z))

restructure(z)

**setColor**(gpz, RED)

**setColor**(parent(gpz), BLK)

return else { sibling(parent(z)) is red }

z ← splitRecolor(z)

**Algorithm isDoubleRed(z)**

if isRoot(z) then

setColor(z, BLK)

return False

else return isRed(parent(z))

**Algorithm splitRecolor(z)**

pz ← parent(z)

setColor(pz, BLK)

setColor(sibling(pz), BLK)

gpz ← parent(pz)

setColor(gpz, RED) return gpz

**Algorithm restructure(z)**

pz ← parent(z)

if isLeft(z) then

if isLeft(pz) then

rotateRight(pz)

else rotateRight(z)

rotateLeft(z)

else { z is a right child }

if isLeft(pz) then

rotateLeft(z)

rotateRight(z)

else rotateLeft(pz)

**\*\*Delete RBTree\*\*\***

Algorithm findNode2Remove(k)

Input returns node r containing key k, node r has at least one external node

v ← findPosition(k, root() )

r ← v

if.isInternal(leftChild(v))˄isInternal(rightChild(v

then // one child must be external

r ← findPosition(k, leftChild(v))

// finds node containing predecessor of k swapElements(v, r)

// swaps items so r is node containing key being deleted

return r // r is the node to be deleted and contains key k unless k is not in tree

**Algorithm fusionRecolor(y, p, r)**

Input r and y are siblings, p is their parent setColor(y, RED)

if isRed(p) then

setColor(p, BLK)

else setColor(p, DOUBLE\_BLACK)

if isInternal(r) then

setColor(r, BLK) return p

**Algorithm removeDoubleBlack(y, r)**

Input r is the double black node and y is sibling(r)

if isDoubleBlack(r) then

if isRed(y) then

{Case 3: when y, the sibling of r, is red}

y ← adjustment(y)

p ← parent(y)

z ← redChildOf(y)

if isBlack(z) then

{Case 1: when y has no red child}

r ← fusionRecolor(y, p, r)

if isRoot(r) then

setColor(r, BLK)

else removeDoubleBlack(sibling(r), r)

// recursive call

else {Case 2: y has a red child z, so we do a transfer}

restructure(z)

setColor(parent(p), color(p))

// new parent after restructure

setColor(p, BLK)

setColor(z, BLK)

if isInternal(r) then

setColor(r, BLK)

// make sure r is not external/null

**\*\*\*\*\*\*\* BFS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**BFS Algorithm**

**Algorithm BFS(G)**

Input graph G

Output labeling of the edges and partition of the vertices of G

for all u ∈ G.vertices() do

setLabel(u, UNEXPLORED)

for all e ∈ G.edges() do

setLabel(e, UNEXPLORED)

for all v ∈ G.vertices() do

if getLabel(v) = UNEXPLORED **BFScomponent**(G, v)\

**Algorithm BFScomponent(G, s)**

Q ← new empty Queue

Q.enqueue(s)

setLabel(s, VISITED)

while !Q.isEmpty() do

v ← Q.dequeue ()

for all e ∈ G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED then

w ← G.opposite(v,e)

if getLabel(w) = UNEXPLORED then setLabel(e, DISCOVERY)

setLabel(w, VISITED)

Q.enqueue(w)

else setLabel(e, CROSS)

**class EulerTour {**

visitExternal(T, p, result) { }

visitPreOrder(T, p, result) { }

visitInOrder(T, p, result) { }

visitPostOrder(T, p, result) { }

eulerTour(T, p) {

let result = new Array(3);

**if** (T.isExternal(p)) {

this.visitExternal(T, p, result);

} **else** { this.visitPreOrder(T, p, result);

result[0]=this.eulerTour(this.\_tree.leftChild(p)); this.visitInOrder(T, p, result);

result[2] = eulerTour(this.\_tree.rightChild(p)); this.visitPostOrder(T, p, result); }

**return** result[1]; }

**class CalculateHeight extends EulerTour** { visitExternal(T, p, result) { result[1] = 0; } visitPostOrder(T, p, result) {

result[1] = 1 + Math.max(result[0], result[2]); } height(T) { return this.eulerTour(T, T.root()); } }

**Algorithm EulerTour(T, v)**

result ← new Array(3)

**if** T.isExternal (v) then

visitExternal(T, v, result)

**else** visitPreOrder(T, v, result)

result[0] ← EulerTour(T, T.leftChild(v)) visitInOrder(v, result)

result[2] ← EulerTour(T, T.rightChild(v)) visitPostOrder(T, v, result)

return result[1]

**Algorithm visitExternal(T, v, result)**

result[1] ← 0

**Algorithm visitPostOrder(T, v, result)**

result[1] ← 1 + MAX(result[0], result[2])

**Algorithm height(T)**

return EulerTour(T, T.root())

**\*\*\*\*\*\*\*\*\*\*\* GRAPH DFS\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Path Finding**

**Algorithm pathDFS(G, v, z)**

setLabel(v, VISITED)

S.push(v)

if v = z

path ← S.elements()

for all e ∈ G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED

w ← opposite(v, e)

if getLabel(w) = UNEXPLORED

setLabel(e, DISCOVERY)

S.push(e)

pathDFS(G, w, z)

S.pop() { e gets popped }

else setLabel(e, BACK)

S.pop() { v gets popped }

**Cycle Finding**

**Algorithm cycleDFS(G, v)**

setLabel(v, VISITED)

S.push(v)

if cycle != null then return

for all e ∈ G.incidentEdges(v)

if getLabel(e) = UNEXPLORED

w ← opposite(v,e)

if getLabel(w) = UNEXPLORED

setLabel(e, DISCOVERY)

S.push(e)

cycleDFS(G, w)

S.pop()

else S.push(e)

cycle ← new empty sequence

o ← w do

cycle.insertLast(o)

o ← S.pop()

while o ≠ w

setLabel(e, BACK) S.pop()

**Algorithm DFS(G)**

Input graph G

Output the edges of G are labeled as discovery edges and back edges

initResult( G )

for all u ∈ G.vertices() do

setLabel(u, UNEXPLORED)

postInitVertex(u)

for all e ∈ G.edges() do

setLabel(e, UNEXPLORED)

postInitEdge(e)

for all v ∈ G.vertices() do

if getLabel(v) = UNEXPLORED preComponentVisit(G, v)

DFScomponent(G, v)

postComponentVisit(G, v)

return result( G )

**Algorithm DFScomponent(G, v)**

setLabel(v, VISITED)

startVertexVisit(G, v)

for all e ∈ G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED

w ← opposite(v,e)

if getLabel(w) = UNEXPLORED

setLabel(e, DISCOVERY)

preDiscoveryVisit(G, v, e, w)

DFS(G, w)

postDiscoveryVisit(G, v, e, w)

else setLabel(e, BACK)

backEdgeVisit(G, v, e, w)

finishVertexVisit(G, v)

Path Finding Override hook operations

**Algorithm DFS(G, v)**

setLabel(v, VISITED)

startVertexVisit(G, v)

for all e ∈ G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED then

w ← opposite(v,e)

if getLabel(w) = UNEXPLORED then

setLabel(e, DISCOVERY)

preDiscoveryVisit(G, v, e, w)

DFS(G, w)

postDiscoveryVisit(G, v, e, w)

else setLabel(e, BACK)

backEdgeVisit(G, v, e, w) finishVertexVisit(G, v)

**Algorithm pathDFS(G, v, z, S)**

setLabel(v, VISITED)

S.push(v)

if v = z then

path ← S.elements()

for all e ∈ G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED then

w ← opposite(v, e)

if getLabel(w) = UNEXPLORED then

setLabel(e, DISCOVERY)

S.push(e)

pathDFS(G, w, z)

S.pop() { e gets popped }

else setLabel(e, BACK)

S.pop() { v gets popped }

**Overriding hook methods in a subclass FindSimplePath**

**Algorithm findPath(G, u, v)**

S ← new empty stack {**S** is subclass field}

z ← v {**z** subclass field, the target vertex} for all u ∈ G.vertices()

setLabel(u, UNEXPLORED)

for all e ∈ G.edges()

setLabel(e, UNEXPLORED)

DFScomponent(G, u) return(path)

**Algorithm startVertexVisit(G, v)**

S.push(v)

if v=z then {z subclass field,the target} path ← S.elements()

{path subclass field & is the result} **Algorithm preDiscoveryVisit(G, v , e, w)** S.push(e)

**Algorithm postDiscoveryVisit(G, v, e, w)** S.pop() {pop e off the stack}

**Algorithm finishVertexVisit(G, v)**

S.pop() {pop v off the stack}

**Template Version of DFS (v2)**

**Algorithm DFS(G)**

Input graph G

Output the edges of G are labeled as discovery edges and back edges initResult( G )

for all u ∈ G.vertices() do

setLabel(u, UNEXPLORED) postInitVertex(u)

for all e ∈ G.edges() do

setLabel(e, UNEXPLORED)

postInitEdge(e)

for all v ∈ G.vertices() do

if isNextComponent(G, v) preComponentVisit(G, v) DFS(G, v) postComponentVisit(G, v)

return result( G )

Algorithm isNextComponent(G, v)

return getLabel(v) = UNEXPLORED

**Algorithm DFS(G, v)**

setLabel(v, VISITED)

startVertexVisit(G, v)

for all e ∈ G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED

w ← opposite(v,e)

if getLabel(w) = UNEXPLORED

setLabel(e, DISCOVERY) preDiscoveryVisit(G, v, e, w)

DFS(G, w)

postDiscoveryVisit(G, v, e, w)

else setLabel(e, BACK)

backEdgeVisit(G, v, e, w) finishVertexVisit(G, v)

**Overriding hook methods in a subclass FindSimplePath (v2)**

**Algorithm findPath(G, u, v)**

S ← new empty stack {S subclass field}

start ← u {start is a subclass field & is the starting vertex}

dest ← v {dest is a subclass field & is the destination vertex}

path ← ∅ {path is a subclass field & is the path from u to v}

return DFS(G)

**Algorithm result(G)** return(path)

**Algorithm isNextComponent(G, v)**

return v=start {

start the component traversal at vertex start} **Algorithm startVertexVisit(G, v)**

S.push(v)

if v=dest then {dest is a subclass field & is the destination vertex}

path ←S.elements() { subclass field,the result} **Algorithm preDiscoveryVisit(G, v , e, w)** S.push(e)

**Algorithm postDiscoveryVisit(G, v, e, w)** S.pop() {pop e off the stack}

**Algorithm finishVertexVisit(G, v)**

S.pop() {pop v off the stack}

**Cycle Finding Override hook operations**

**Algorithm DFScomponent(G, v)** setLabel(v, VISITED)

startVertexVisit(G, v)

for all e ∈ G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED

w ← opposite(v,e)

if getLabel(w) = UNEXPLORED

setLabel(e, DISCOVERY) preDiscoveryVisit(G, v, e, w) DFScomponent(G, w) postDiscoveryVisit(G, v, e, w)

else setLabel(e, BACK)

backEdgeVisit(G, v, e, w) finishVertexVisit(G, v)

**Algorithm cycleDFS(G, v)**

setLabel(v, VISITED)

**if cycle != ∅ then return**

**S.push(v)**

for all e ∈ G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED

w ← opposite(v,e)

S.push(e)

if getLabel(w) = UNEXPLORED

setLabel(e, DISCOVERY)

cycleDFS(G, w) S.pop()

else setLabel(e, BACK)

cycle ← new empty List

o ← w do cycle.insertLast(o)

o ← S.pop() while o != w S.pop()

**Cycle**Finding**Override**Hook**Operations-V2**

**Algorithm DFS(G, v)**

setLabel(v, VISITED)

startVertexVisit(v)

for all e ∈ G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED

w ← opposite(v,e)

preEdgeVisit(G, v, e)

if getLabel(w) = UNEXPLORED

setLabel(e, DISCOVERY) preDiscoveryVisit(G, v, e, w)

DFS(G, w)

postDiscoveryVisit(G, v, e, w)

else setLabel(e, BACK)

backEdgeVisit(G, v, e, w)

finishVertexVisit(G, v)

**Algorithm cycleDFS(G, v)**

setLabel(v, VISITED)

if cycle ≠ null then return S.push(v)

for all e ∈ G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED

w ← opposite(v,e) S.push(e)

if getLabel(w) = UNEXPLORED

setLabel(e, DISCOVERY)

cycleDFS(G, w) S.pop()

else setLabel(e, BACK)

cycle ← new empty sequence

o ← w do cycle.insertLast(o)

o ← S.pop() while o ≠ w S.pop()

**Overriding template methods in subclass FindCycles**

**Algorithm startVertexVisit(G, v)**

if ! cycleFound then S.push(v)

**Algorithm finishVertexVisit(G, v)**

if ! cycleFound then S.pop()

**Algorithm preEdgeVisit(G, v, e)**

if ! cycleFound then S.push(e)

**Algorithm postDiscoveryVisit(G, v, e, w)**

if ! cycleFound then S.pop()

**Algorithm backEdgeVisit (G, v, e, w)**

if ! cycleFound then

cycle ← new empty sequence

o ← w do cycle.insertLast(o)

o ← S.pop() while o ≠ w

cycleFound ← true

{cycleFound is a subclass field, initially false}

**Template Version of DFS**

**Algorithm DFS(G)**

Input graph G

Output the edges of G are labeled as discovery edges and back edges initResult( G )

for all u ∈ G.vertices() do

setLabel(u, UNEXPLORED) postInitVertex(u)

for all e ∈ G.edges() do

setLabel(e, UNEXPLORED)

postInitEdge(e)

for all v ∈ G.vertices() do

if getLabel(v) = UNEXPLORED preComponentVisit(G, v) DFS(G, v) postComponentVisit(G, v)

return result( G )

**Algorithm DFScomponent(G, v)**

setLabel(v, VISITED)

startVertexVisit(G, v)

for all e ∈ G.incidentEdges(v) do

preEdgeVisit(G, v, e)

if getLabel(e) = UNEXPLORED

w ← opposite(v,e)

edgeVisit(G, v, e, w)

if getLabel(w) = UNEXPLORED

setLabel(e, DISCOVERY) preDiscoveryVisit(G, v, e, w)

DFS(G, w)

postDiscoveryVisit(G, v, e, w)

else setLabel(e, BACK)

backEdgeVisit(G, v, e, w) finishVertexVisit(G, v)

**The Fractional Knapsack Algorithm**

**Algorithm fractionalKnapsack(S, W)** Input: set S of items w/ benefit bi and weight wi ; max. weight W

Output: amount xi of each item i to maximize benefit with weight at most W for each item i in S do

xi ← 0

vi ← bi / wi {value}

w ← 0 {total weight}

while w < W do

remove item i with highest vi

xi ← min{wi , W - w}

w ← w + min{wi , W - w}

**Algorithm fractionalKnapsack(S, W)**

Q ← new Max Priority Queue

x ← new Array of size n

for i ← 0 to S.size() - 1 do

(bn, wt) ← S.elemAtRank(i)

x[i] ← 0

v ← bn / wt {benefit per unit} Q.insertItem(v, (bn, wt, i) )

w ← 0 {total weight}

while w < W do

(bn, wt, i) ← Q.removeMax()

x[i] ← min{wt , W - w}

w ← w + x[i]

**A Heap (or priority queue**

**Algorithm DijkstraDistances(G, s)**

PQ ← new priority queue

for all v ∈ G.vertices() do

if v = s setDistance(v, 0)

else setDistance(v, ∞) PQ.insertItem(getDistance(v), v)

while ! PQ.isEmpty() do

u ← PQ.removeMin()

for all e ∈ G.incidentEdges(u) do

{ relax edge e }

z ← G.opposite(u,e)

d ← getDistance(u) + weight(e)

if d < getDistance(z) then s

etDistance(z,d)

PQ.replaceKey(z,d) {new method}

**Dijkstra’s Algorithm with Positions**

**Algorithm DijkstraDistances(G, s)**

PQ ← new heap-based priority queue

for all v ∈ G.vertices() do

if v = s then setDistance(v, 0)

else setDistance(v, ∞)

p ← PQ.insertItem(getDistance(v), v) setLocator(v, p)

while ! PQ.isEmpty() do

u ← PQ.removeMin()

for all e ∈ G.incidentEdges(u) do

{ relax edge e }

z ← G.opposite(u,e)

d ← getDistance(u) + weight(e)

if d < getDistance(z) then

setDistance(z,d) PQ.replaceKey(getLocator(z),d)

**Template Pattern Extension**

**Algoritm DijkstraShortestPathsTree(G, s)** …

for all v ∈ G.vertices() do

…

setParent(v, ∅)

…

for all e ∈ G.incidentEdges(u) do

{ relax edge e }

z ← G.opposite(u,e)

r ← getDistance(u) + weight(e)

if r < getDistance(z) then

setDistance(z,r)

setParent(z,e) PQ.replaceKey(getLocator(z),r)

**\*\*\* Minimum Spanning Trees\*\*\***

**Algorithm PrimJarnikMST(G)**

PQ ← new priority queue

for all v ∈ G.vertices() do

setParent(v, ∅)

d ← ∞

p ← PQ.insertItem(d, v)

setLocator(v, p)

s ← G.aVertex ()

p ← getLocator (s)

PQ.replaceKey(p, 0)

while ! PQ.isEmpty() do

u ← PQ.removeMin()

setLocator(u, ∅) {u is now in MST}

for all e ∈ G.incidentEdges(u) do

z ← G.opposite(u, e)

r ← weight(e)

p ← getLocator (z) {p=(weight, vertex)}

if p ≠ ∅ {z not yet in MST} /\ r < p.key() then setParent(z, e) PQ.replaceKey(p, r)

**Algorithm KruskalMST(G)**

for each vertex v in G do

define a Cloud(v) ← {v}

Q ← new heap-based priority queue.

for all e ∈ G.edges()

Q.insert(weight(e), e)

T ← ∅

while T has fewer than n-1 edges do

e ← Q.removeMin()

(u, v) ← G.endVertices(e)

if Cloud(v) ≠ Cloud(u) then

Add edge e to T

Merge Cloud(v) and Cloud(u)

return T

**Algorithm BaruvkaMST(G)**

T ← V {just the vertices of G, no edges, n connected components}

while T has fewer than n-1 edges do {T is not yet an MST}

for each connected component C in T do Find edge e with smallest-weight edge from C to another component in T.

if e is not already in T then

Add edge e to T

return T

**After labeling each vertex with component number**

**Algorithm DFS(G)**

Input graph G

Output the edges of G are labeled as discovery edges and back edges initResult( )

for all u ∈ G.vertices()

setLabel(u, UNEXPLORED) postVertexInit(u)

for all e ∈ G.edges()

setLabel(e, UNEXPLORED)

postEdgeInit(e)

for all v ∈ G.vertices()

if getLabel(v) = UNEXPLORED preComponentVisit(v) DFS(G, v) postComponentVisit(v) result( )

**Algorithm DFS(G, v)**

setLabel(v, VISITED)

startVertexVisit(v)

for all e ∈ G.incidentEdges(v)

if getLabel(e) = UNEXPLORED

w ← opposite(v,e)

edgeVisit(v, e, w)

if getLabel(w) = UNEXPLORED

setLabel(e, DISCOVERY) preDiscoveryTraversal(v, e, w)

DFS(G, w)

postDiscoveryTraversal(v, e, w)

else setLabel(e, BACK)

backTraversal(v, e, w)

finishVertexVisit(v)

BFS Algorithm (revised)

**Algorithm BFS(G) {top level}**

Input graph G

Output labeling of the edges and partition of the vertices of G

for all u ∈ G.vertices()

setLabel(u, UNEXPLORED)

for all e ∈ G.edges()

setLabel(e, UNEXPLORED)

for all v ∈ G.vertices()

if getLabel(v) = UNEXPLORED BFS(G, v)

**Algorithm BFS(G, s)**

L ← new empty sequence

L.insertLast(s)

setLabel(s, VISITED)

while ¬L.isEmpty()

v ← L.removeAtRank(0)

for all e ∈ G.incidentEdges(v)

if getLabel(e) = UNEXPLORED

w ← opposite(v,e)

if getLabel(w) = UNEXPLORED

setLabel(e, DISCOVERY)

setLabel(w, VISITED)

L.insertLast(w)

else setLabel(e, CROSS)

Template Version of BFS

**Algorithm BFS(G) {top level}**

Input graph G

Output labeling of the edges of G as discovery edges and cross edges initResult(G)

for all u G.vertices() do

setLabel(u, UNEXPLORED) postInitVertex(u)

for all e ∈ G.edges() do

setLabel(e, UNEXPLORED)

postInitEdge(e)

for all v ∈ G.vertices() do

if isNextComponent(G, v) preComponentVisit(G, v)

BFS(G, v)

postComponentVisit(G, v)

return result(G)

**Algorithm isNextComponent(G, v)**

return getLabel(v) = UNEXPLORED

**Algorithm BFS(G, s)**

setLabel(s, VISITED)

L.insertLast(s)

startBFS(G, s)

while ¬L.isEmpty() do

v ← L.removeAtRank(0)

preVertexVisit(G, v)

for all e ∈ G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED

w ← opposite(v, e)

preEdgeVisit(G, v, e, w)

if getLabel(w) = UNEXPLORED preDiscEdgeVisit(G, v, e, w)

setLabel(e, DISCOVERY)

setLabel(w, VISITED)

L.insertLast(w)

postDiscEdgeVisit(G, v, e, w)

else setLabel(e, CROSS)

crossEdgeVisit(G, v, e, w) postVertexVisit(G, v)

finishBFS(G, s)

BFS Algorithm (revised)

**Algorithm BFS(G)**

Input graph G

Output labeling of the edges and partition of the vertices of G

for all u ∈ G.vertices()

setLabel(u, UNEXPLORED)

for all e ∈ G.edges()

setLabel(e, UNEXPLORED)

for all v ∈ G.vertices()

if getLabel(v) = UNEXPLORED

BFScomponent(G, v)

**Algorithm BFScomponent(G, s)**

L ← new empty sequence

L.insertLast(s)

setLevel(s, 0)

{use only if level # is needed}

setLabel(s, VISITED)

while ¬L.isEmpty()

v ← L.remove (L.first())

for all e ∈ G.incidentEdges(v)

if getLabel(e) = UNEXPLORED

w ← opposite(v,e)

if getLabel(w) = UNEXPLORED

setLabel(e, DISCOVERY)

setLabel(w, VISITED)

L.insertLast(w)

setLevel(w, getLevel(v)+1)

else setLabel(e, CROSS)