Name: Pheakdey Luk **Assignment 13**

ID: 986591

The following are to be written up and turned in separately from the rest of the homework.

1. Using the DFS Template Method Pattern algorithm given in the lecture notes, override the appropriate methods so this algorithm computes the connected components of a graph G. Your method should return a sequence of vertices, 1 representative from each connected component.

**Answer:**

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()

setLabel(u, UNEXPLORED)

**postInitVertex(u)**

for all e ∈ G.edges() setLabel(e, UNEXPLORED)

**postInitEdge(e)**

for all v ∈ G.vertices()

if getLabel(v) = UNEXPLORED

**preComponentVisit(G, v)**

DFS(G, v)

**postComponentVisit(G, v)**

return result( G )

==>

Algorithm initResult(G)

S<- new sequence

Algoritm preComponentVisit(G,v)

S.insertLast(v)

Algorithm result(G)

return S

2.

a. Modify the breadth-first search algorithm so it can be used as a Template Method Pattern.

Answer:

Algorithm BFS(G)

Input graph G

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

**initResult(G)**

for all u ∈ G.vertices()

setLabel(u, UNEXPLORED)

**postInitVertex(u)**

for all e ∈ G.edges()

setLabel(e, UNEXPLORED)

**postInitEdge(e)**

for all v ∈ G.vertices()

if getLabel(v) = UNEXPLORED

**preComponentVisit(G, v)**

BFS(G, v)

**postComponentVisit(G, v)**

**return result**

Algorithm BFS(G, s)

setLabel(s, VISTED)

L.insertLast(s)

**startBFS(G, s)**

while ¬L.isEmpty() do

v <- L.removeAtRank(0)

**preVertexVisit(G, v)**

for all e ∈ G.incidentEdges(v)

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)**

b. Override the appropriate methods so that given two vertices of G, your BFS finds a path in G between them with the minimum number of edges, or report that no such path exists.

**Answer:**

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

S<-new Sequence

path<-null

pathFound<-false

z<-v

for all n in G.vertices() do

setLabel(p,UNEXPLORED)

for all l in G.edges() do

setLabel(l,UNEXPLORED)

BFS(G,u)

if !pathFound = false then

return NO\_SUCH\_PATH

else

return path

Algorithm **preEdgeVisit**(G, v)

if v=s then

v.setPath(v) {path is a property of node}

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

if !pathFound then

w.setPath(v.getPath()+e+w)

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

if z=w then

pathFound=true

path<-w.getPath()

c. Override the appropriate methods so your solution finds a simple cycle in a graph G. You as the algorithm designer are to decide what to return from your method.

Answer:

Algorithm **preEdgeVisit**(s)

if v=s then

setParent(s,null,null)

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

if !cyleFound=false then

setParent(w,v,e) {set parent and related edge}

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

cyleFound=true

S<-new Stack()

Q<-new Queue()

while getParent(v)!=getParent(w) then

S.push(v)

S.push(getParentConnectedEdge(v))

Q.enqueue(w)

Q.enqueue(getParentConnectedEdge(w))

v<- getParent(v)

w<-getParent(w)

cyclePath<-new Sequence()

cyclePath.insertLast(getParent(v))

while !S.isEmpty() then

cyclePath.insertLast(s.pop())

cyclePath.insertLast(e)

while !Q.isEmpty() then

cyclePath.insertLast(Q.dequeue())

d. Can the template version of DFS be used to find the path between two vertices with the minimum number of edges? Briefly explain why or why not.

**Answer:**

DFS is used for traversing a graph. It’s used to find the path node. To find path from node A to C, it will traverse from node A to B and then from B to C. Therefore, we cannot guarantee that template version or non-template version will find minimum edge path between two vertices.

3. Modify Dijkstra’s shortest path algorithm so it can be used as a Template Method Pattern. Then override the appropriate methods so it returns the shortest path between two vertices.

**Answer:**

Algorithm DijkstraDistances(G, s)

Q ← new heap-based priority queue

for all v ∈ G.vertices()

**preInitVertex(u)**

if v = s

setDistance(v, 0)

else

setDistance(v, ∞)

Q.insertItem(getDistance(v), v)

while ¬Q.isEmpty()

u ← Q.removeMin()

**vertexVisit(v)**

for all e ∈ G.incidentEdges(u)

{ relax edge e }

z ← G.opposite(u,e)

**preDiscoveryTraversal(G, u, e, z)**

r ← getDistance(u) + weight(e)

if r < getDistance(z)

setDistance(z,r)

**beforeDistanceChange(G,u,e,z)**

Q.replaceKey(z,r) {new method}

**afterDistanceChange(G,u,e,z)**

**postDiscoveryTraversal(G, u, e, z)**

Algorithm findShortestPath(G,s,d)

z<-d {z is a subclass variable}

**initResult( G )**

DijkstraDistances(G, s)

**result(G)**

Algorithm afterDistanceChange*(G,u,e,z)*

*setParent(z,u,e)* {set parent and related edge}

Algorithm result(G)

S<-new Sequence()

while getparent(d)!=s do

S.insertLast(d)

e<-getParentConnectedEdge(d)

S.insertLast(e)

d<- getparent(d)

S.insertLast(d)

S.insertLast(getParentConnectedEdge(d))

S.insertLast(s)

4. Based on either the DFS or the BFS template method algorithms, write the overriding methods so that all nodes in each connected component of a graph G are labeled with a sequence number, i.e., each vertex in a component would be labeled with the same number. For example, each node in the first connected component would be labeled with a 1, each node in the second connected component would be labeled with a 2, etc.

**Answer:**

Algorithm initResult( G )

connectedComponent<-1

{ connectedComponent is a subclass variable}

Algorithm postComponentVisit(G,v)

connectedComponent<- connectedComponent+1

Algorithm startVertexVisit(s)

setLabel(s, connectedComponent)