Problem 1: Connected Components using DFS Template Method Pattern

To compute the connected components of a graph using the DFS Template Method Pattern, we need to extend the provided DFS template and override specific methods to track and identify each connected component.

Steps:

Initialization (initResult(G)):

This method initializes a list components to store representatives of each connected component.

Pre-Component Visit (preComponentVisit(G, v)):

This method is called before visiting a new component. It adds the starting vertex of the new component to the components list, making it a representative of that component.

DFS Algorithm (DFS(G)):

This is the main algorithm that iterates over all vertices. For each unexplored vertex, it starts a DFS traversal (via DFScomponent(G, v)) and marks all reachable vertices as visited.

The DFScomponent method is a standard DFS traversal that labels vertices and edges to track discovery and back edges.

Result (result(G)):

This method returns the list of component representatives collected in components.

Pseudocode Explanation:

Class ConnectedComponents extends DFS

```
Algorithm initResult(G)
    components = []

Algorithm preComponentVisit(G, v)
    components.append(v)
```

Algorithm result(G)

  return components

initResult(G) initializes the list to hold component representatives.

preComponentVisit(G, v) adds the current vertex to the list of representatives when a new component is found.

result(G) returns the list of representatives, each from a different connected component.

Problem 2: Breadth-First Search as a Template Method Pattern

(a) BFS Template Method Pattern

The Breadth-First Search (BFS) algorithm can be adapted to act as a template method by defining hooks for initialization, processing vertices, and processing edges.

Pseudocode Explanation:

```
Algorithm BFS(G, start_vertex)
  initResult(G)
  queue ← empty queue
  setLabel(start_vertex, VISITED)
  queue.enqueue(start_vertex)
  while not queue.isEmpty()
    v ← queue.dequeue()
    processVertex(G, v)
    for all e ∈ G.incidentEdges(v)
      w ← opposite(v, e)
      if getLabel(w) = UNEXPLORED
        setLabel(w, VISITED)
        queue.enqueue(w)
        preEdgeVisit(G, v, e, w)
  return result(G)
```

initResult(G): Initializes data structures needed for BFS.

processVertex(G, v): Processes each vertex as it is dequeued.

preEdgeVisit(G, v, e, w): Processes each edge as it is explored.

result(G): Returns the final result after BFS completes.

(b) Find Path using BFS Template Method

To find the shortest path between two vertices, override the necessary methods to track predecessors and reconstruct the path.

Pseudocode Explanation:

Class PathBFS extends BFS

```
Algorithm initResult(G)
   predecessor = {}

Algorithm preEdgeVisit(G, v, e, w)
   predecessor[w] = v

Algorithm processVertex(G, v)
   if v = target_vertex
      return constructPath(v)

Algorithm constructPath(v)
   path = []
   while v is not None
      path.append(v)
      v = predecessor[v]
   return path.reverse()
```

predecessor keeps track of the parent vertex for each visited vertex.

preEdgeVisit(G, v, e, w) sets the predecessor of w to v.

processVertex(G, v) checks if the current vertex is the target and constructs the path if it is.

constructPath(v) reconstructs the path from the target to the start vertex using the predecessor map.

To use this for finding the path:

```
Algorithm findPath(G, u, v)
   target_vertex = v
   return BFS(G, u)
```

Set target_vertex and initiate BFS from u.

(c) Find Cycle using BFS Template Method

To find a cycle in the graph, override methods to detect and return a simple cycle.

Pseudocode Explanation:

Class CycleBFS extends BFS

```
Algorithm initResult(G)
   parent = {}

Algorithm preEdgeVisit(G, v, e, w)
   if w in parent and parent[v] ≠ w
      return constructCycle(v, w)
   parent[w] = v

Algorithm constructCycle(v, w)
   cycle = [w, v]
   while v ≠ w
      v = parent[v]
      cycle.append(v)
   return cycle
```

parent keeps track of the parent vertex for each visited vertex.

preEdgeVisit(G, v, e, w) checks if w is already visited and is not the parent of v. If so, a cycle is detected and constructCycle(v, w) is called to reconstruct the cycle.

constructCycle(v, w) reconstructs the cycle by following the parent pointers from v and w.

(d) Can DFS be used to find the path with the minimum number of edges?

Answer:

No, DFS cannot guarantee finding the path with the minimum number of edges. DFS explores as far as possible along each branch before backtracking, which may lead to finding a longer path before a shorter one. In contrast, BFS explores all neighbors level by level, ensuring the shortest path in terms of the number of edges.

Problem 4: Labeling Connected Components using DFS or BFS

To label all nodes in each connected component of a graph with a sequence number, override methods to assign labels during traversal.

Pseudocode Explanation:

Class LabelComponentsDFS extends DFS

```
Algorithm initResult(G)
   component_label = {}
   current_label = 0

Algorithm preComponentVisit(G, v)
   labelComponent(G, v)
   current_label = current_label + 1

Algorithm labelComponent(G, v)
   component_label[v] = current_label
   for all e ∈ G.incidentEdges(v)
      w = opposite(v, e)
```

```
        if getLabel(w) = VISITED and w not in component_label
            labelComponent(G, w)
```

```
Algorithm result(G)
    return component_label
```

component_label is a map that stores the label for each vertex.

current_label keeps track of the current component number.

preComponentVisit(G, v) starts labeling a new component.

labelComponent(G, v) labels all reachable vertices from v with current_label.

result(G) returns the map of component labels.

This approach ensures that each connected component is assigned a unique label, allowing easy identification of components.