Kevin Amilund Nilsson, kn222gn

Vertices = V

Edges = E

DFS = In my code the time complexity in my case is O(V + E). This is explained as following, for every node/vertex(V) in the graph the program visits all the successors/edges(E).

/*The worst case in this situation would be if all the vertices were connected to each other in this case it would look like this O(n^2). And in a best case the formula would look like O(V) where there is no successors or edges. */

```
@Override
public List<Node<E>> dfs(DirectedGraph<E> graph) {

   List<Node<E>> visitedNode = new ArrayList<>(); // O(1)
   HashSet<Node<E>> visited = new HashSet<>(); // O(1)

   if(graph.headCount() > 0) // O(1)
   {
       Iterator<Node<E>> head = graph.heads(); // O(1)

       while(head.hasNext()) // O(V)
       {
           visitedNode = dfsRecursive(visitedNode, head.next(),
visited); // O(1)
       }
   }
   else
   {
       visitedNode = dfsRecursive(visitedNode,
graph.getNodeFor(graph.allItems().get(0)), visited); // O(1)
   }

   return visitedNode; // O(1)
}

private List<Node<E>> dfsRecursive(List<Node<E>> visitedNode,
Node<E> root, HashSet<Node<E>> visited){

   root.num = visitedNode.size(); // O(1)

   visitedNode.add(root); // O(1)
   visited.add(root); // O(1)
```

```
    Iterator<Node<E>> successor = root.succsOf(); // O(1)

    while (successor.hasNext()) { // O(E)

        Node<E> node = successor.next(); // O(1)

        if(!visited.contains(node)) { // O(1)
            visitedNode = dfsRecursive(visitedNode, node, visited);
// O(1)
        }
    }
    return visitedNode; // O(1)
}
```

BFS = The time complexity is the same for bfs as it is for the dfs O(V + E) since for every vertex in the graph all the Edges is visited as well.

```
@Override
public List<Node<E>> bfs(DirectedGraph<E> graph) {

    List<Node<E>> result = new ArrayList<>(); //O(1)
    HashSet<Node<E>> toVisit = new HashSet<>(); //O(1)
    HashSet<Node<E>> visited = new HashSet<>(); //O(1)

    toVisit.clear(); //O(1)
    result.clear(); //O(1)
    visited.clear(); //O(1)

    Iterator<Node<E>> heads = graph.heads(); //O(1)

    if(graph.headCount() != 0){ //O(1)

        while(heads.hasNext()){ //O(V)

            toVisit.clear(); //O(1)
            toVisit.add(heads.next()); //O(1)
            bfsRecursive(toVisit, result, visited); //O(1)
        }
    }
    else{
        toVisit.clear(); //O(1)
        toVisit.add(graph.getNodeFor(graph.allItems().get(0)));
//O(1)

        bfsRecursive(toVisit, result, visited); //O(1)
    }

    return result; //O(1)
}

private List<Node<E>> bfsRecursive(HashSet<Node<E>>toVisit,
List<Node<E>> result, HashSet<Node<E>> visited){

    Iterator<Node<E>> iterator = toVisit.iterator();// O(1)
    toVisit = new HashSet<>();//O(1)

    while(iterator.hasNext()){ // O(V)

        Node<E> node = iterator.next(); // O(1)
        if(!visited.contains(node)){ // O(1)

            node.num = result.size(); // O(1)
            visited.add(node); // O(1)
            if(!result.contains(node)){ // O(1)

                result.add(node); // O(1)
            }
        }
        Iterator<Node<E>> successor = node.succsOf(); // O(1)

        while(successor.hasNext()){ // O(E)

            Node<E> successor2 = successor.next(); // O(1)

            if(!visited.contains(successor2)){ // O(1)

                toVisit.add(successor2); // O(1)
            }
        }
    }

    if(!toVisit.isEmpty()){ // O(1)

        bfsRecursive(toVisit, result, visited); // O(V)
    }
    return result; // O(1)
}
```

Transitive closure - O(V^2 + VE), since the DFS is called in the class this must be calculated in the time complexity and the class also iterates through all the next() elements. For every Vertex in the graph there is a call to the DFS (the Transitive closure uses the DFS for every node in the graph).

```
@Override
public Map<Node<E>, Collection<Node<E>>>
computeClosure(DirectedGraph<E> dg) {

   Collection<Node<E>> nodeCollection; // O(1)
   Map<Node<E>, Collection<Node<E>>> map = new HashMap<>(); //
O(1)

   MyDFS<E> dfs = new MyDFS<>(); // O(1)

   Iterator<Node<E>> iterator = dg.iterator(); // O(1)

   while(iterator.hasNext()){ // O(V)

       Node<E> node = iterator.next(); // O(1)
       nodeCollection = dfs.dfs(dg, node); // O(V + E)
       map.put(node, nodeCollection); // O(1)
   }

   return map; // O(1)
}
```

Connected component - The function calls the dfs for each node in the graph. So the Time for that function will have to be calculated to. The DFS time is O(V^2 + VE) and in combination with the other loops in the function the time complexity would look something like this - `O(V^2 + VE + (V^5)) or O(V^2 + (VE)) depending on how you add.`

V(V+E + (V)) --- V(V+E + (V)(V)) --- V(V+E + (V)(V)(V)) --- V(V+E + (V)(V)(V)(V)) --- V(V+E + (V)(V)(V)(V)(V)) --- V(V+E + (V)(V)(V)(V)(V)(V)) --- O(V^2 + VE + (V^5))

```
@Override
public Collection<Collection<Node<E>>>
computeComponents(DirectedGraph<E> dg) {

   boolean connectionKey; // O(1)
   Collection<Node> visitedNode = new HashSet<>(); // O(1)
   Collection<Collection<Node<E>>> collectionNodes = new
HashSet<>(); // O(1)
   Collection<Node<E>> component; // O(1)
   List<Node<E>> dfsList; // O(1)
   MyDFS<E> dfs = new MyDFS<>(); // O(1)

   Iterator<Node<E>> iterator = dg.iterator(); // O(1)

   while(iterator.hasNext()){ // O(V)

       Node<E> nextNode = iterator.next(); // O(1)
       connectionKey = false; // O(1)

       if(!visitedNode.contains(nextNode)){ // O(1)

           dfsList = dfs.dfs(dg, nextNode); // O(V + E)

           for(Node<E> itemInNode : dfsList){ // O(V) --- O(V^2 +
VE) + V^2) = O(2V^2 + (VE))

               if(visitedNode.contains(itemInNode)) { // O(1)

                   for (Collection<Node<E>> c : collectionNodes) {
// O(V) --- O(V^2 + VE) + V(V^2) =  O(3V^2 + (VE))

                       if (c.contains(itemInNode)) { // O(1) ---
//O(V^2 + VE) + V(V^2)(V))

                           visitedNode.addAll(dfsList); // O(V)
--- O(V^2 + VE) + V(V^2)(V)(V)) =  O(4V^2 + (VE))
                           c.addAll(dfsList); // O(V) --- O(V^2 +
VE) + V(V^2)(V)(V)(V)) =  O(5V^2 + (VE))
                           connectionKey = true; // O(1)
                       }
                   }
               }
           }
           if(connectionKey == false){

               component = new HashSet<>(); // O(1)
               visitedNode.addAll(dfsList); // O(V)
               component.addAll(dfsList); // O(V)
               collectionNodes.add(component); // O(1)
           }
       }
       visitedNode.add(nextNode); // O(1)
   }

   return collectionNodes; // O(1)}
```