I am using V(Vertex) for nodes and E(Edges) for the edges. For most of the complexities for functions I have been using http://bigocheatsheet.com/ and http://docs.oracle.com/
Specifically on docs.oracle I have used
http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html

DFS

For each vertex(V) in the graph we visit the node and its edges(E). The average time complexity case will be O(V + E) where the graph visits each vertex and each vertex successor.

```
@Override
public List<Node<E>> dfs(DirectedGraph<E> graph)
{
  LinkedList<Node<E>> list = new LinkedList<>(); //O(1)
  Set<Node<E>> visited = new HashSet<>(); //O(1)
  if(graph.headCount() != 0)  //O(1)
  {
     Iterator<Node<E>> headItr = graph.heads();//O(1)
     while(headItr.hasNext()) // O(V)
     {
        dfs(headItr.next(), list, visited); // O(1)
     }
  }
  else
  {
     dfs(graph.getNodeFor(graph.allItems().get(0)), list, visited); //O(1)
  }
  return list; // O(1)
}

private List<Node<E>> dfs(Node<E> root, LinkedList<Node<E>> list, Set<Node<E>> visited)
{
  if(!visited.contains(root)) {
     root.num = list.size();//O(1)
     list.add(root);//O(1)
     visited.add(root);//O(1)
     Iterator<Node<E>> it = root.succsOf();//O(1)
     while (it.hasNext()) { //O(E)
        dfs(it.next(), list, visited);//O(V)
     }
  }
  return list;//O(1)
}
```

The above code will result in O(V + E) as all the constants are removed. In the case that there are heads the result will be O(2V + E) and in the case with no heads O(V + E) both are written as O(V + E)

BFS

```
@Override
public List<Node<E>> bfs(DirectedGraph<E> graph) {
  Set<Node<E>> set = new HashSet<>();O(1)
  if(graph.headCount() != 0)//O(1)
  {
    Iterator<Node<E>> headItr = graph.heads();//O(1)
    while(headItr.hasNext())//O(V)
    {
      set.add(headItr.next());//O(1)
    }
  }
  else
  {
    set.add(graph.getNodeFor(graph.allItems().get(0)));//O(1)
  }
  bfs(set);//O(1)
  return list;
}
public void bfs(Set<Node<E>> set)
{
  if(set.isEmpty())//O(1){ return; }
  Iterator<Node<E>> itr = set.iterator();//O(1)
  set = new LinkedHashSet<>();//O(1)
  while (itr.hasNext())//O(V)
  {
    Node<E> node = itr.next();
    if(!visited.contains(node))////O(1)
    {
      visited.add(node);//O(1)
      node.num = list.size();//O(1)
      list.add(node);//O(1)
    }
    Iterator<Node<E>> successors = node.succsOf();//O(1)
    while(successors.hasNext())//O(E)
    {
      Node<E> n = successors.next();//O(1)
      if(!visited.contains(n))//O(1)
      {
        set.add(n);//O(1)
      }
    }
  }
  bfs(set);//O(V)
}
```

This is pretty much the same as the DFS. In the worst case scenario the case will be O(3V + E) while the best case it will be O(2V + E) both are written as O(V + E) as you remove the constants.

Transitive closure

```java
@Override
public Map<Node<E>, Collection<Node<E>>> computeClosure(DirectedGraph<E> dg)
{
  MyDFS<E> dfs = new MyDFS<>();//O(1)
  Map<Node<E>, Collection<Node<E>>> map = new HashMap<>();//O(1)
  Iterator<Node<E>> itr = dg.iterator();//O(1)
  while (itr.hasNext())//O(V)
  {
     Node<E> node = itr.next();//O(1)
     map.put(node, dfs.dfs(dg, node));//DFS is O(V+E)
  }
  return map;//O(1)
}
```

$O(V)*O(V+E) = O(V^2+VE)$

The transitive closure uses the dfs in a while loop to compute the closure. The the case will therefore be $O(V^2+VE)$

Connected Components

```
@Override
public Collection<Collection<Node<E>>> computeComponents(DirectedGraph<E> dg) {
  Collection<Collection<Node<E>>> toReturn = new HashSet<>();//O(1)
  Map<Node<E>, Collection<Node<E>>> visited = new HashMap<>();//O(1)
  Set<Node<E>> toVisit = new HashSet<>();//O(1)
  MyDFS<E> dfs = new MyDFS<>();//O(1)
  dg.iterator().forEachRemaining(toVisit::add); //O(V)                    ----> V
  Iterator<Node<E>> itr = dg.heads();
  while(!toVisit.isEmpty() && itr.hasNext())//O(V) {                      ----> V+V
     Node<E> head = itr.next();//O(1)
     if(!visited.containsKey(head))//O(1) {
        HashSet<Node<E>> toAdd = new HashSet<>();//O(1)
        boolean merge = false;//O(1)
        Node<E> tmpNode = null;//O(1)
        List<Node<E>> dfsList = dfs.dfs(dg, head);//O(V+E)    ----> V+V((V+E))
        Iterator<Node<E>> iterator = dfsList.iterator();//O(1)
        while(iterator.hasNext()) { ----> V+V((V+E)+V)
           Node<E> node = iterator.next();
           if(!visited.containsKey(node))//O(1) {
              toAdd.add(node);//O(1)
              visited.put(node, toAdd);//O(1)
           } else{
              merge = true;//O(1)
              tmpNode = node;//O(1)
           }
        }
        if(merge){
           visited.get(tmpNode).addAll(toAdd);//O(V)    ----> V+V((V+E)+V+V)
        }  else{
           toReturn.add(toAdd);//O(1)
        }
        toVisit.removeAll(dfsList);//O(V)             ----> V+V((V+E)+V+V+V)
     }
  }
  itr = toVisit.iterator();//O(1)
  while(itr.hasNext())//O(V)              ----> V+V((V+E)+V+V+V)+V
  {
     Node<E> n = itr.next();//O(1)
     if(!visited.containsKey(n))//O(1)
     {
        List<Node<E>> dfsList = dfs.dfs(dg, n);//O(V+E)                 ----> V+V((V+E)+V+V+V)+V((V+E))
        dfsList.iterator().forEachRemaining(node -> visited.put(node, dfsList));//O(V)----> V+V((V+E)+V+V+V)+V((V+E)+V)
        toReturn.add(dfsList);//O(1)
     }
     itr.remove();//O(1)
  }
  return toReturn;
}
```

$V+V((V+E)+V+V+V)+V((V+E)+V) = V+6V^2+2VE = \mathbf{V^2+VE+V}$

This method uses the dfs multiple times to get all connected components. The method has a time complexity of V^2+VE+V.