Collin Gros
10/30/2019
CS-372-M01

# LAB 6

## INTRODUCTION

Representing data as a graph is useful in the real world. Big data, like the kind that Google collects, must be traversed quickly and efficiently. Since trading memory for speed isn't viable in a big data setting, correct algorithms must be used so that processes can finish in a reasonable amount of time.

We know how to traverse and identify connected components, and strongly connected components on small graphs, but we need to make it possible for a computer to analyze extremely large graphs.

## METHODS

For finding connected components, I modified my DFS_recursive function to increment the number of connected components in the graph every time explore is called from a new vertex, and I assign each node its corresponding connected component in explore.

```
find_connected_components:
        DFS_recursive(G)

        for each node in G:
                at current spot in vector of connected component ids, assign node's cc id
                (cc_ids[count] = cc_id)
```

For finding strongly connected components, I used Kosaraju's algorithm and the solution to Homework 4. The pseudocode for my reverse function is the solution to 3.5 in Homework 4 (was provided), and the pseudocode for Kosaraju's algorithm was on Wikipedia.

(Kosaraju's algorithm:
"For each vertex *u* of the graph, mark *u* as unvisited. Let *L* be empty.

1. For each vertex *u* of the graph do Visit(*u*), where Visit(*u*) is the recursive subroutine:
   If *u* is unvisited then:
   1. Mark *u* as visited.
   2. For each out-neighbour *v* of *u*, do Visit(*v*).
   3. Prepend *u* to *L*.
2. Otherwise do nothing.
3. For each element *u* of *L* in order, do Assign(*u*,*u*) where Assign(*u*,*root*) is the recursive subroutine:
   If *u* has not been assigned to a component then:
   1. Assign *u* as belonging to the component whose root is *root*.
   2. For each in-neighbour *v* of *u*, do Assign(*v*,*root*).

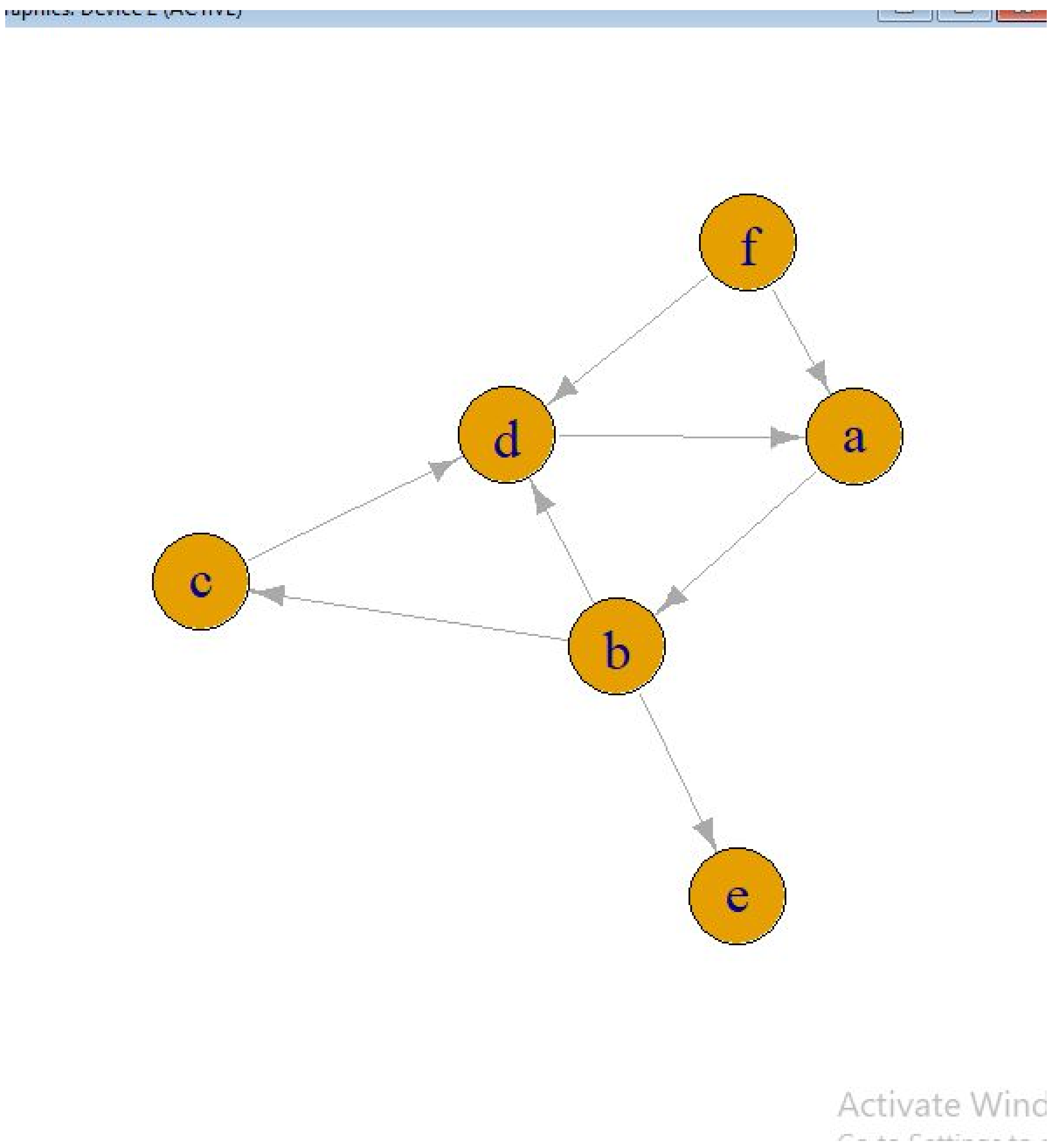
4. Otherwise do nothing.

"

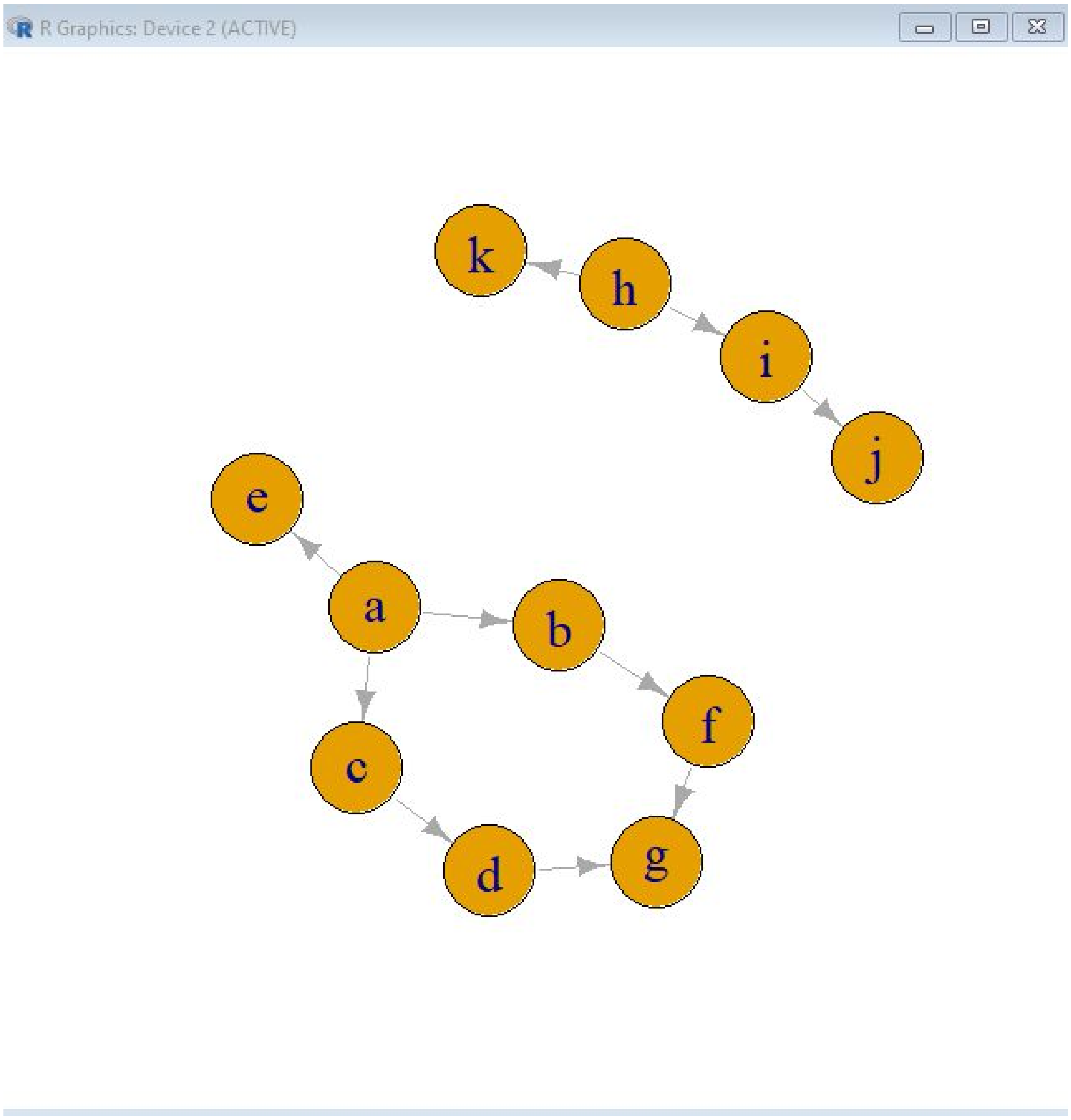
**RESULTS**
After testing my code using my test graphs, I've found that my strongly connected components function is sometimes wrong, but for the majority of input, correct. This is a huge problem that I need to fix, as my data I collected here may be completely wrong because of it. I've struggled with this bug for a long time, and I suppose I will be able to find it soon. However, finding connected components works perfectly, as well as reversing the graph. I expected the reverse graph function to greatly impact runtime, but it appears to have a miniscule amount of impact.

My ten test graphs used to test find_connected_components and find_strongly_connected_components:

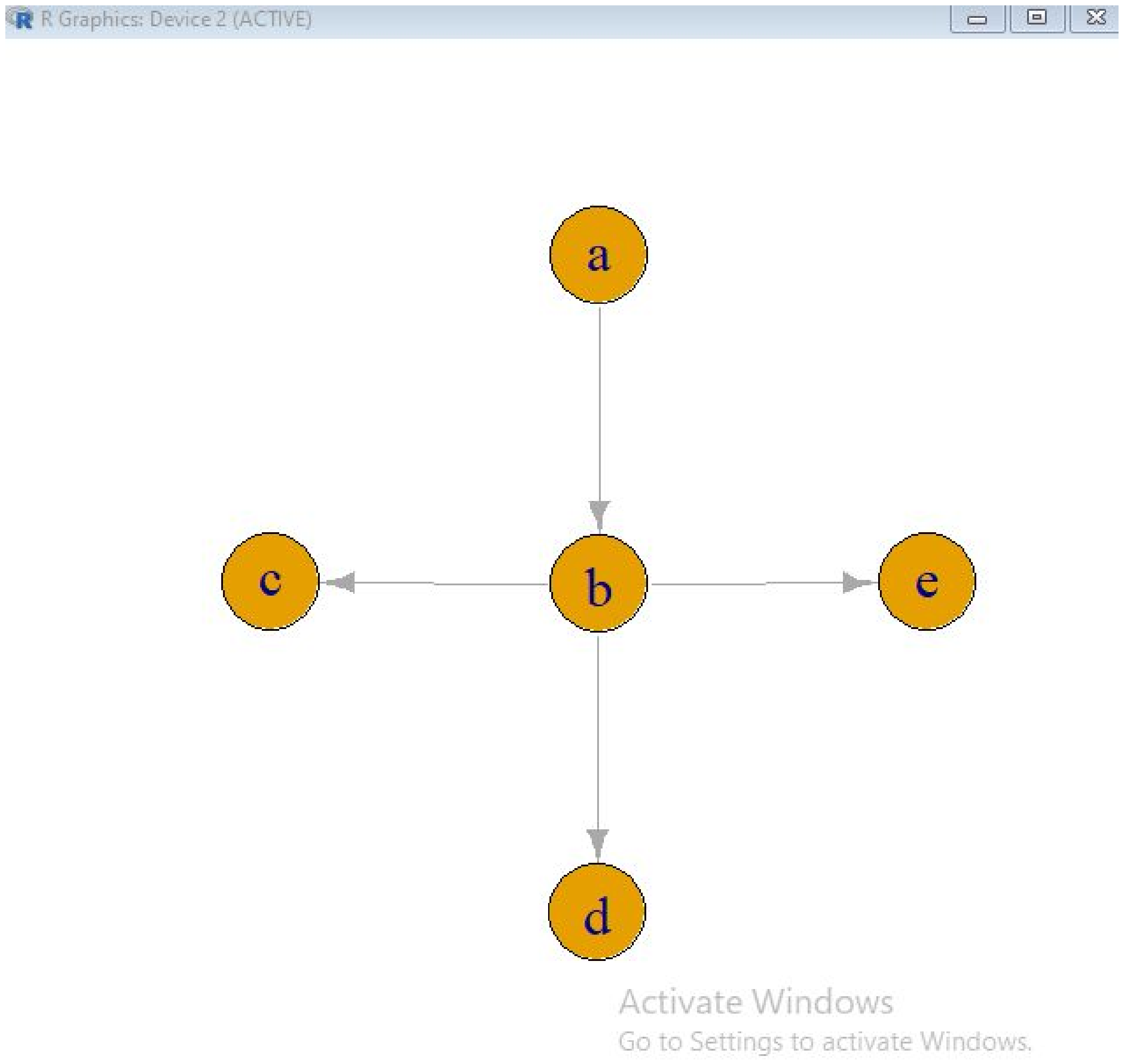
<u>DIRECTED</u>
acyclic 1:



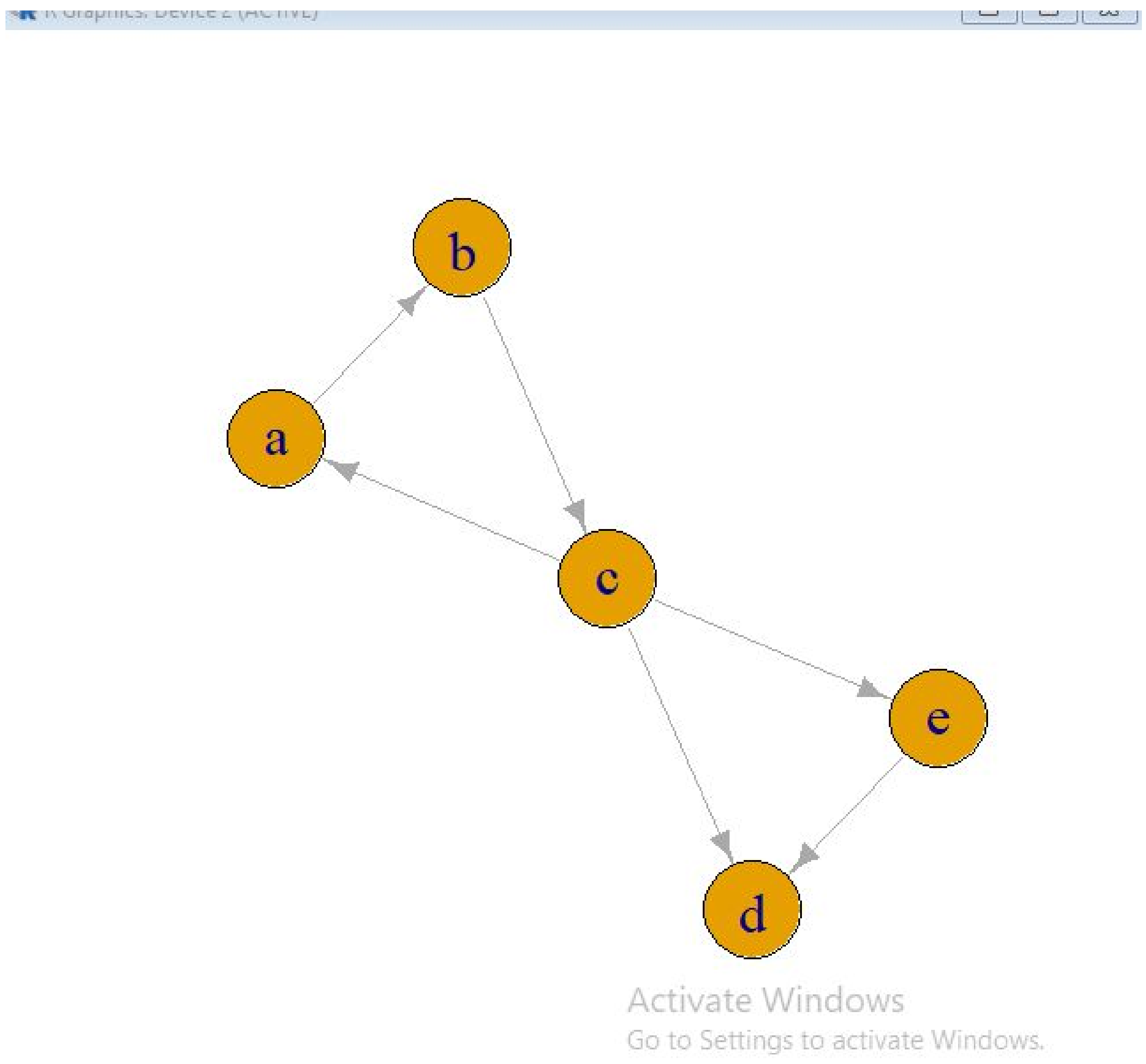
acyclic 2:



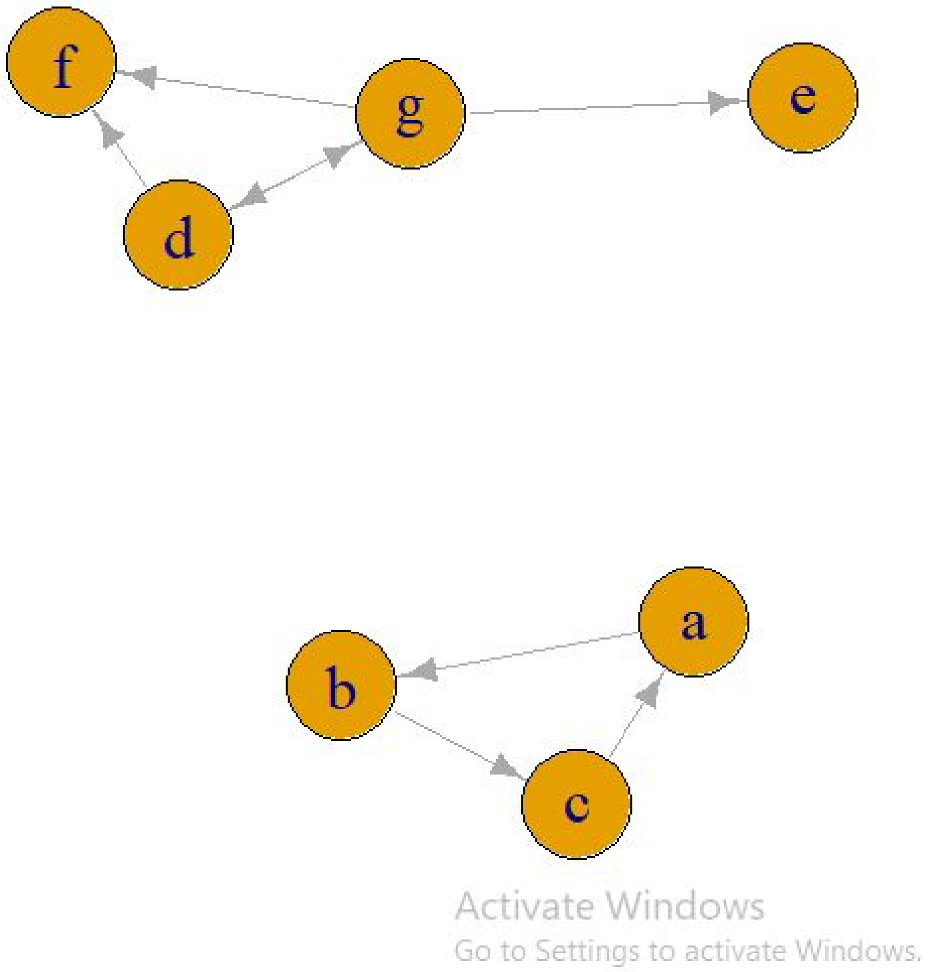
acyclic 3:


R Graphics: Device 2 (ACTIVE)
a
b
c
d
e
Activate Windows
Go to Settings to activate Windows.

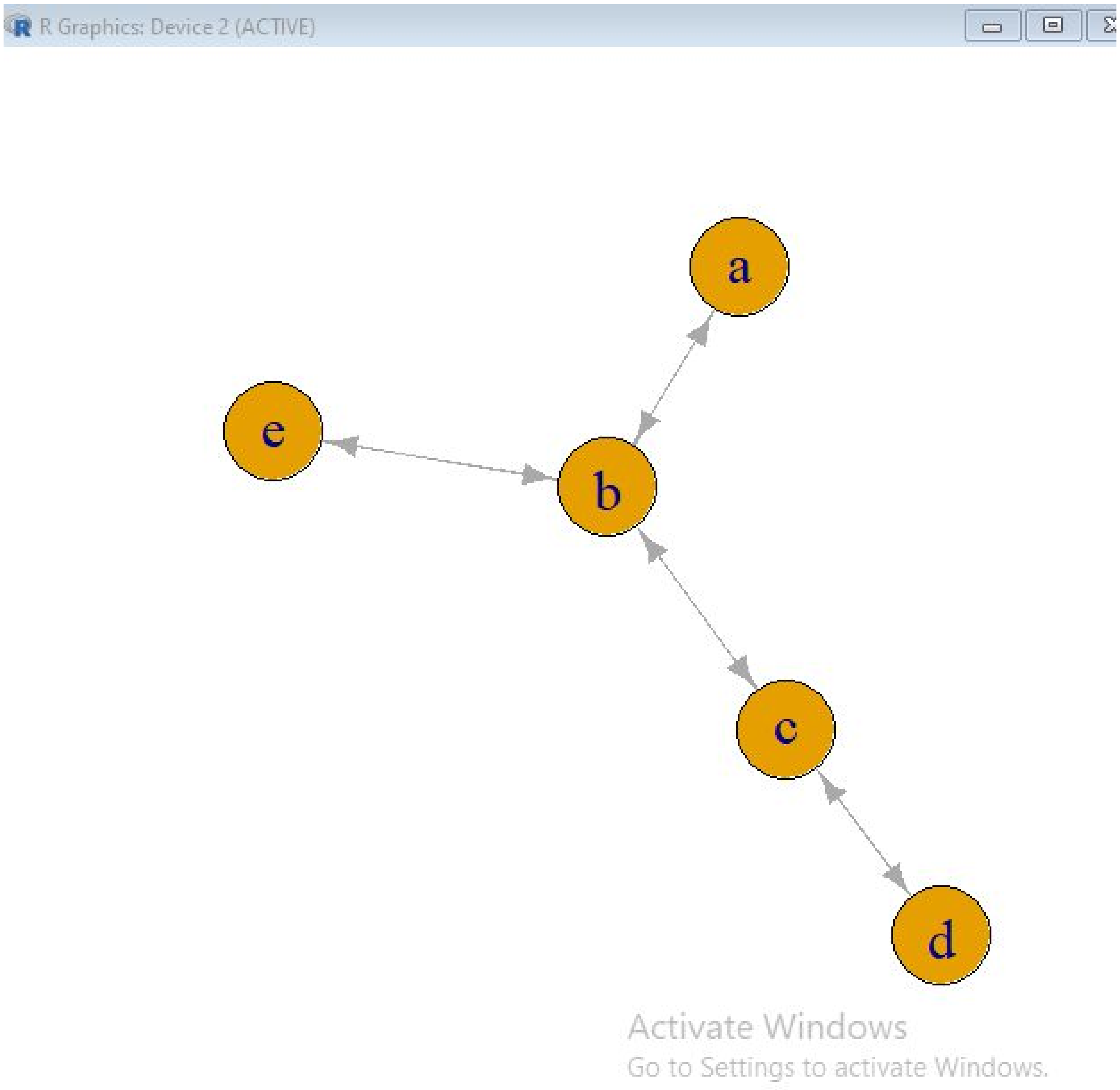
cyclic 1:



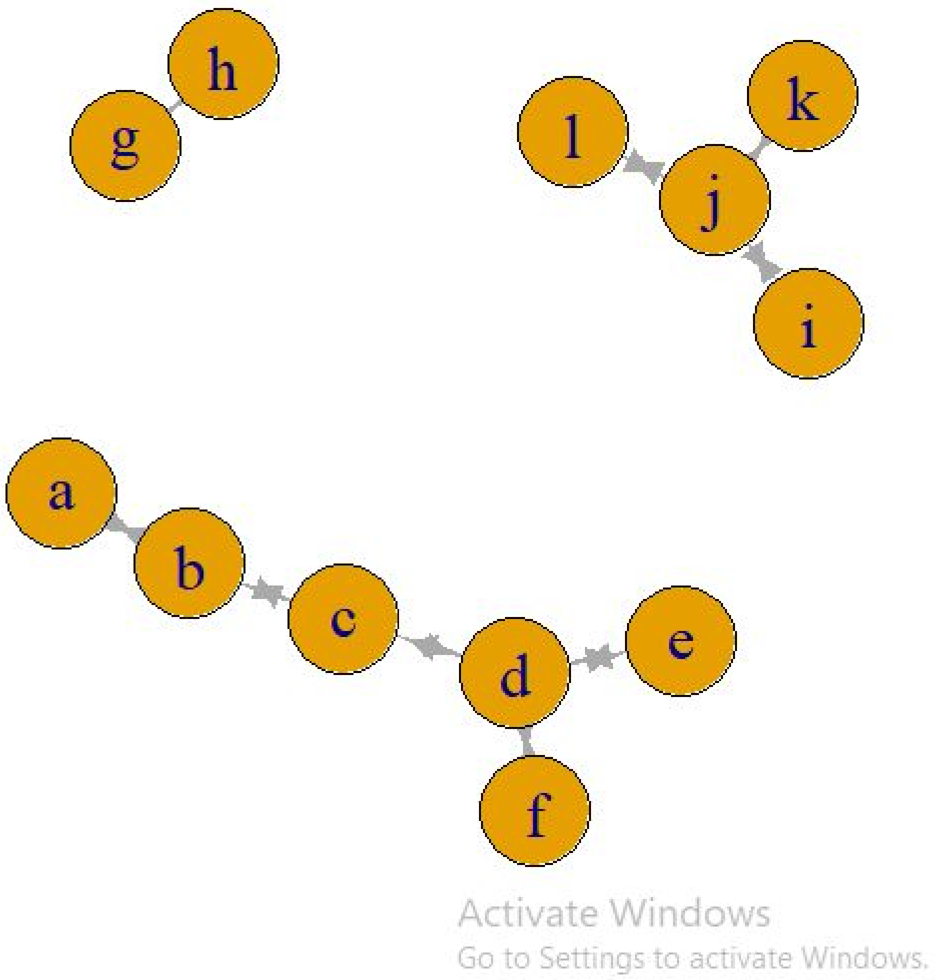
cyclic 2:



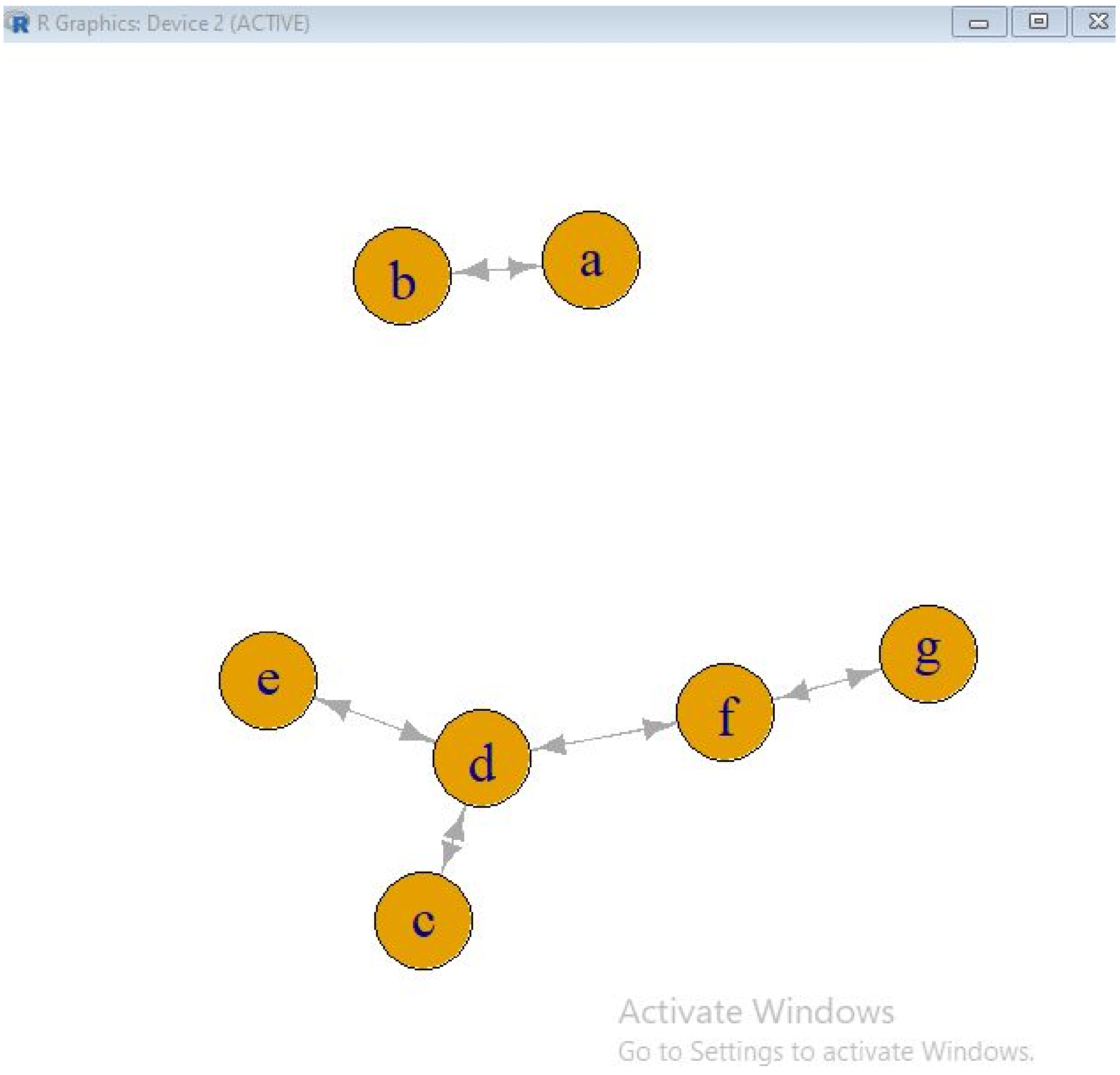
<u>UNDIRECTED</u>
acyclic 1:


R Graphics: Device 2 (ACTIVE)
a
e
b
c
d
Activate Windows
Go to Settings to activate Windows.

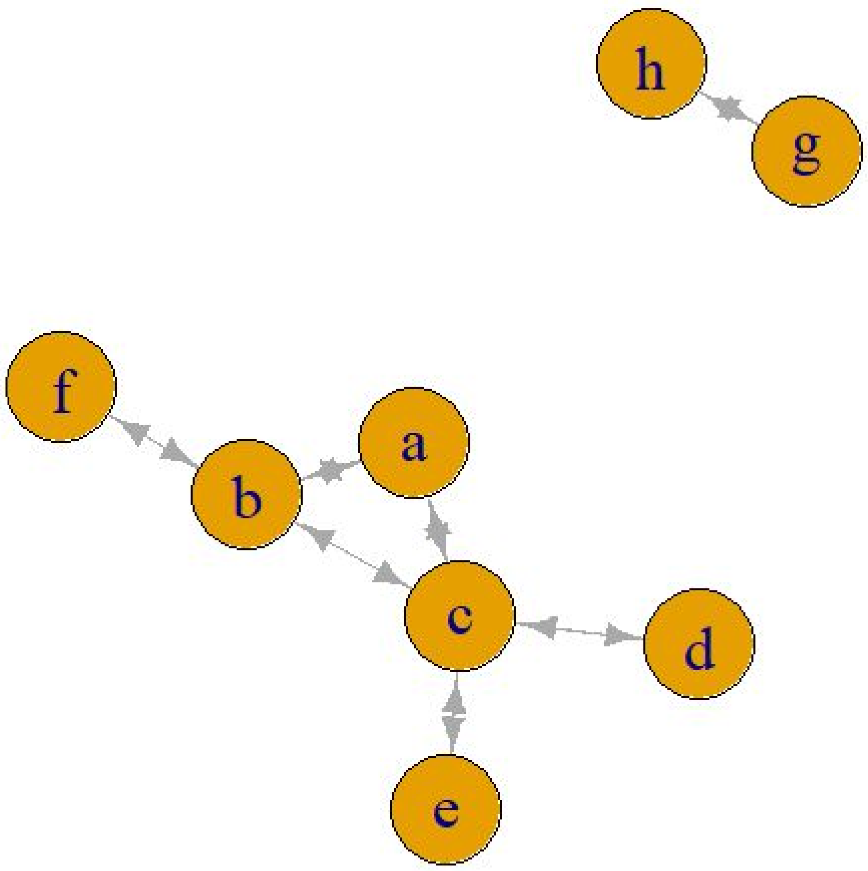
acyclic 2:



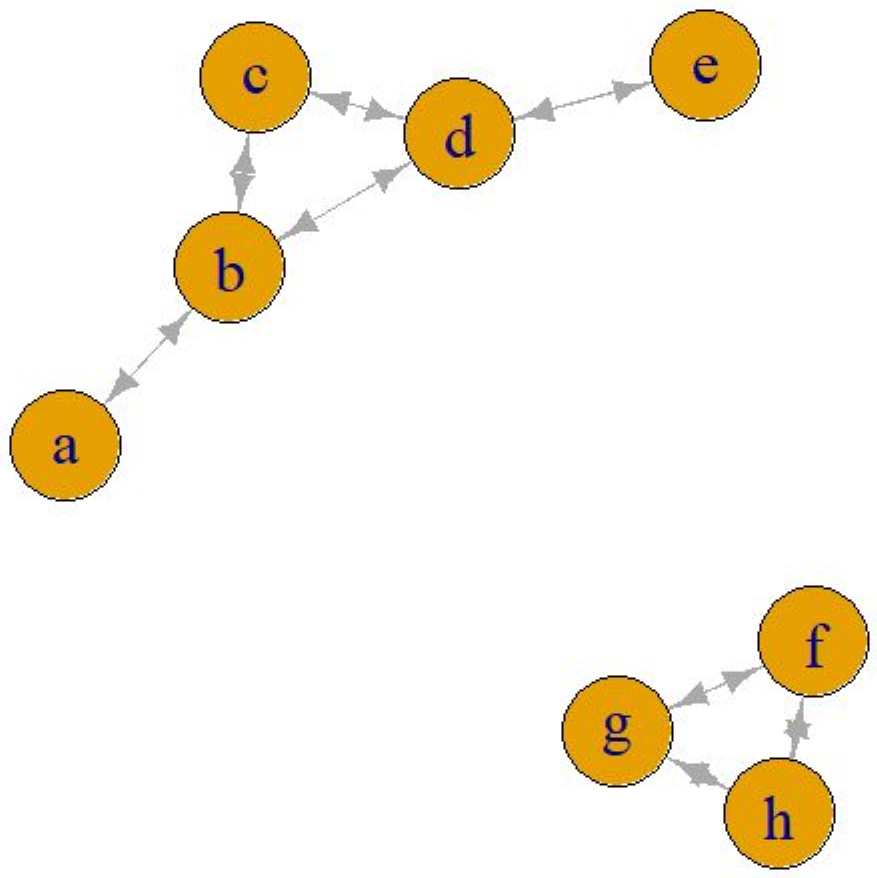
acyclic 3:


R Graphics: Device 2 (ACTIVE)
b
a
e
d
f
g
c
Activate Windows
Go to Settings to activate Windows.

cyclic 1:

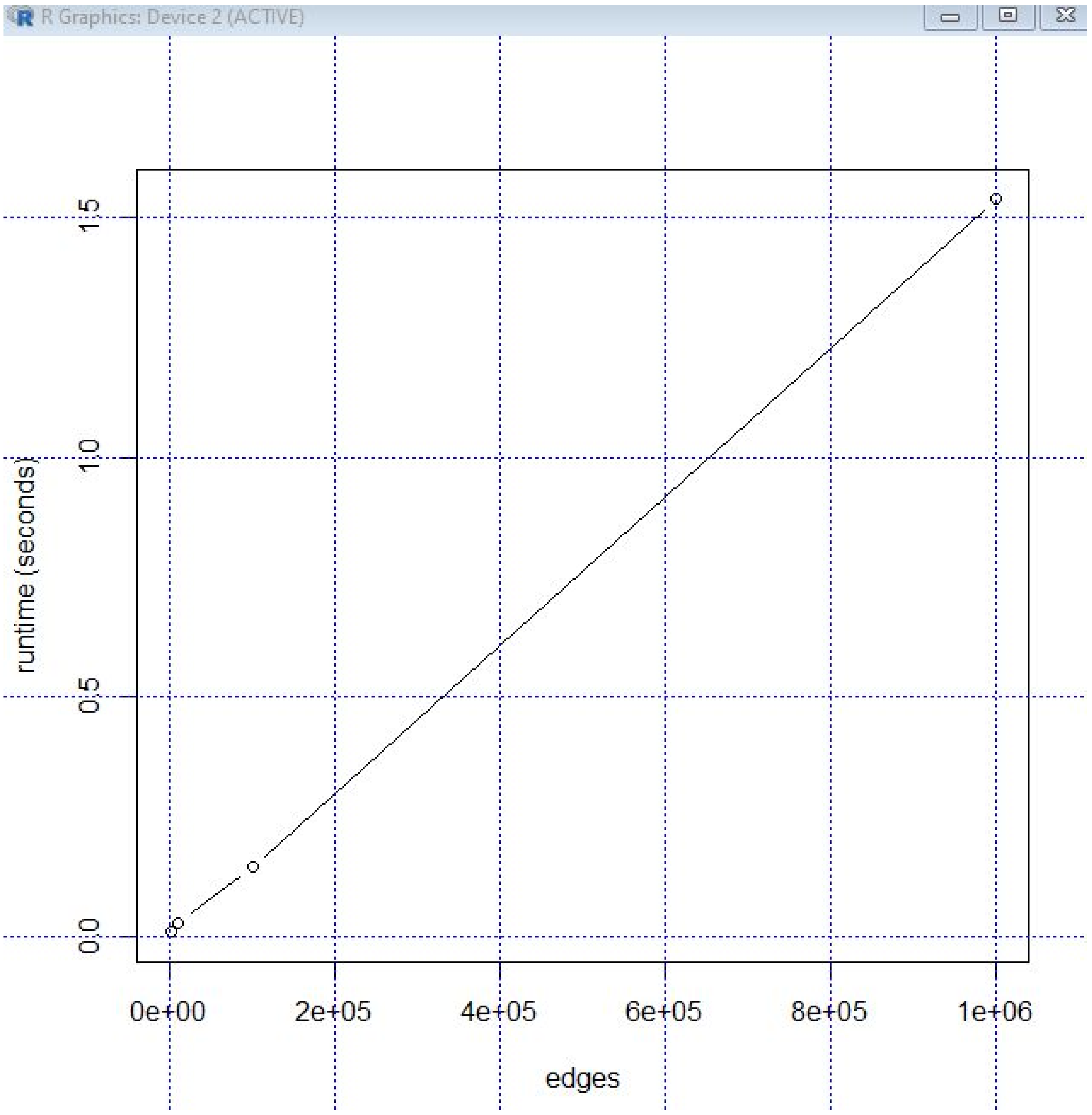


cyclic 2:

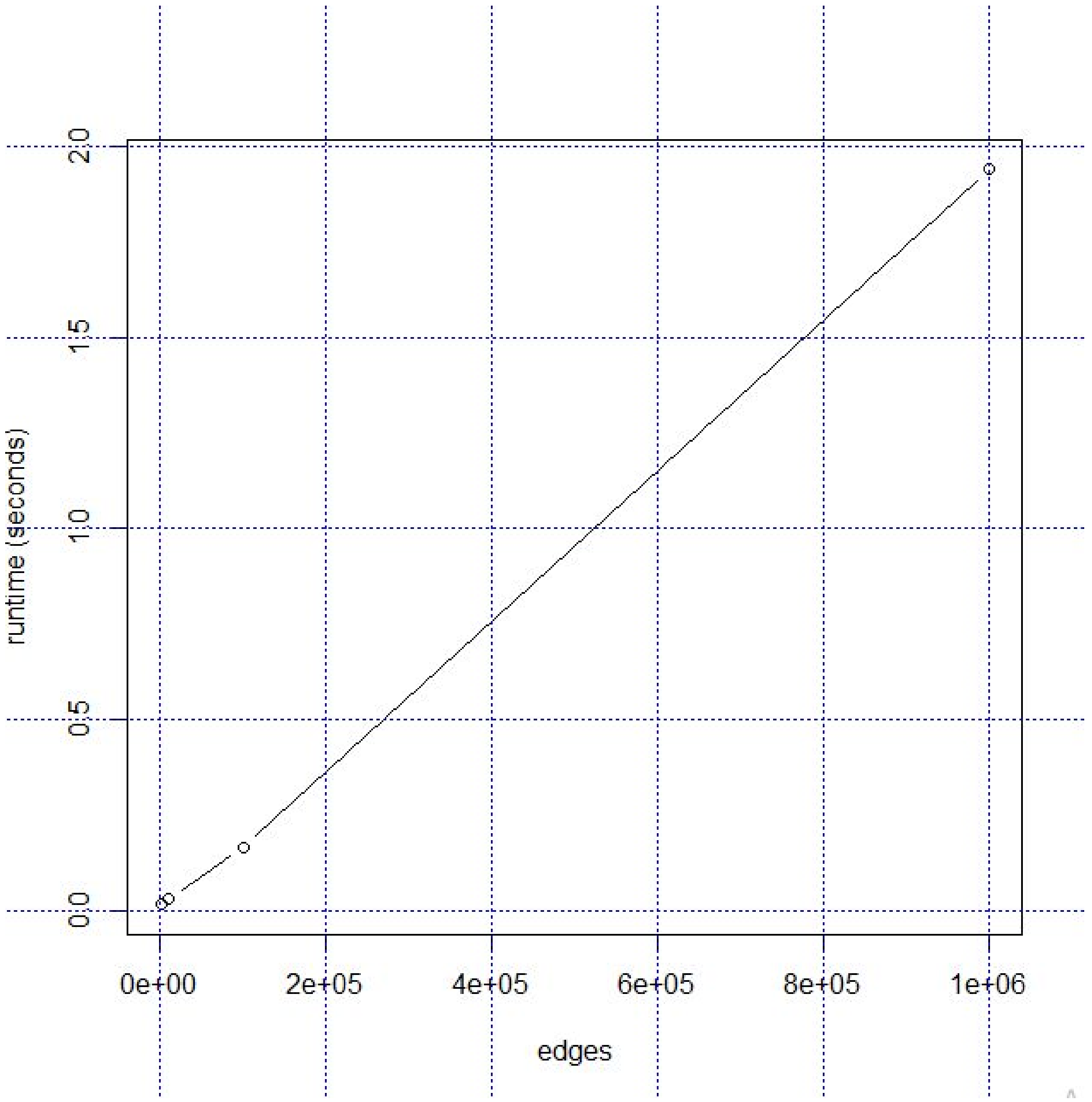


The following are my runtime graphs:

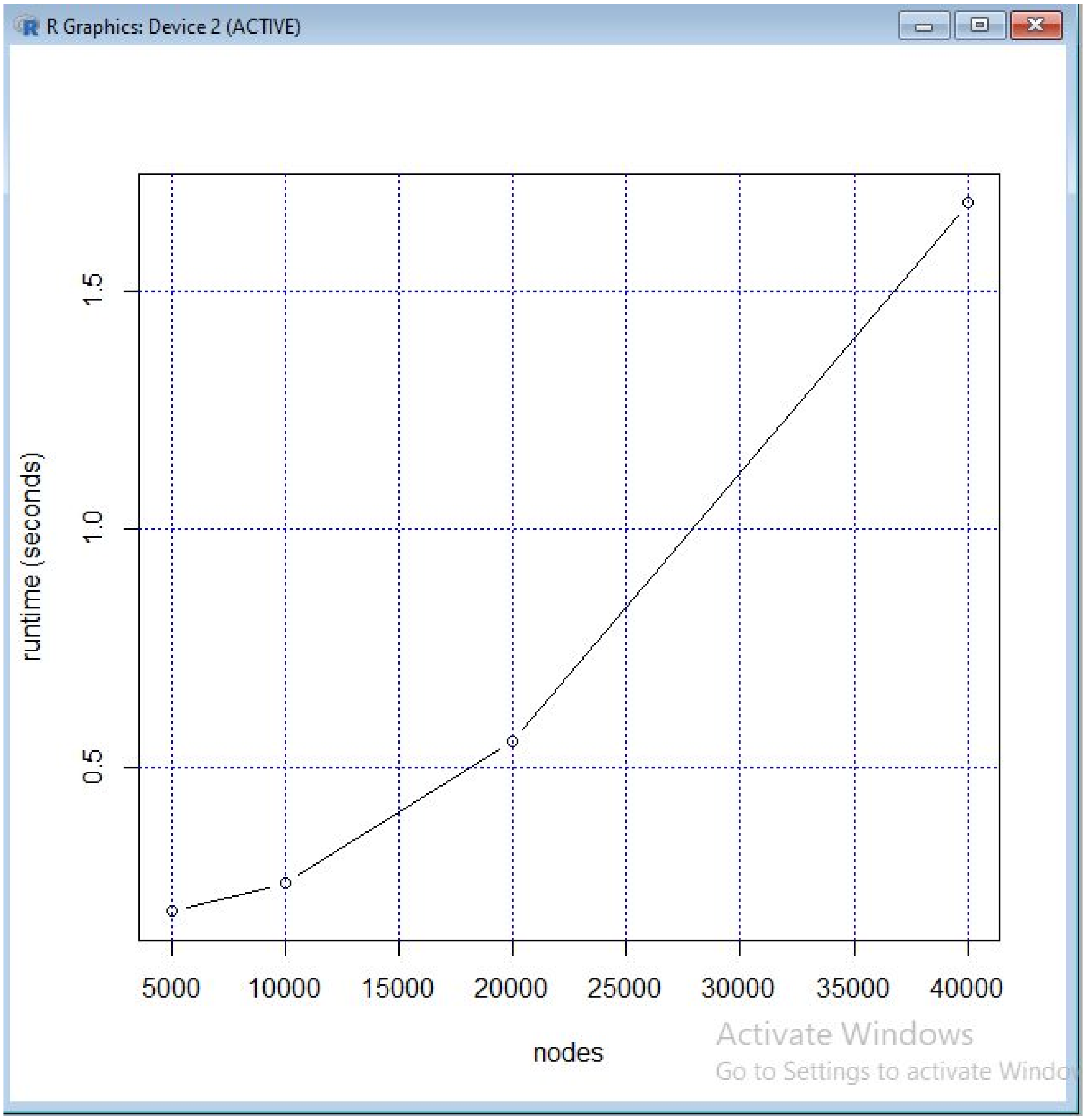
connected components as a function of edges



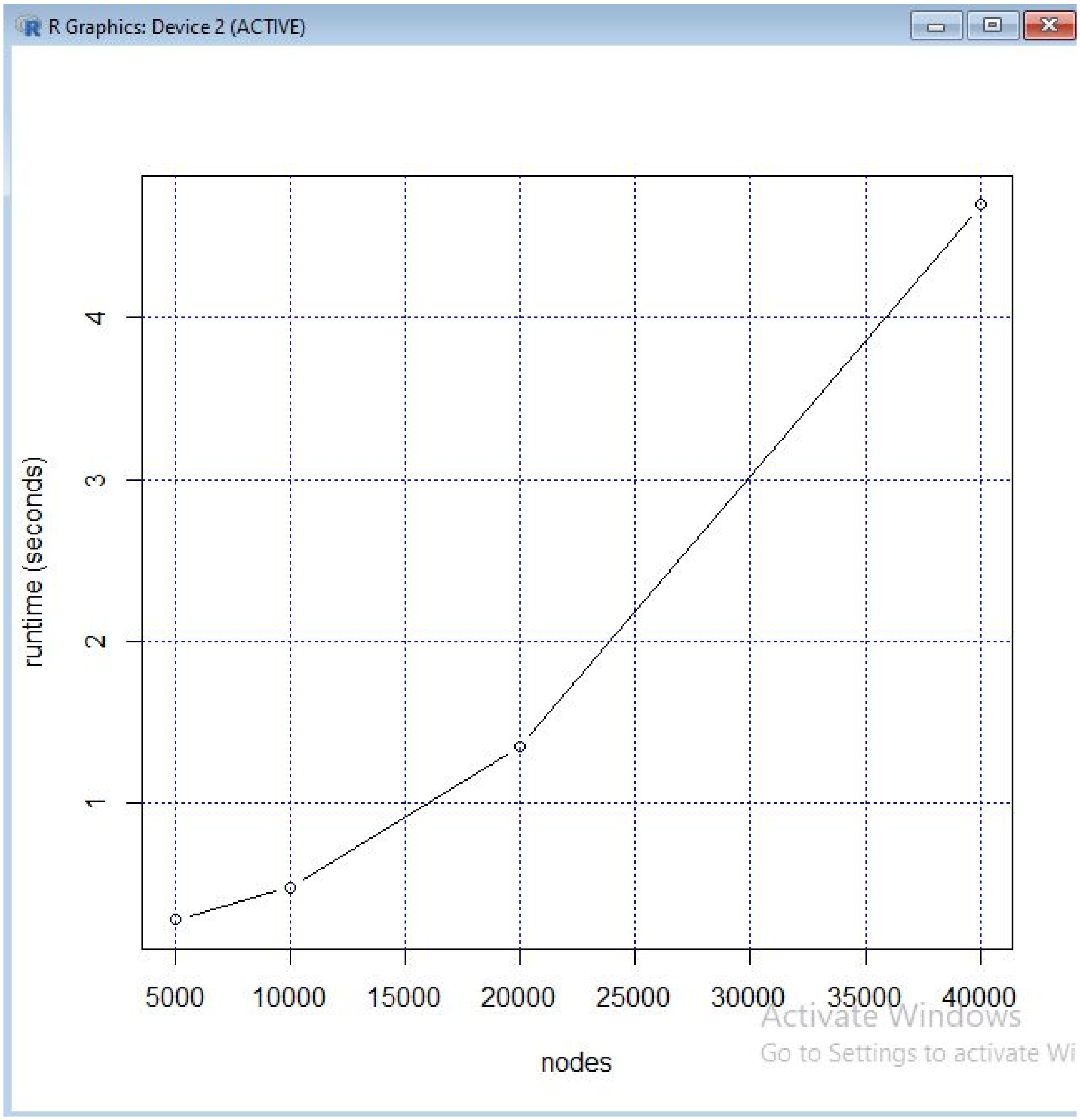
strongly connected components as a function of edges



connected components as a function of nodes



strongly connected components as a function of nodes



The graphs that are functions of nodes do NOT look linear. This may be because of some part of my code that inefficiently completes DFS_recursive. However, the graphs that are functions of edges do look linear! I was surprised to see that reversing the graph did not greatly impact the runtime of strongly connected components. At large values, there is almost an indiscernible difference between the find strongly connected and find connected components functions.

**DISCUSSION**

In my previous lab, I saw that iterative solutions were always slower than the recursive ones. I'm still not exactly sure what causes that, but since the recursive solution was always faster, I used it. It may be possible that this result has only happened to me, due to inefficient or bad code, but if not, I can't really think of any reason the recursive function should be faster.

**CONCLUSIONS**

Because of the quick algorithms that people have come up with today, we can complete extremely complex and large problems, like find connected or strongly components of a huge graph, very quickly. Now that we can find connected components, we may make our code even more versatile by finding a minimum spanning tree.