# Finding connected components with SAS/Base

Youri BAEYENS

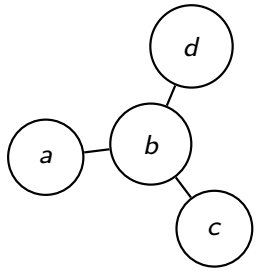
# Outline

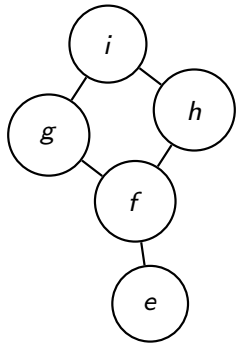
# In business language

# Commonly used dataset structure

Is_in_relation_with:

| Id_A | Id_B |
|---|---|
| a | b |
| b | c |
| b | d |
| e | f |
| f | g |
| g | i |
| h | i |
| f | h |



# Common questions

Questions:

- Are a and d connected?
- Who is connected to f?

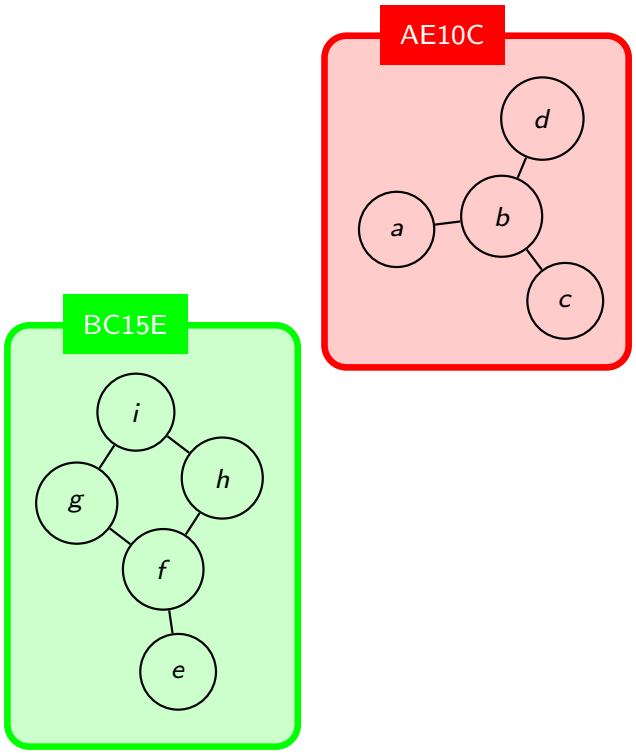
Wishes:

- A SAS macro to address these questions
- Very performant in order to process datasets with millions of records
- Easy to deploy (SAS/Base)

# Desired output

Is_member_of:

| Id | Member_of |
|---|---|
| a | AE10C |
| b | AE10C |
| c | AE10C |
| d | AE10C |
| e | BC15E |
| f | BC15E |
| g | BC15E |
| h | BC15E |



# Meaning of labels

Is_member_of:

| Id | Member_of |
|---|---|
| a | AE10C |
| b | AE10C |
| c | AE10C |
| d | AE10C |
| e | BC15E |
| f | BC15E |
| g | BC15E |
| h | BC15E |

"Member_of" labels are arbitrary codes. Could also be the Id of any member of the group.

# In mathematical language

# Problem classification

Let $G = (V, E)$ denote an **undirected** graph:

- $V$ are vertices
- $E$ are edges
- $E \subset V \times V$

## Problem to solve

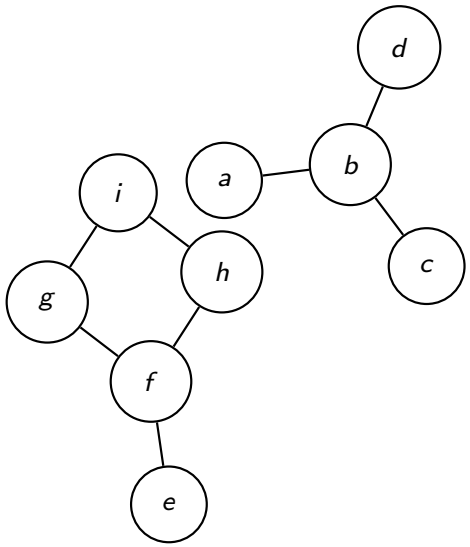
Find connected components of G, ie, find subgraphs $G_1, \ldots, G_k$ $\forall i \in 1 \ldots k$ such that:

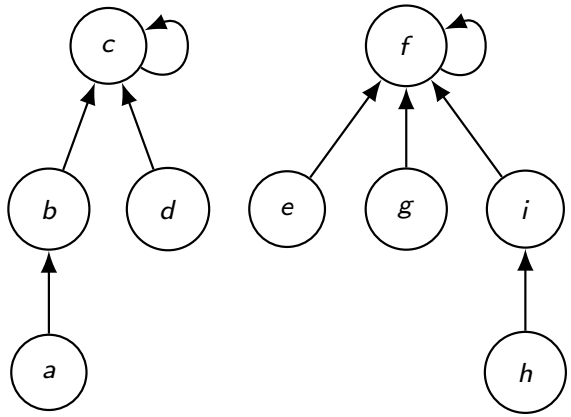
- any two vertices of $G_i$ are connected by a path
- none of $G_i$'s vertices are connected to other $G_j$'s vertices (where $j \neq i$)

# In pseudo-language

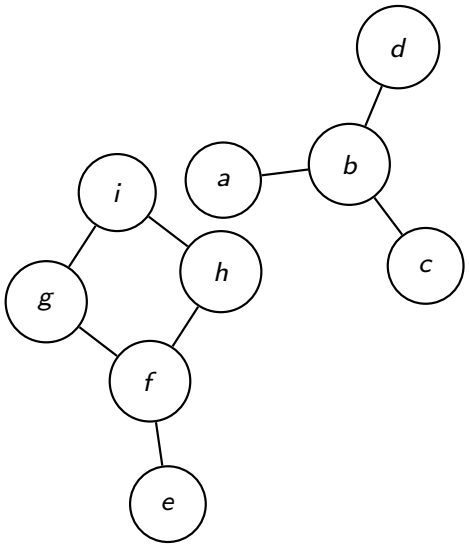
# Trees to find connected components

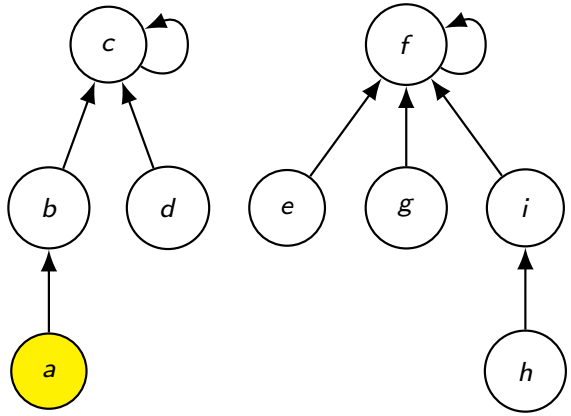
- We'll use trees to identify the connected components the vertex belongs to.
- Tree root = connected component label



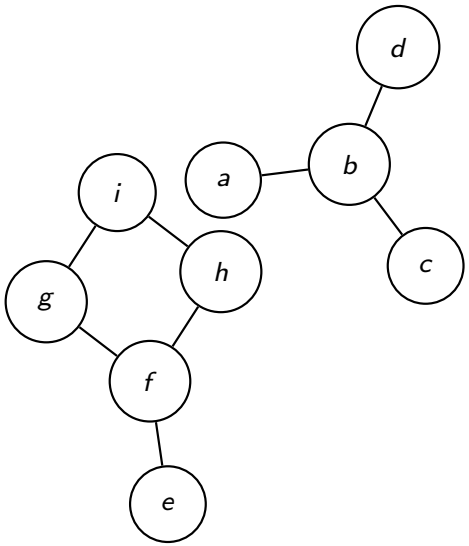



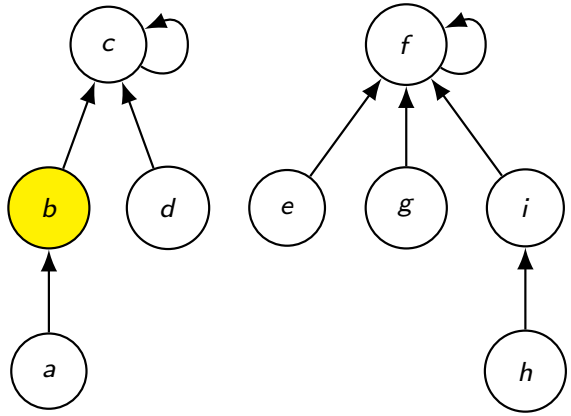
# Find connected component label for vertex a



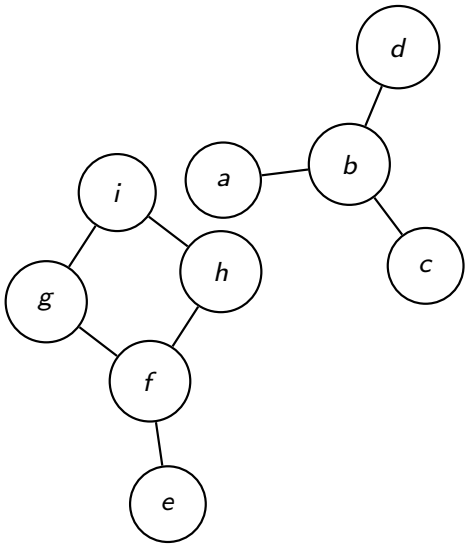



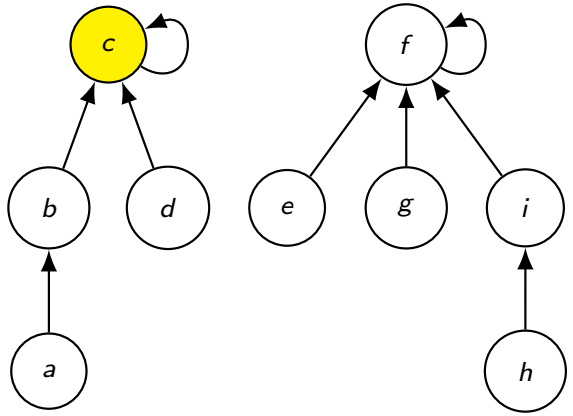
# Find connected component label for vertex a



# Find connected component label for vertex a



# Pseudo-code to find the root

## Solution 1 (using recursivity):

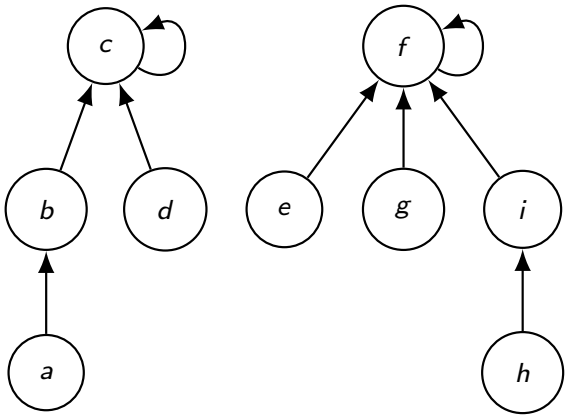
```
findTreeRoot(node){

 if( node->parent != node )
 then return( findTreeRoot( node->parent ) );
 else return( node                          );

}
```

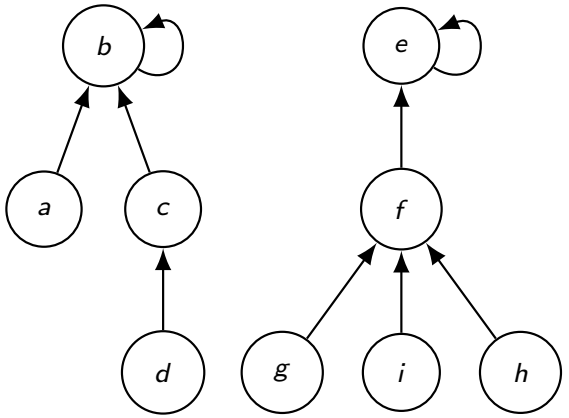
## Solution 2 (list):

```
findTreeRoot(node){

 Node Cursor = node;

 do while (Cursor != Cursor->parent) {
 Cursor=Cursor->parent;
 }

 return(Cursor);

}
```

# Non-uniqueness of trees

- Tree root = any connected component member
- Structure of tree doesn't matter

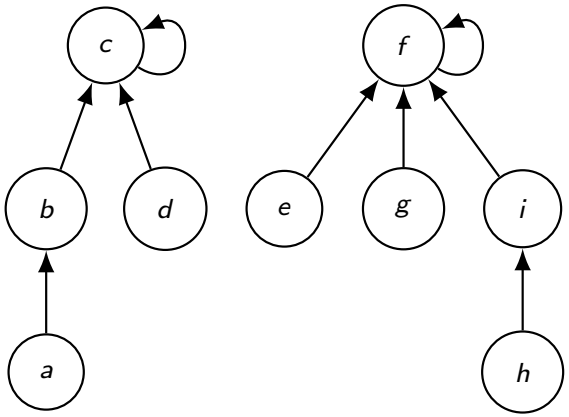


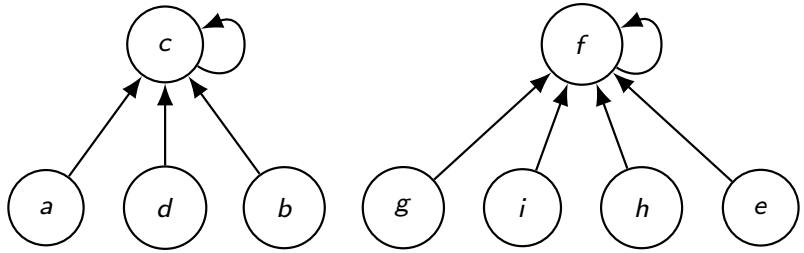
# Restructuring the forest

- Set of trees = forest
- Restructure the forest to improve performance of searches

Before:




After:



# Restructuring the forest

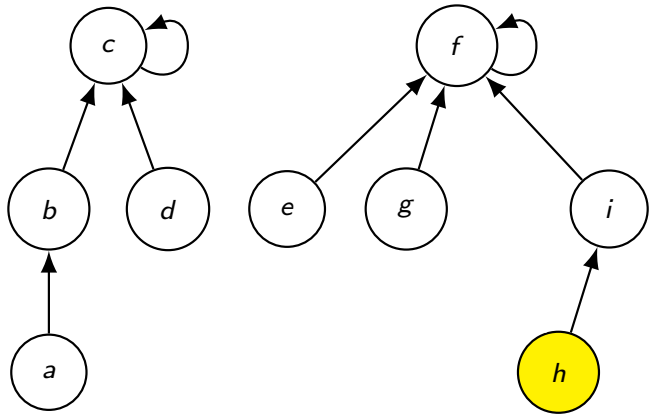
Taking advantage of recursivity to partly restructure the tree while unstacking recursive calls:

```
findTreeRoot(node){

 if( node->parent != node)
 then {

       node->parent=findTreeRoot( node->parent ); /* The "way-back" will be an opportunity to "partly" restructure the tree */
       return( findTreeRoot( node->parent ) );
      }
 else return( node                          );

}
```

To restructure the forest, find tree root of all nodes.

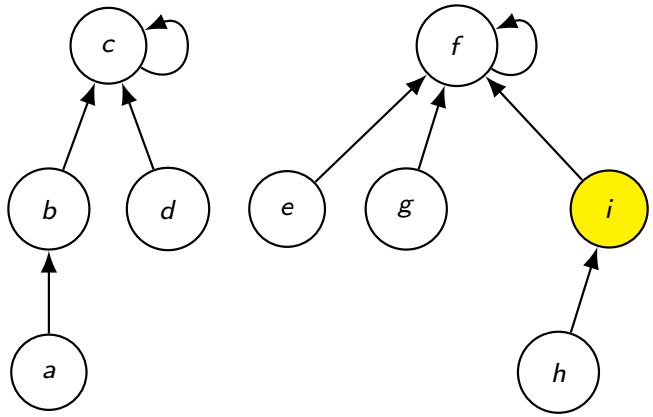
# findTreeRoot(h)

Way forward



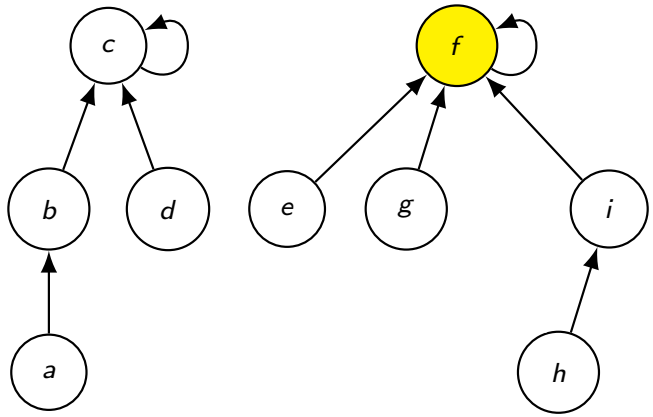
# findTreeRoot(h)

Way forward



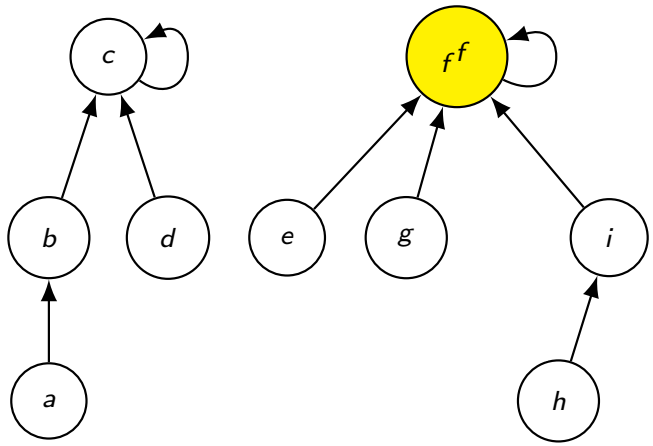
# findTreeRoot(h)

Way forward



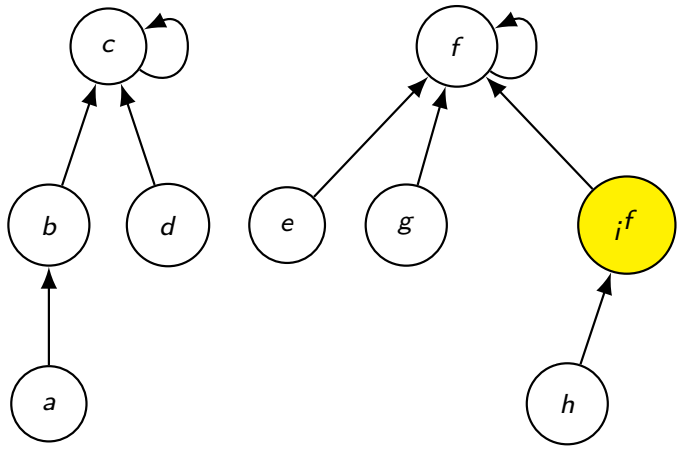
# findTreeRoot(h)

Way back



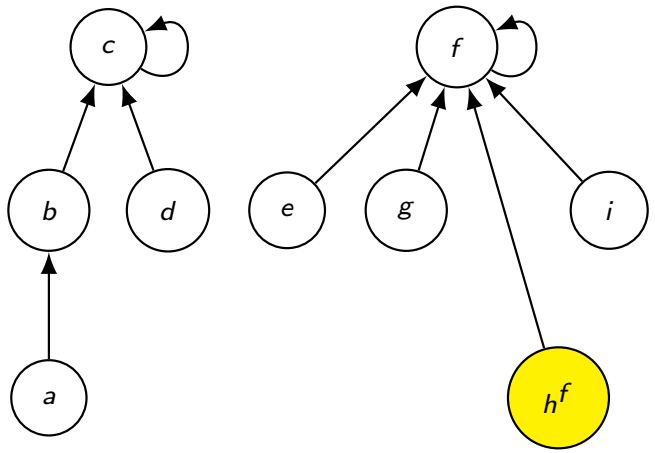
# findTreeRoot(h)

Way back



# findTreeRoot(h)

Way back



# Algorithm to build the trees

To build the trees we review every edge of $G$, one after the other, and apply the following processing.

Let:

- (E1,E2) one of the edges
- R1, the root of tree which E1 belongs to.
- R2, the root of tree which E2 belongs to.

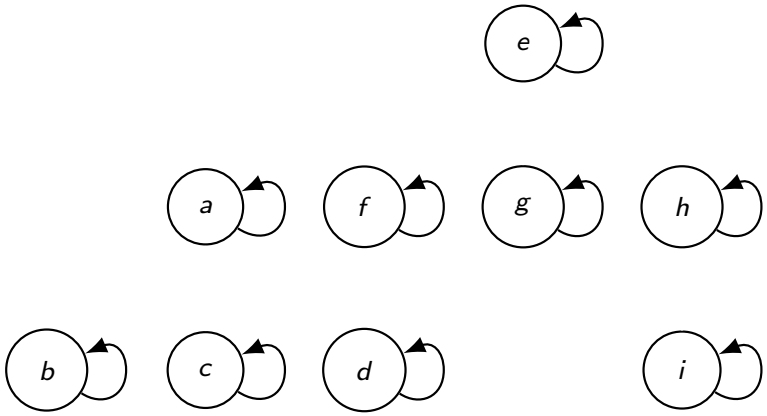
Processing:

- If R1=R2, then E1 and E2 are already part of the same tree. Do nothing
- If R1$\neq$ R2, then E1 and E2 are not part of the same tree: this situation must be corrected! To do it, we have R2 pointed to R1.

# Algorithm in action

Is_in_relation_with:

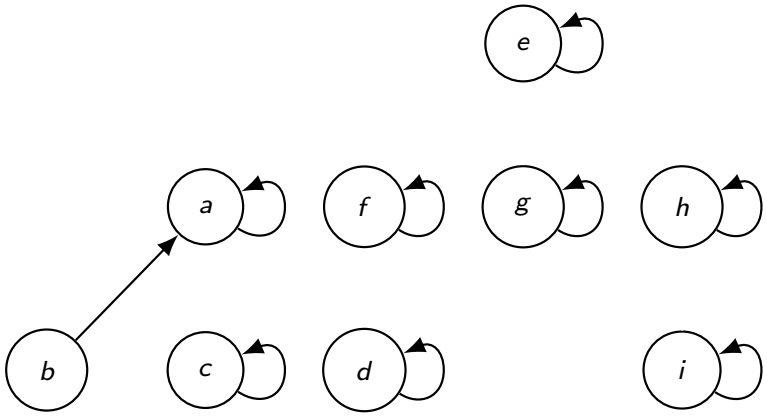
| Id_A | Id_B |
|---|---|
| a | b |
| b | c |
| b | d |
| e | f |
| f | g |
| h | i |
| g | i |
| f | h |



# Algorithm in action

Is_in_relation_with:

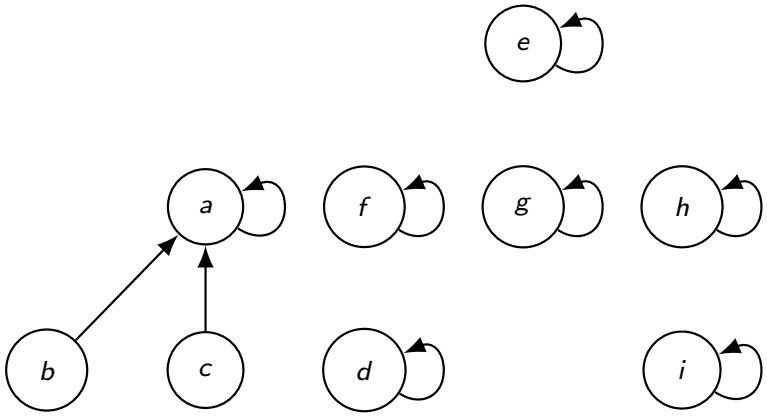
| Id_A | Id_B |
|---|---|
| a | b |
| b | c |
| b | d |
| e | f |
| f | g |
| h | i |
| g | i |
| f | h |



# Algorithm in action

Is_in_relation_with:

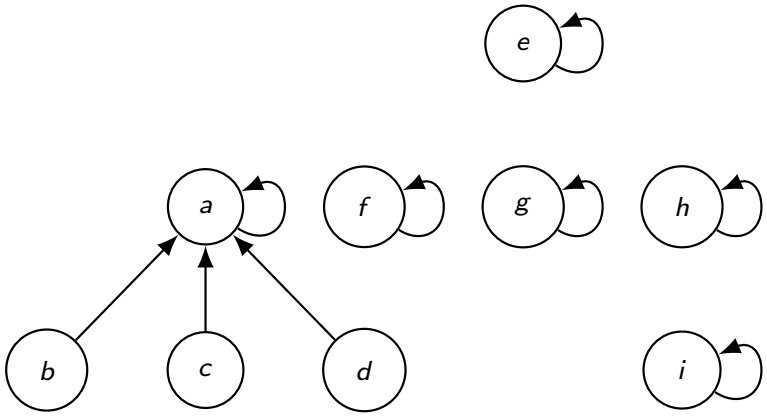
| Id_A | Id_B |
|---|---|
| a | b |
| b | c |
| b | d |
| e | f |
| f | g |
| h | i |
| g | i |
| f | h |



# Algorithm in action

Is_in_relation_with:

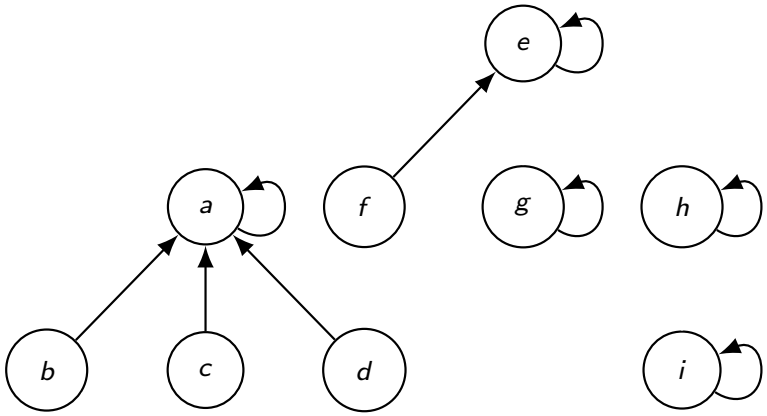
| Id_A | Id_B |
|---|---|
| a | b |
| b | c |
| b | d |
| e | f |
| f | g |
| h | i |
| g | i |
| f | h |



# Algorithm in action

Is_in_relation_with:

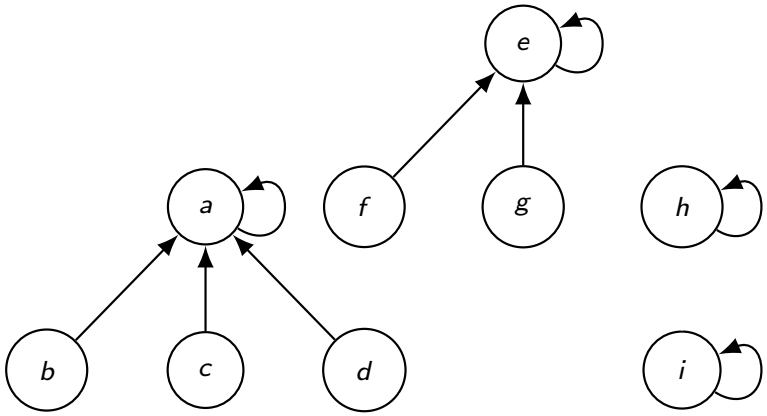
| Id_A | Id_B |
|---|---|
| a | b |
| b | c |
| b | d |
| e | f |
| f | g |
| h | i |
| g | i |
| f | h |



# Algorithm in action

Is_in_relation_with:

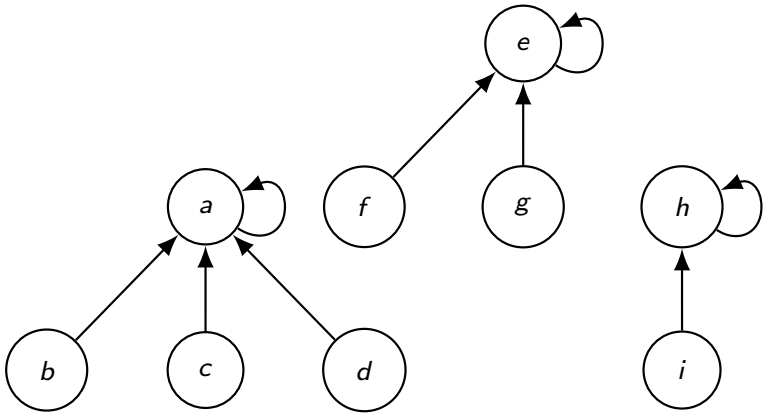
| Id_A | Id_B |
|---|---|
| a | b |
| b | c |
| b | d |
| e | f |
| f | g |
| h | i |
| g | i |
| f | h |



# Algorithm in action

Is_in_relation_with:

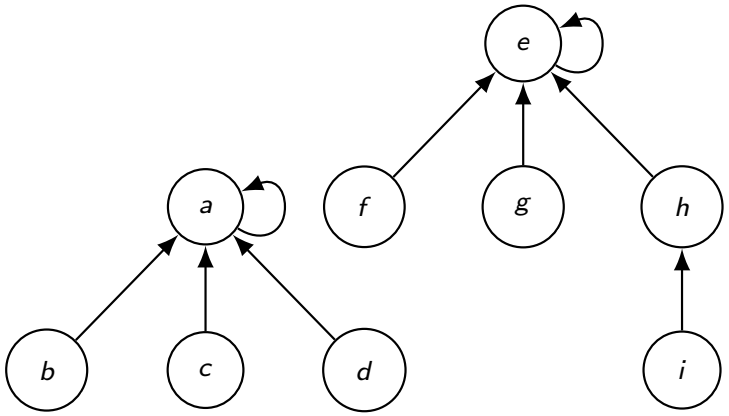
| Id_A | Id_B |
|---|---|
| a | b |
| b | c |
| b | d |
| e | f |
| f | g |
| h | i |
| g | i |
| f | h |



# Algorithm in action

Is_in_relation_with:

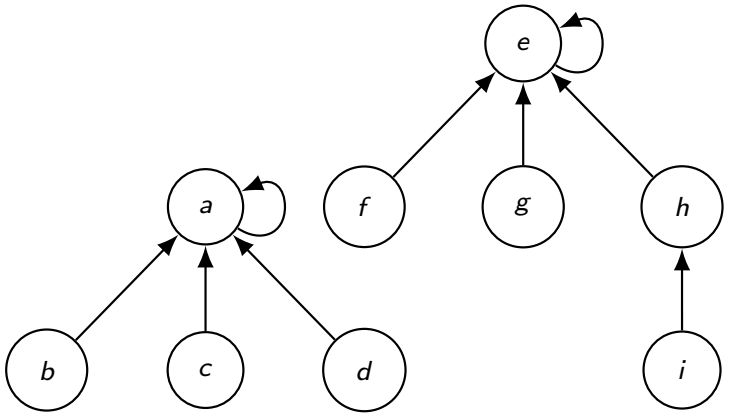
| Id_A | Id_B |
|---|---|
| a | b |
| b | c |
| b | d |
| e | f |
| f | g |
| h | i |
| g | i |
| f | h |



# Algorithm in action

Is_in_relation_with:

| Id_A | Id_B |
|---|---|
| a | b |
| b | c |
| b | d |
| e | f |
| f | g |
| h | i |
| g | i |
| f | h |



# In SAS language

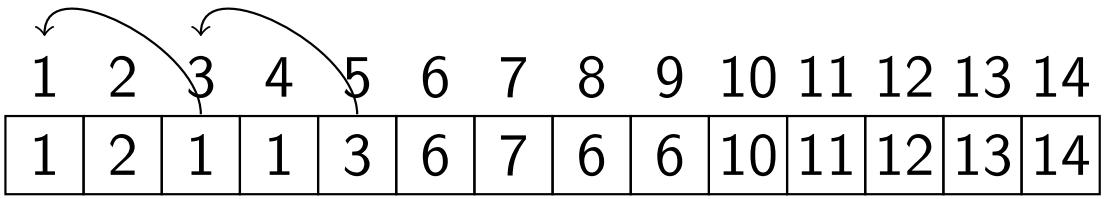
# Pointers

## A little problem ...

There is no pointers in SAS. What shall we do?

# Pointers

- Node Id's will be replaced by sequentiel numbers.
- An array will be used:
  - Each node will be identified by an array element
  - The array content = pointer



# Pointers

```
array pointer_of_node {&number_of_vertices } 8 _TEMPORARY_;

* INITIALISATION OF POINTERS;
* --------------------------;

if _N_=1 then do node=1 to &number_of_vertices;
                pointer_of_node{node}=node;
              end;
```

# Tree.FindRoot

```
root:
    F_root=0;
    element_current=id;
    element_next=pointer_of_node{id};
    longueur_path=1;
    do while(F_root=0);
       if element_next=element_current
       then do;
     F_root=1;
             root=element_current;
            end;
       else do;
             element_current=element_next;
     element_next=pointer_of_node{element_current};
            end;
       longueur_path=longueur_path+1;
    end;
    if longueur_path>2 then link restructure_path;
return;
```

# Graph.ProcessingEdges

```
* PROCESSING OF EDGES;
* -------------------;

set CG_003_EDGES end=fin nobs=nobs;

id=verticeSerial_of_edgeOrigin;      link root; root1=root;
id=verticeSerial_of_edgeDestination; link root; root2=root;
if root1 ne root2
then do;
      pointer_of_node{root2}=root1;
     end;
```

# Tree.RestructuringPathToRoot

```
* RESTRUCTURE_TREE FUNCTION;
* -------------------------;

restructure_tree:
do node=1 to &number_of_vertices;
  id=node; link root; pointer_of_node{node}=root;
end;
return;
```

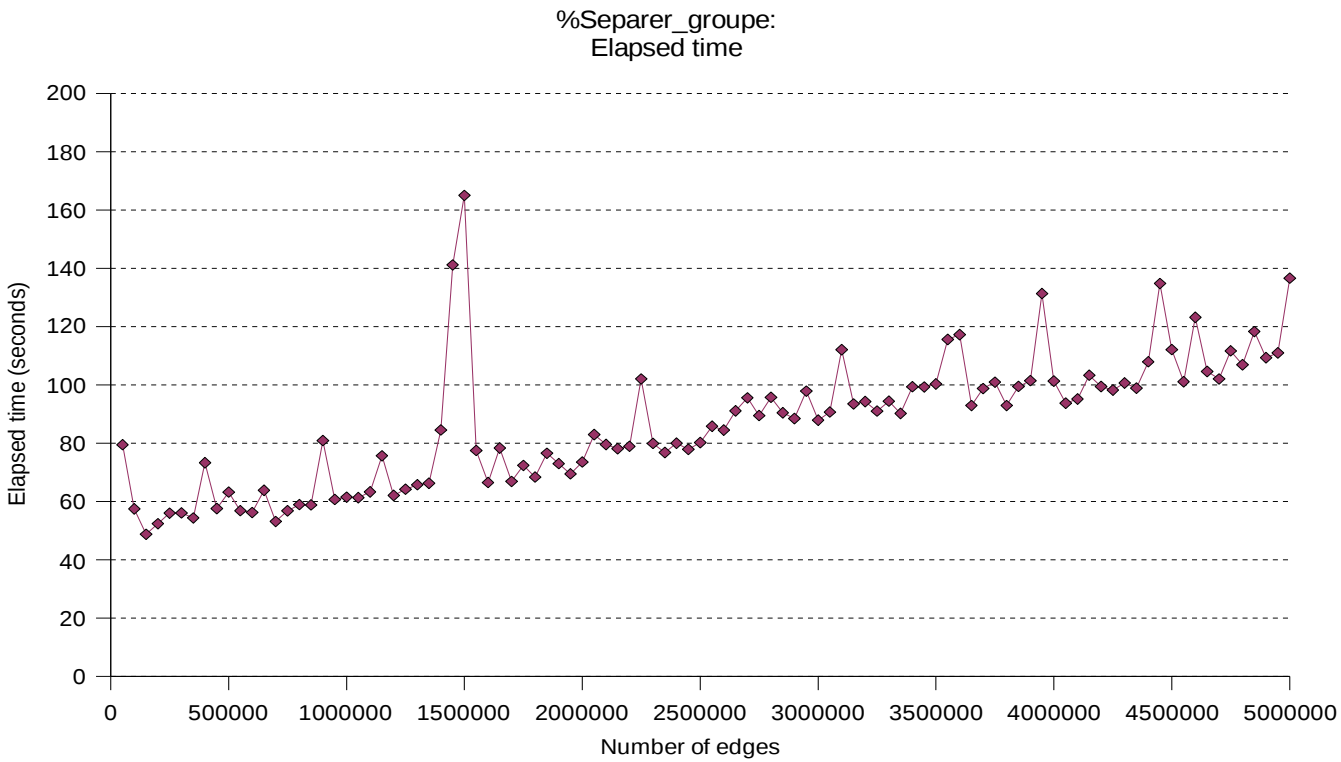
# Output generation

```
* EXPORT OF RESULTS;
* -----------------;

if fin then do;
              link restructure_tree;
              do node=1 to &number_of_vertices;
                root=pointer_of_node{node};
                output;
              end;
            end;

return;
```

# SAS 9.3 improvements

```
proc fcmp outlib=work.funcs.trial;
  subroutine findTreeRoot(depth,node,forest[*]);
        outargs depth, node, forest;
depth=1;
do while(node ne forest[node]);
         node=forest[node];
 depth=depth+1;
end;
  endsub;
  subroutine restructureTree(forest[*]);
        outargs forest;
do i=1 to dim(forest);
 node=i;
 call findTreeRoot(l,node,forest);
 forest[i]=node;
end;
  endsub;
run;
```

# Performance considerations


%Separer_groupe:
Elapsed time
Elapsed time (seconds)
200
180
160
140
120
100
80
60
40
20
0
0
500000
1000000
1500000
2000000
2500000
3000000
3500000
4000000
4500000
5000000
Number of edges

# Questions

Questions?