TP série N°1: test de connexité des réseaux «critiques»

Christoph Samuel – Jankowiak Matthias

I-Position du problème:

1- L'objectif de ce TP est celui du **test de connexité de réseaux "critiques"**. En effet, lors de la conception et lors du test d'un réseau de communication dans un domaine tel que la sécurité, il est primordial de s'assurer que la configuration utilisée garantit, pour certaines parties du réseau, une **communication** depuis **n'importe quel nœud vers n'importe quel autre.**

2- C'est l'ingénieur qui est chargé de la conception du réseau, il doit garantir une **communication entre deux nœuds,** ce qui revient à étudier la **connexité** de ce sous-réseau et donc à l'analyse de la connexité du modèle de graphe. Ainsi, on peut ramener ce problème à un problème de recherche de **composantes fortement connexes dans un graphe.** Ce problème est donc un problème classique de connexité dans un graphe orienté (les communications étant à sens unique).

	1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	1
2	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0
3	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1
5	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0
6	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0
7	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0
9	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
12	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
14	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
15	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0

3- Pour résoudre ce problème efficacement, l'ingénieur peu utiliser des outils tel que **l'algorithme de Kosaraj-Sharir,** qui permet d'apporter une solution de **complexité optimale** au problème de recherche, dans un graphe orienté en fonction du nombre de sommets et d'arcs du graphe. Dans le cas ou il trouve plusieurs composantes fortement connexes cela voudra dire qu'il peut y avoir une communication depuis n'importe quel nœud du réseau. Pour implémenter cet algorithme l'ingénieur peu se servir de la bibliothèque **Boost Graph Library,** disponible en C++ ainsi que l'environnement de développement VS code.

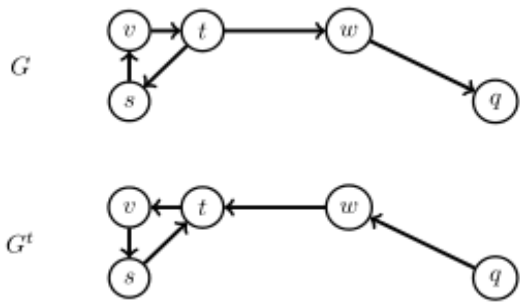
II-Réalisation:

1- Une façon de résoudre ce problème est de **modéliser le réseau** à l'aide d'un **graphe orienté G = (S,A).** Par rapport au problème réel chaque sommet s appartenant à S de G représente **un nœud du réseau** et chaque arc (si, sj) appartenant à A de G **modélise une possibilité de communication** (un arc) depuis un nœud si vers un nœud sj.

2- Le problème du test de connectivité des réseaux critiques est **réduit au problème** classique **de recherche de connectivité dans un graphe orienté.** Plus précisément, ce problème peut être formulé par la recherche dans un graphe orienté de composantes fortement connexes et la construction d'un graphe, où en fait chaque nœud du graphe représente un sous-réseau.

3- Étudier la connexité de certaines parties du réseau revient à **chercher les différentes composantes fortement connexes du graphe** représentant ce réseau. Pour cela, de nombreuses méthodes sont démontrées en appliquant des algorithmes sur la connexité des graphe pour résoudre ce problème. Par exemple l'algorithme de **Kosaraju-Sharir** nous est proposé pour apporter une solution à ce problème de recherche de composantes fortement connexes dans un graphe orienté, celui-ci se **déroule en deux étapes:**

- ***Étape 1:*** on lance un premier parcours en profondeur sur le graphe afin d'obtenir une liste de ses sommets triée dans l'ordre décroissant par rapport à leur date fin.
- ***Étape 2:*** un second parcours en profondeur, sur le graphe dual, avec dans chaque boucle de parcours, un marquage des sommets dans l'ordre établi précédemment.



Calculons la **complexité de cet algorithme,** soit n la taille du modèle de graphe et m le nombre de ses arcs. La complexité de l'algorithme se mesure en fonction des paramètres n et m.

Lors de l'étape 1, on parcourt entièrement le modèle de graphe, chaque sommet s est marqué et est entrée dans la liste une et une seule fois. Le temps total requis est en O(n + m).

Lors de l'étape 2, on réalise un parcourt en profondeur à partir de chaque arc. On en déduit que la complexité est en O (n + m). Donc la complexité de l'algorithme de Kosaraju-Sharir est en:

$$O(n + m + n + m) = O(2n + 2m) = O(n +m)$$

La complexité de l'algorithme de Kosaraju-Sharir étant don linéaire est très efficace pour la résolution de notre problème.

4- Nous commençons alors par construire le **graphe quotient** en utilisant la bibliothèque **Boost Graph Library** sur VisualStudio code de façon à obtenir sont graphe G. Pour cela nous commençons par inclure ainsi créé l'ensemble des librairies ainsi que des fonctions de parcourt de graphe nécessaires pour l'analyse d'un cas concret.

```cpp
// STL.
// pour std::cout.
#include <iostream>
// Inclusion de la librairie Boost.
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/depth_first_search.hpp>
#include <boost/graph/reverse_graph.hpp>
#include <boost/graph/graph_utility.hpp>
using namespace std;
using namespace boost;

```

```cpp
// Dans la structure, on définit les propriété des vertex, c'est à dire des noeux.
struct VertexProperties
{
    // Un identifiant manuel (qui n'est pas assigné automatiquement par la librairie) donc 1,2,3, etc ...
    unsigned id;
    // Un constructeur par défaut si aucune donnée n'est rentrée lors de la création du noeud.
    VertexProperties() : id(0) {}
    // La fonction qui permet d'assigner les valeur saisie par l'utilisateur au noeud correspondant.
    VertexProperties(unsigned i) : id(i) {}
};

```

```cpp
// Passe les paramètres nécéssaires à adjacency_list afin d'avoir le graph désiré
typedef adjacency_list <vecS, vecS, bidirectionalS, VertexProperties> Graph;
// OutEdgeList est un des types de conteneurs choisi en interne
// bidirectionalS graphe orienté avec des arcs bidirectionnels
// VertexProperties demande de prise en compte de nos noeuds personalisés et non les noeuds par defau
// Création du typedef Graph avec les paramètres ci-dessus.

// Création d'un DFS Visitor personnalisé pour noter et récupérer les sommets dans l'ordre date fin.
class custom_dfs_visitor_finish : public boost::default_dfs_visitor
{
    public:
        // On créee un constructeur qui va gébérer un vecteur dans lequel sera stocké le résultat de la recherche.
        custom_dfs_visitor_finish() : vv(new std::vector<int>()) {}
        // Ici on paramètre la recherche.
        template < typename Vertex, typename Graph >
        // Fonction de BGL, elle signifie que les noeuds sont ajoutés que lorsqu'ils sont des feuilles.
        void finish_vertex(Vertex u, const Graph & g) const
        {
            // On crée un tableau de noeuds.
            VertexProperties const& vertexProperties = g[u];
            // On affiche dans la console le deroulement de la recherche.
            std::cout << "Fin: " << vertexProperties.id <<endl;
            // On stocke l'id du noeud dans notre vecteur de int.
            vv->push_back(g[u].id);
        }
        // Fonction servant à restituer le vecteur.
        std::vector<int> &GetVectorF() const { return *vv; }

    private:
        // Déclaration du vecteur vv.
        boost::shared_ptr<std::vector<int> > vv;
};

```

```cpp
// Création d'un DFS Visitor personnalisé pour noter et restituer les sommets dans l'ordre date fin.
class custom_dfs_visitor_discover : public boost::default_dfs_visitor
{
    public:
        // On créee un constructeur qui va gébérer un vecteur dans lequel sera stocké le résultat de la recherche.
        custom_dfs_visitor_discover() : vw(new std::vector<int>()) {}
        // Ici on paramètre la recherche.
        template < typename Vertex, typename Graph >
        // Fonction de BGL, elle signifie que les noeuds sont ajoutés que lorsqu'ils sont des feuilles.
        void discover_vertex(Vertex u, const Graph & g) const
        {
            // On crée un tableau de noeuds.
            VertexProperties const& vertexProperties = g[u];
            // On affiche dans la console les noeuds découvert lors de la recherche.
            std::cout << "Découvert: " << vertexProperties.id << "," <<endl;
            // On stocke l'id du noeud dans notre vecteur de int.
            vw->push_back(g[u].id);
        }
        // Fonction servant à restituer le vecteur.
        std::vector<int> &GetVectorD() const { return *vw; }

    private:
        // Déclaration du vecteur vw.
        boost::shared_ptr<std::vector<int> > vw;
};

```

Ensuite on se base sur la cas d'application présenté du tp1, on créé donc un graphe correspondant au **test de connexité de réseaux "critiques"** en construisant notre main.

```
84  // Fonction d'application.
85  int main(int, char*[])
86  {
87      // Définiton des types noeuds et arcs
88      typedef graph_traits<Graph>::vertex_descriptor vertex_t;
89      typedef graph_traits<Graph>::edge_descriptor edge_t;
90
91      //Création du graphe.
92      Graph g;
```

On déclare ensuite chaque nœuds et arcs pour notre cas voici un exemple pour les deux premiers nœuds et arcs.

```
93      // Déclaration de tous les noeuds de nom : "1", "2", "3" ... dans le Graphe g.
94      vertex_t V1 = add_vertex(VertexProperties(1),g);
95      vertex_t V2 = add_vertex(VertexProperties(2),g);
```

```
110     // Création d'un tableau Contenant le nom et la date de fin de chaque nœud : il est nécessaire à la créaation de l'image.
111     const string names [] = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13", "14", "15"};
112
```

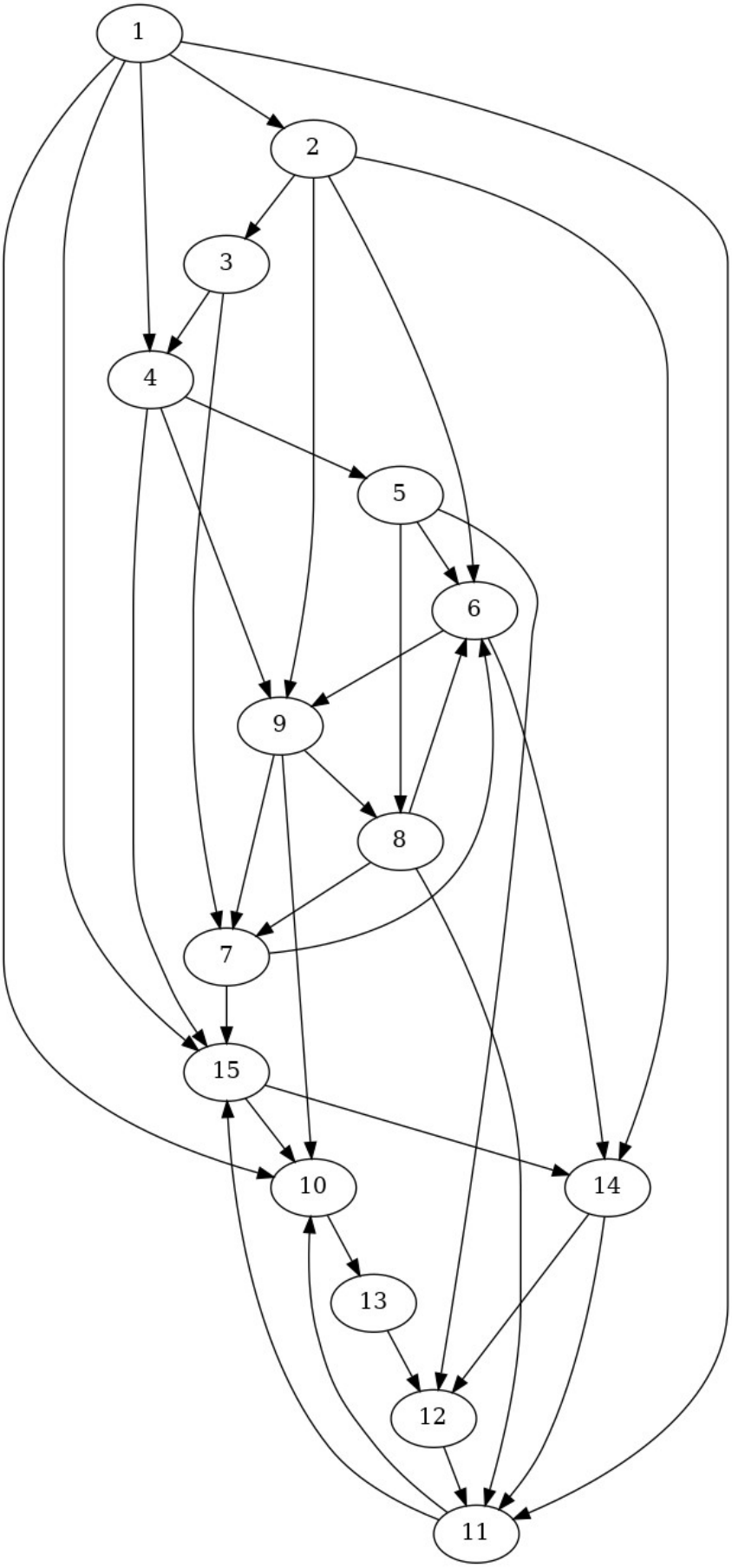
```
113     // Déclaration de tous les arcs (ex: nom : e1 qui est dirigé de A vers B dans le Graphe g).
114     pair <Graph::edge_descriptor, bool> e1 = add_edge(V1, V2, g);
115     pair <Graph::edge_descriptor, bool> e2 = add_edge(V1, V4, g);
```

Puis on passe a l'initialisation et a la création des fichier nécessaires pour nous afficher les différent graphes (graphe normal, graphe dual…).

```
151     // Première recherche en profondeur, sur le graphe.
152     custom_dfs_visitor_finish vis;
153     cout << "Recherche en profondeur sur le Graphe:" << endl;
154     depth_first_search(g , visitor(vis));
155
156     // Création des fichiers et streams nécessaires à la création des images.
157     // Création du graphe normal.
158     string filename = "graphe.dot";
159     ofstream fout(filename.c_str());
160     // Création du résultat.
161     string filename2 = "dual.dot";
162     ofstream fout2(filename2.c_str());
163     // Création du graphe dual.
164     string filename3 = "result.dot";
165     ofstream fout3(filename3.c_str());
166
167     // Création des images du graphe normal et du graphe dual.
168     write_graphviz(fout, g, make_label_writer(&names[0]));
169     system("dot -Tpng graphe.dot > graphe.png");
170     write_graphviz(fout2, make_reverse_graph(g), make_label_writer(&names[0]));
171     system("dot -Tpng dual.dot > dual.png");
```

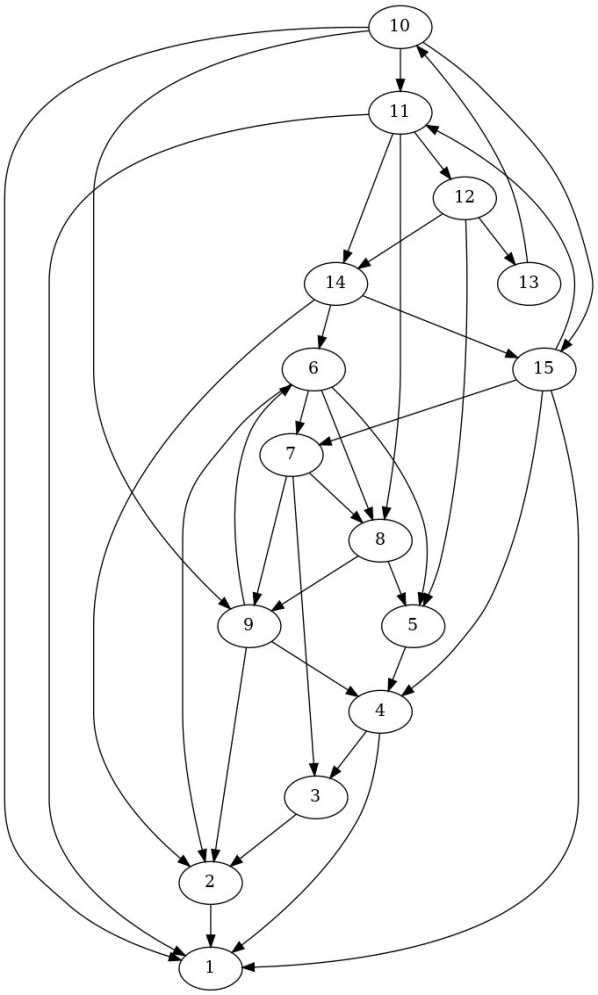
```
173     // Récupération de l'ordre date fin de la première recherche, et inversion de l'ordre pour l'avoir décroissant.
174     vector<int> datefinDecroissant = vis.GetVectorF();
175     reverse(datefinDecroissant.begin(), datefinDecroissant.end());
176
177     // Création d'un vecteur aidant à isoler les composantes fortement connexes.
178     // En ne contenant que les sommets pas encore explorés.
179     vector<int> vect;
180     for (int i = 0; i < 15; i++) {vect.push_back(i+1);}
181     custom_dfs_visitor_discover tempVis;
182
183     // On récupére dans indexmap l'index des noeuds de g.
184     auto indexmap = get(vertex_index, g);
185     // Dans colormap on crée des couleurs pour les noeuds à partir de leur index
186     auto colormap = make_vector_property_map<default_color_type>(indexmap);
187     vector<int> tempVect;
188     vector<int> tempVect2;
189     vector<int> tempDiff;
190
```

Pour réaliser l'algorithme de Kosaraju-Sharir dans notre étude de cas nous commençons par mettre en place un parcours en profondeur du graphe G en fonction de **la date début dans l'ordre croissant** et nous obtenons L1 : (1,2,3,4,5,6,9,8,7,15,14,10,13,12,11), de plus on implémente la bibliothèque GraphBoost. L'algorithme va ensuite trier par ordre croissant ces dates de fin. Nous obtenons alors le résultat du **parcours en profondeur sur le graphe G** suivant:


1
2
3
4
5
6
9
8
7
15
10
14
13
12
11

```
191     // Parcours en profondeur sur le graphe dual, en suivant l'ordre date fin décroissant.
192     cout <<endl<<endl<< "Recherche en profondeur sur le graphe dual, dans l'ordre décroissant de date fin:" <<endl;
193     for (int i = 0; i < datefinDecroissant.size(); i++)
194     {
195         // On ne lance une recherche que sur les sommets que l'on n'a pas encore visité.
196         vector<int>::iterator it = std::find(vect.begin(), vect.end(), datefinDecroissant[i]);
197         if (it != vect.end())
198         {
199             cout <<endl<< "Recherche à partir de: " << datefinDecroissant[i] << endl;
200             depth_first_visit(make_reverse_graph(g), i, tempVis, colormap);
201             tempVect = tempVis.GetVectorD();
202             set_difference(tempVect.begin(), tempVect.end(), tempVect2.begin(), tempVect2.end(), inserter(tempDiff, tempDiff.begin()));
203             cout << "Composante fortement connexe trouvée: ";
204             for (int p = 0; p < tempDiff.size(); p++)
205               cout << tempDiff[p] << " ";
206             cout <<endl;
207 
208             // Mise à jour du vecteur d'aide.
209             for (int j = 0; j < tempDiff.size(); j++)
210             {
211                 vector<int>::iterator it2 = std::find(vect.begin(), vect.end(), tempDiff[j]);
212                 if (it2 != vect.end()) {vect.erase(it2);}
213             }
214                 // Pour la représentation graphique, suppression des arcs n'appartenant pas aux composantes fortement connexes.
215                 if (tempDiff.size() == 1)
216                 {
217                     clear_out_edges(i,g);
218                     clear_in_edges(i,g);
219                 }
220                 else for (int j = 0; j < tempDiff.size(); j++)
221                 {
222                     for (int k = 0; k < vect.size(); k++)
223                     {
224                         remove_edge(tempDiff[j]-1, vect[k]-1, g);
225                         remove_edge(vect[k]-1, tempDiff[j]-1, g);
226                     }
227                 }
228                 tempVect2 = tempVect;
229                 tempDiff.clear();
230         }
231     }
```

Le graphe du réseau se présente donc de la manière suivante sur lequel chaque sommet est identifié par **son identifiant** (au dessus) et par **sa date de fin** (au dessous). Puis dans **l'ordre croissant de date fin** et nous obtenons L2 : (11,12,13,10,14,15,7,8,9,6,5,4,3,2,1). Pour finir le fonctionnement de l'algorithme, on **cherche le graphe dual Gt:**

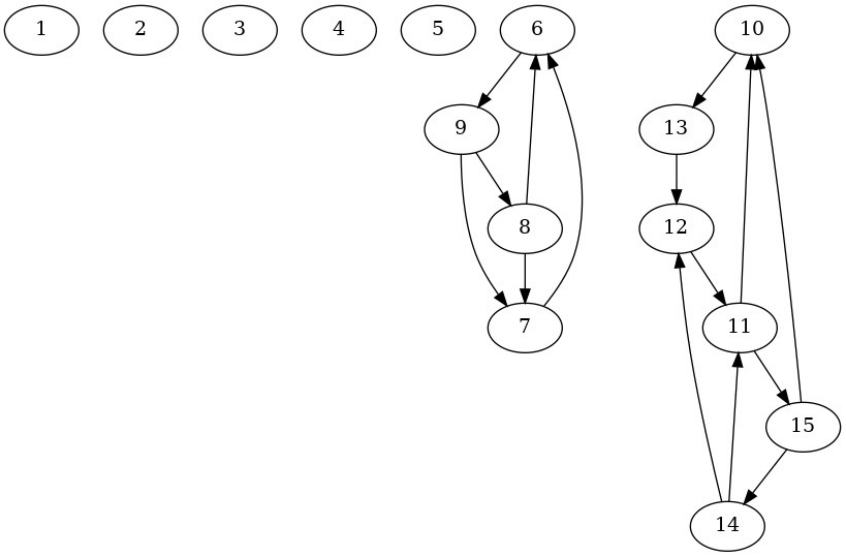


On lance alors un **second parcours en profondeur sur le graphe dual,** en marquant les sommets et dans l'ordre spécifié par la liste L1 et donc en partant du sommet 1 ici. Le résultat obtenu est **une forêt de 7 arborescences de racines** 1, 2, 3, 4, 5, 6 et 15

→ {1}, {2}, {3}, {4}, {5}, {6,7,8,9}, {15,10,11,12,13,14}

```
232     // Création de l'image finale, contenant les composantes fortement connexes.
233     write_graphviz(fout3, g, make_label_writer(&names[0]));
234     system("dot -Tpng result.dot > result.png");
235     cout << endl;
236     return 0;
```

Ici on affiche le résultat après le passage dans l'algorithme de Kosaraj-Sharir du graphe dual



Pour s'assurer que le résultat obtenu est bien une forêt de 7 arborescences de racines, on compile et exécute notre programme (tp1.cpp). Nous observons dans le **terminal les résultats ci dessous.**

```
schristoph@scinfe061 ~
cd Bureau/graphes/tp1/
schristoph@scinfe061 ~/Bureau/graphes/tp1
g++ -o tp1.o tp1.cpp
schristoph@scinfe061 ~/Bureau/graphes/tp1
./tp1.o
Recherche en profondeur sur le Graphe:
Fin: 11
Fin: 12
Fin: 13
Fin: 10
Fin: 14
Fin: 15
Fin: 7
Fin: 8
Fin: 9
Fin: 6
Fin: 5
Fin: 4
Fin: 3
Fin: 2
Fin: 1
```

Premier résultat de notre terminal lors de la première recherche en profondeur et fin des différents nœuds du graphe.

```
Recherche en profondeur sur le graphe dual, dans l'ordre décroissant de date fin:

Recherche à partir de: 1
Découvert: 1,
Composante fortement connexe trouvée: 1

Recherche à partir de: 2
Découvert: 2,
Composante fortement connexe trouvée: 2

Recherche à partir de: 3
Découvert: 3,
Composante fortement connexe trouvée: 3

Recherche à partir de: 4
Découvert: 4,
Composante fortement connexe trouvée: 4

Recherche à partir de: 5
Découvert: 5,
Composante fortement connexe trouvée: 5
```

Résultat lors de la seconde recherche en profondeur et découverte des composantes fortement connexe 1, 2, 3, 4 et 5.

```
Recherche à partir de: 6
Découvert: 6,
Découvert: 7,
Découvert: 8,
Découvert: 9,
Composante fortement connexe trouvée: 6 7 8 9
```

Composantes fortement connexe 6, composée des nœuds 6, 7, 8 et 9.

```
Recherche à partir de: 15
Découvert: 10,
Découvert: 11,
Découvert: 12,
Découvert: 13,
Découvert: 14,
Découvert: 15,
Composante fortement connexe trouvée: 10 11 12 13 14 15
```

Composantes fortement connexe 15, composée des nœuds 10, 11, 12, 13, 14 et 15,

On obtient alors les **sommets 1, 2, 3, 4, 5, 6 et 15** qui sont déclarés comme des composantes fortement connexes dans le graphe g. Après observation du **graphe quotient,** nous pouvons assurer que la configuration utilisé garantit, pour certaines parties du réseau, une communication depuis **n'importe quel nœud** du réseau **vers n'importe quel autre.**

III-Bilan/Conclusion:

1- Sur le plan de l'application des modèles et algorithmes sur les graphes, ce TP nous démontre qu'il est aisé de **ramener de nombreux problèmes réels à un problème de la théorie des graphes,** ici celui de la connexité dans un graphe et de la recherche de composantes fortement connexes, et la puissance de ses solutions algorithmiques pour résoudre les-dits problèmes. Nous avons appris à appliquer nos connaissances sur **la connexité d'un graphe pour réaliser un graphe** via un algorithme et les différentes manières de stocker et manipuler des graphes.

2- Sur le plan de résolution des problèmes réels, cet exemple nous démontre **l'importance des test sur la connexité des réseaux réels** de machines interconnectées au seins des différentes entreprises et nous donne un aperçu d'un futur travail qui nous attend.