Projet réalisé en binôme : Julien Deveaux et Justine Lemesle.

Le routage

I - Introduction.

Le routage permet de gérer où les paquets sont envoyés à chaque commutateur pour arriver à la fois le plus rapidement et le plus efficacement possible. Ainsi, à chaque commutateur est associé une table de routage qui répertorie pour aller à un commutateur donné, vers lequel de ses voisins est le plus efficace pour y aller jusqu'au moins efficace.

1) La création du réseau graphique

A la base, ne connaissant pas du tout la structure et le fonctionnement de la bibliothèque GraphStream, nous somme partis du principe que nous allions créer nous-mêmes nos noeuds et nous avons ainsi créé une classe noeud qui avait un tableau répertoriant ses voisins, un attribut pour son nom et un tableau complémentaire à celui de ses voisins qui sauvegarderait le poids du chemin pour aller à son voisin. Après cela, nous avons commencé à regarder plus en détail les exemples d'utilisation de GraphStream et avons vu que notre classe noeud était inutile et elle a ainsi été supprimée. Nous avons recopié l'exemple dans le documentation de GraphStream et nous avons commencé à le modifier et à regarder comment on interagissait avec la bibliothèque et surtout comment ajouter des attributs à un noeud où à un edge (liaison entre noeud) pour pouvoir sauvegarder un nom où un poids. Nous avons en parallèle à cela rajoutés une IHM pour pouvoir contrôler plus fluidement les ajouts de nœuds / liaison et les suppression des ces derniers. Par la suite est venu se rajouter un bouton pour modifier le poids d'une liaison et pour avoir accès au panneau de contrôle pour générer une table de routage et le chemin le plus court entre deux nœuds.

2) L'algorithme de Dijkstra

Après des recherches dans la documentation de GraphStream, nous avons remarqué que cette bibliothèque incorporait des fonctions pour calculer le chemin le plus court entre deux nœuds. Nous avons choisi d'utiliser l'algorithme de Dijkstra parce qu'il semblait être le plus efficace. Ainsi pour calculer le chemin le plus court entre un nœud de départ et un nœud de destination, il nous renvoie directement un chemin qui peut être facilement transformé en texte contenant le chemin avec les nœuds intermédiaires pour atteindre le nœud destination. La capture d'écran ci-dessous montre comment l'algorithme est utilisé de façon très facile pour avoir le chemin le plus court entre deux noeuds, l'argument à la fin de l'initialisation de la

variable dijkstra “p” est notre variable de lien où est stocké le poids du lien et est pris en compte par l’algorithme pour obtenir le chemin le plus court entre les noeuds.

```
Dijkstra dijkstra = new Dijkstra(Dijkstra.Element.EDGE, resultAttribute: null, lengthAttribute: "p");
dijkstra.init(g);
dijkstra.setSource(g.getNode(comm1.getSelectedIndex()));
dijkstra.compute();
String ch = String.valueOf(dijkstra.getPath(g.getNode(comm2.getSelectedIndex())));
if(ch == "[]") {
    res.setText("Aucune connexion entre les commutateurs");
} else {
    res.setText(ch);
}
dijkstra.clear();
```

Pour le calcul de la table de routage, nous utilisons le même algorithme mais c’est ce qui l'entoure qui diffère. En effet, nous établissons la table de routage du commutateur 1 par exemple vers le commutateur 2, puis 3, puis 4, … et en mettant le résultat de la fonction routage dans un tableau. Dans cette fonction routage, nous récupérons d’abord les voisins du noeud que nous voulons récupérer la table de routage :

```
for(Edge e : g.getNode(noeudDepart)) {
    voisins.add(e.getOpposite(g.getNode(noeudDepart)));
}
```

Ensuite, nous enregistrons le poids des liens entre le nœud de départ et les voisins dans un tableau puis nous mettons ces poids à 500000 pour que l’algorithme de dijkstra ne repasse pas par le nœud de départ, ce nombre étant improbablement haut :

```
int[] pStore = new int[voisins.size()];
for(int i = 0; i < voisins.size(); i++) {
    try {
        pStore[i] = (int) g.getEdge(S: g.getNode(noeudDepart).getAttribute(S: "ui.label") + "-" + voisins.get(i).getAttribute(S: "ui.label")).getAttribute(S: "p");
        g.getEdge(S: g.getNode(noeudDepart).getAttribute(S: "ui.label") + "-" + voisins.get(i).getAttribute(S: "ui.label")).setAttribute(S: "p", ...objects: 500000);
    } catch (NullPointerException e) {
        pStore[i] = (int) g.getEdge(S: voisins.get(i).getAttribute(S: "ui.label") + "-" + g.getNode(noeudDepart).getAttribute(S: "ui.label")).getAttribute(S: "p");
        g.getEdge(S: voisins.get(i).getAttribute(S: "ui.label") + "-" + g.getNode(noeudDepart).getAttribute(S: "ui.label")).setAttribute(S: "p", ...objects: 500000);
    }
}
```

Après avoir enregistré et changé les valeurs des poids des liens reliant le noeud de départ, nous mettons le noeud de départ pour l’algorithme de dijkstra le noeud d’arrivée (où l’on veut arriver à partir du noeud de départ) puis nous extrayons le chemin le plus court pour arriver à ce noeud pour tous les voisins du noeud de départ. Simultanément, nous calculons le poids de ce chemin et nous y ajoutons la valeur du lien qui relie le voisin duquel on a calculé le chemin le plus rapide au noeud de départ et nous sauvegardons le résultat ainsi que le noeud voisin associé à ce résultat dans une hashmap :

```
HashMap <Integer, String> map = new HashMap<>();

for(int i = 0; i < voisins.size(); i++) {
    dijkstra.setSource(noeudArrive);
    dijkstra.compute(); //chemin entre l'arrivée et le voisin i
    String chemin = String.valueOf(dijkstra.getPath(voisins.get(i)));
    chemin = chemin.substring(1, chemin.length() - 1);
    String[] noeuds = chemin.split( regex: ", ");
    int poids = 0;
    for(int j = 0; j < noeuds.length - 1; j++) {
        try {
            poids = poids + (int) g.getEdge( s: noeuds[j] + "-" + noeuds[j + 1]).getAttribute( s: "p");
        } catch (NullPointerException e) {
            poids = poids + (int) g.getEdge( s: noeuds[j + 1] + "-" + noeuds[j]).getAttribute( s: "p");
        }
    }
    poids += pStore[i];
    if(map.get(poids) != null) {
        map.put(poids, "( " + map.get(poids) + " / " + voisins.get(i) + " )");
    } else {
        map.put(poids, String.valueOf(voisins.get(i)));
    }
}
```

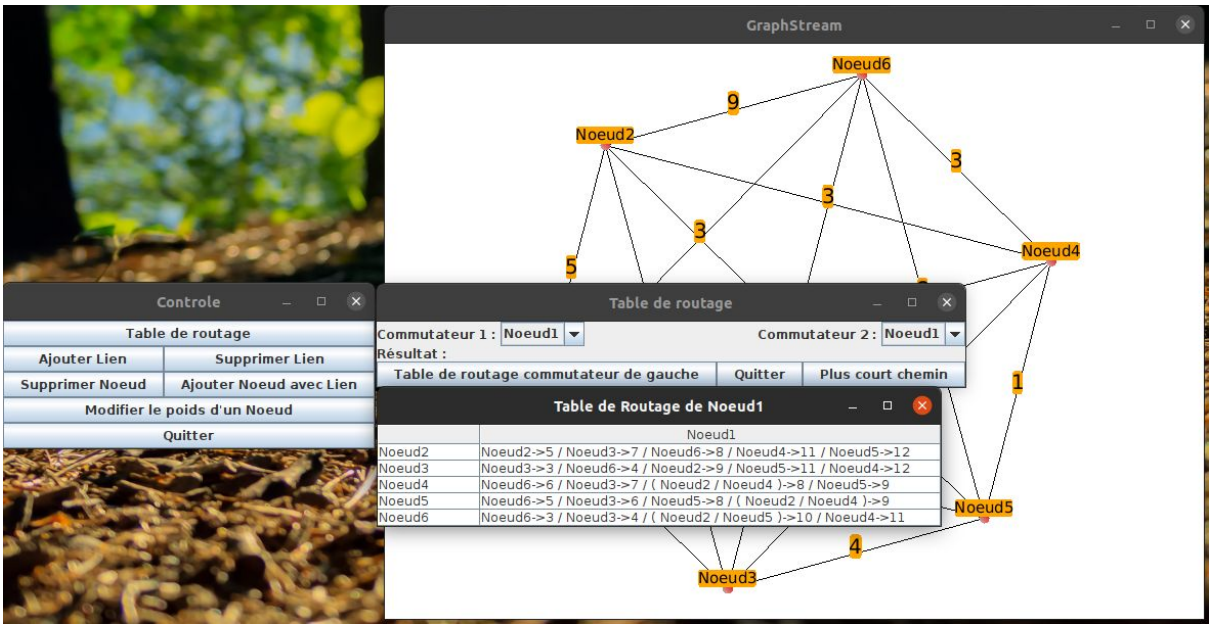
Ensuite, nous trions les valeurs des poids des chemins les plus courts et construisons le string qui sera inséré dans le tableau de la table de routage et qui sera trié du chemin le plus court au plus long :

```
List<Integer> sorted = new ArrayList<>(map.keySet());
Collections.sort(sorted);
for(int i = 0; i < sorted.size(); i++) {
    if(i < sorted.size() - 1 && sorted.get(i) == sorted.get(i + 1)) {
        res = res + "( " + map.get(sorted.get(i)) + "->" + sorted.get(i) + " / " + map.get(sorted.get(i + 1)) + " " + sorted.get(i + 1) + " ) / ";
    } else {
        res = res + map.get(sorted.get(i)) + "->" + sorted.get(i) + " / ";
    }
}
res = res.substring(0, res.length() - 2);
```

Ensuite, nous restaurons les valeurs des liens qui relient notre noeud de départ à ses voisins :

```
for(int i = 0; i < voisins.size(); i++) {
    try {
        g.getEdge( s: g.getNode(noeudDepart).getAttribute( s: "ui.label") + "-" + voisins.get(i).getAttribute( s: "ui.label")).setAttribute( s: "p", pStore[i]);
    } catch (NullPointerException e) {
        g.getEdge( s: voisins.get(i).getAttribute( s: "ui.label") + "-" + g.getNode(noeudDepart).getAttribute( s: "ui.label")).setAttribute( s: "p", pStore[i]);
    }
}
```

Ainsi, nous obtenons le résultat suivant :



III - Conclusion

Pour conclure, ce TP fut très intéressant à coder, étant donné que nous n'utilisons que très rarement voir jamais de librairies externes pour nos projets, ce qui est une expérience très enrichissante. De plus étant très graphique, c'est très gratifiant de voir directement l'effet de ce que nous codons sur l'écran.
Les perspectives d'améliorations que nous voyons sont que nous pourrions faire en sorte de réduire le nombre de fenêtres qui s'ouvrent, trouver comment intégrer un graph dans une JFrame et implémenter les lignes de tables de circuits virtuels entre deux machines.

Justine Lemesle & Julien Deveaux.