Routage et mensonge

Réalisé par :
Ousmane Bah
Sara Real Santos

Dans le cadre du cours :
Algorithmique de la mobilité 2021-22.

Travail présenté à
Chargés de TD : Arnaud Casteigts
Nicolas Hanusse
Colette Johnen
Cyril Gavoille

Du département de Master Informatique
Université de Bordeaux

Lien vers le projet : https://github.com/sarars1/projetAlgomob.

Table des matières

1 Introduction

Les protocoles et algorithmes de routage sont un objet d'étude et une grande problématique dans le domaine des réseaux. Pour mieux comprendre comment ces algorithmes peuvent fonctionner, nous avons décidé de déployer un algorithme de routage dans un graphe.
Pour ce faire nous considérons que les suivantes conditions sont satisfaites :

- Nous enlevons et ajoutons un lien à la fois
- Il y a un noeud appelé la destination, à partir duquel on lancera la construction
- Chaque noeud connaît la taille du graphe

Nous avons donc un graphe, où il y a un noeud que nous considérons la **destination**. L'objectif de cet algorithme de routage est de calculer la meilleure route à emprunter vers une destination pour chaque noeud de notre graphe. Dans le cas où il y a des liens qui sont ajoutés ou supprimés, les noeuds affectés sont considérés comme des possibles menteurs : il se peut que le chemin qu'ils prenaient n'est plus le plus court. Si c'est le cas, nous considérons ces noeuds des **menteurs**. Il faudra détecter ceci et recalculer le chemin le plus court. Si lors de modifications il y a des noeuds qui ne sont plus connectés à la racine, nous considérons ces noeuds comme déconnectés.

2 Contexte

Nous nous trouvons dans un graphe qui peut varier dans le temps. Nous avons un noeud appelé destination. Nous allons construire un sous-graphe tel que chaque noeud de ce sous-graphe ait comme parent son voisin le plus proche de la destination. L'idée est de construire des 'chemins' vers la destination. La destination sera mise en vert et augmentée de taille. Nous visualisons les relations parents-enfants entre les noeuds avec des liens en gras et des flèches.
La construction de ce sous-graphe sera lancée au moment de la sélection de la racine. Si jamais des liens sont ajoutés ou éliminés, le chemin devra être actualisé pour emprunter le chemin le plus court, si ça a changé. Si un noeud n'est plus connecté à un chemin allant vers la racine, il est considéré comme déconnecté de celle-ci. Les noeuds suspectés d'êtres menteurs seront mis en orange, les noeuds qui seront confirmés menteurs seront mis en rouge, et les noeuds déconnectés en rose.

3 Approche

L'approche présentée dans notre projet est une approche séquentielle. À chaque ronde, les noeuds (qui ne sont pas la destination) vont faire deux actions principalement.

- En premier, chaque noeud va parcourir la liste des messages qu'il a reçu et va les filtrer. De ces messages il va retenir la distance la plus petite envoyé par un de ses voisins et le voisin qui est à cette distance.
- En deuxième, il va vérifier qu'il est toujours connecté à la racine. Si c'est pas le cas il est déconnecté, sinon on prend la distance la plus petite et le noeud vérifie si c'est toujours son père qui a la distance la plus petite. Si c'est plus son père qui a la distance la plus petite, ça veut dire que le noeud doit changer de père et donc qu'il était en fait un menteur.

Finalement, le noeud envoie un message avec sa distance à tous ses voisins.
Dans le cas du noeud destination, il enverra juste à chaque fois un message avec son hop.
Cette approche nous permet de gérer la construction ainsi que l'élimination ou ajout de liens entre des différents noeuds et la déconnexion d'un ou plusieurs noeuds de la destination dans un seul algorithme.

4 Algorithme

Nous allons centrer nos explications autour de la méthode **onClock()** et **onSelection()**. Nous utilisons aussi d'autres méthodes comme **onMessage()**, **onLinkRemoved()** et **onLinkAdded()** qui sont importantes aussi mais sont plutôt simples et envoient que des messages pour changer des parents, de couleur ou changer l'aspect des liens (gras).

4.1 Arguments des noeuds

Dans chaque noeud nous avons des arguments :
— **parent** - le père du noeud, initialisé à null
— **children** - une liste des enfants du noeud
— **racine** - un boolean qui indique si le noeud est la racine, initialisé à false
— **hop** - la distance du noeud à la racine, initialisé à 0
— **graph_size** - le nombre de noeuds dans le graphe
Voilà comment ils sont initialisés :

4.2 Sélection du noeud de destination

La méthode onSelection() nous permet de lancer la construction du sous-graphe et le routage. Avant avoir sélectionné la destination, tous les noeuds sont considérés déconnectés, car il y a pas de destination.

```
@Override
public void onSelection() {
    parent = this;
    setIconSize(20);
    setColor(Color.yellow);
    racine = true;
}
```

FIGURE 1 – Méthode **onSelection()**

4.3 Méthode onClock()

Comme nous l'avons indiqué avant, la méthode onClock() est divisé en deux : les noeuds qui ne sont pas la destination et la destination.

1. **Le noeud n'est pas la destination** : Le noeud prend tous les messages reçus jusqu'à retrouver le message avec la distance la plus petite et le noeud qui l'a envoyé. Les messages contenant des distances sont des messages avec des flags "UPDATE".

```java
@Override
public void onClock(){
    List<Message> msgs = getMailbox();
    // If we are not the destination node and have received messages, we update our state
    if(!racine){
        Node new_parent = null;
        int new_hop = graph_size;

        // We look for the smaller distance we received
        for (Message m : msgs) {
            if (getOutLinkTo(m.getSender()) != null && m.getFlag().equals("UPDATE")) {
                int h = Integer.parseInt(String.valueOf(m.getContent()));
                if (h < new_hop) {
                    new_parent = m.getSender();
                    new_hop = h;
                }
            }
        }
```

FIGURE 2 – Méthode **onClock()**

Nous vérifions si le noeud est déconnecté ou pas. Si un noeud est déconnecté et qu'il a des voisins, ils vont augmenter leur 'distance' à la destination à chaque ronde jusqu'à arriver au nombre de noeuds. Si un noeud déconnecté n'a pas de voisins la distance la plus petite qu'il trouvera sera le nombre de noeuds.

```java
// If a node is disconnected from the destination node, it will become pink and the console will print it
if(new_hop == graph_size ){
    System.out.println("DISCONNECTED NODE : " + getID());
    setColor(Color.pink);
    send(parent, new Message( content: "",  flag: "ABANDON"));
    parent = null;
}
```

FIGURE 3 – Méthode **onClock()**

Si le noeud est toujours connecté, il faut juste vérifier si son parent est toujours le voisin le plus proche de la racine, si ce n'est pas le cas il doit changer de parent et il est un menteur.

```java
// We update our distance to the destination node and our parent if it isn't the closest to the destination node
else{
    setColor(null);
    hop = new_hop + 1;
    if(parent != new_parent){
        send(parent, new Message( content: "",  flag: "ABANDON"));
        send(new_parent, new Message( content: "",  flag: "ADOPTION"));
        parent = new_parent;
        setColor(Color.red);
    }
    sendAll(new Message(hop,  flag: "UPDATE"));
}
```

FIGURE 4 – Méthode **onClock()**

À la fin chaque noeud, menteur, déconnecté ou aucun des deux cas envoie à ses voisins

un message "UPDATE" avec sa distance à la racine.

2. **Le noeud est la racine** : La destination ou racine, envoie un message "UPDATE" à tous ses voisins avec son hop qui est initialisé à 0 et ne change pas.

```
// If we are the destination node, we just send a message with our hop
else{
    sendAll(new Message(hop, flag: "UPDATE"));
    setColor(Color.green);
}
```

FIGURE 5 – Méthode **onClock()**

4.4 Exemple

Afin de visualiser le fonctionnement de notre algorithme, voici différents exemples :
Légende :

— **Vert :** destination
— **Orange :** sommets possiblement menteurs
— **Rouge :** sommets menteurs
— **Rose :** sommets isolés

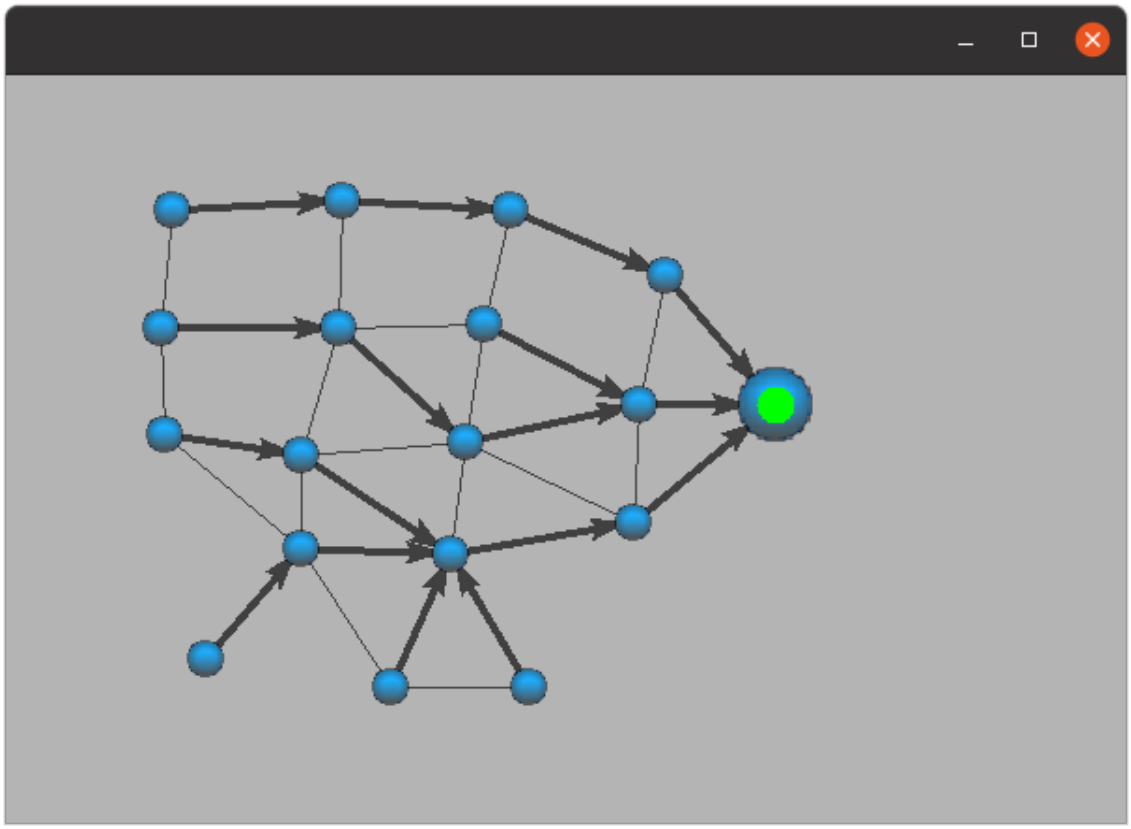

FIGURE 6 – Tout les sommets sont dirigés vers le sommet destination en Vert

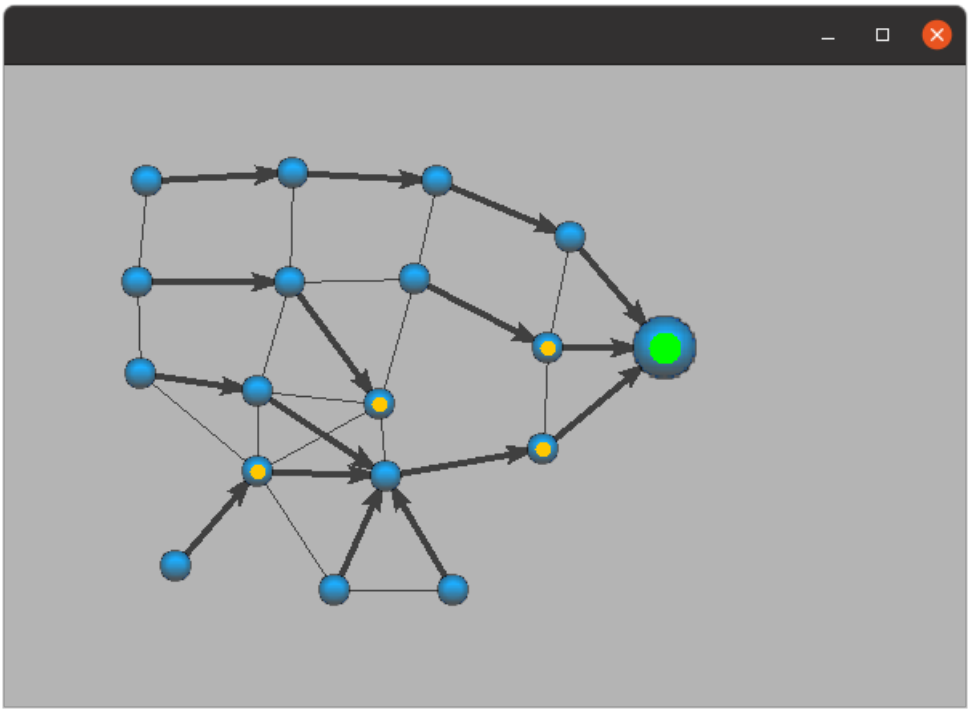

FIGURE 7 – En cassant un lien, les noeuds affectés deviennent des possibles menteurs en Orange

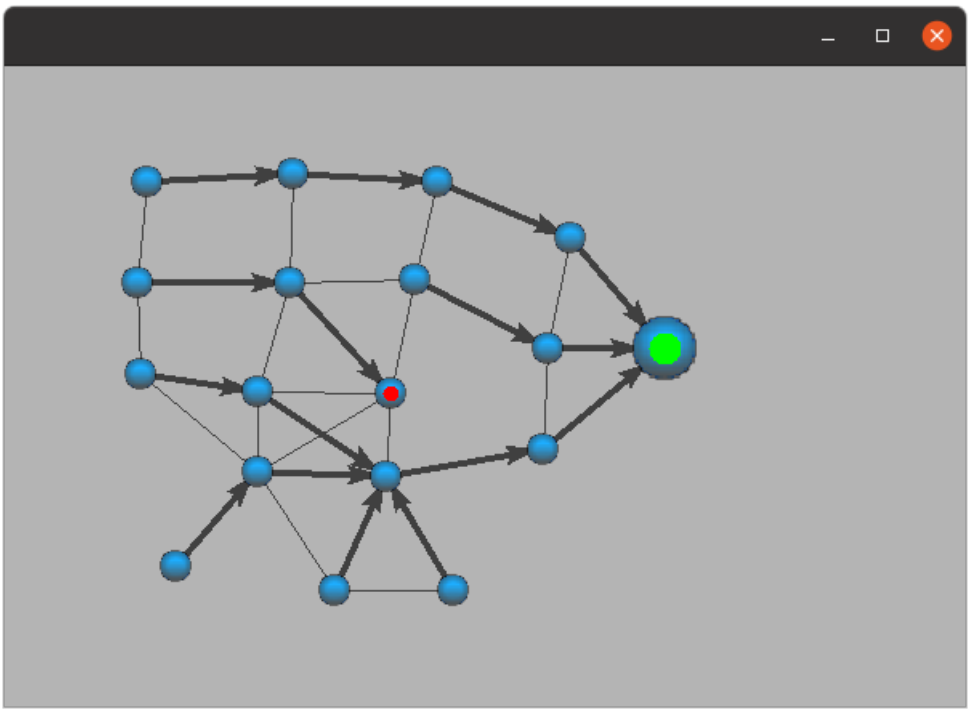

FIGURE 8 – Le vrai menteur passe à Rouge

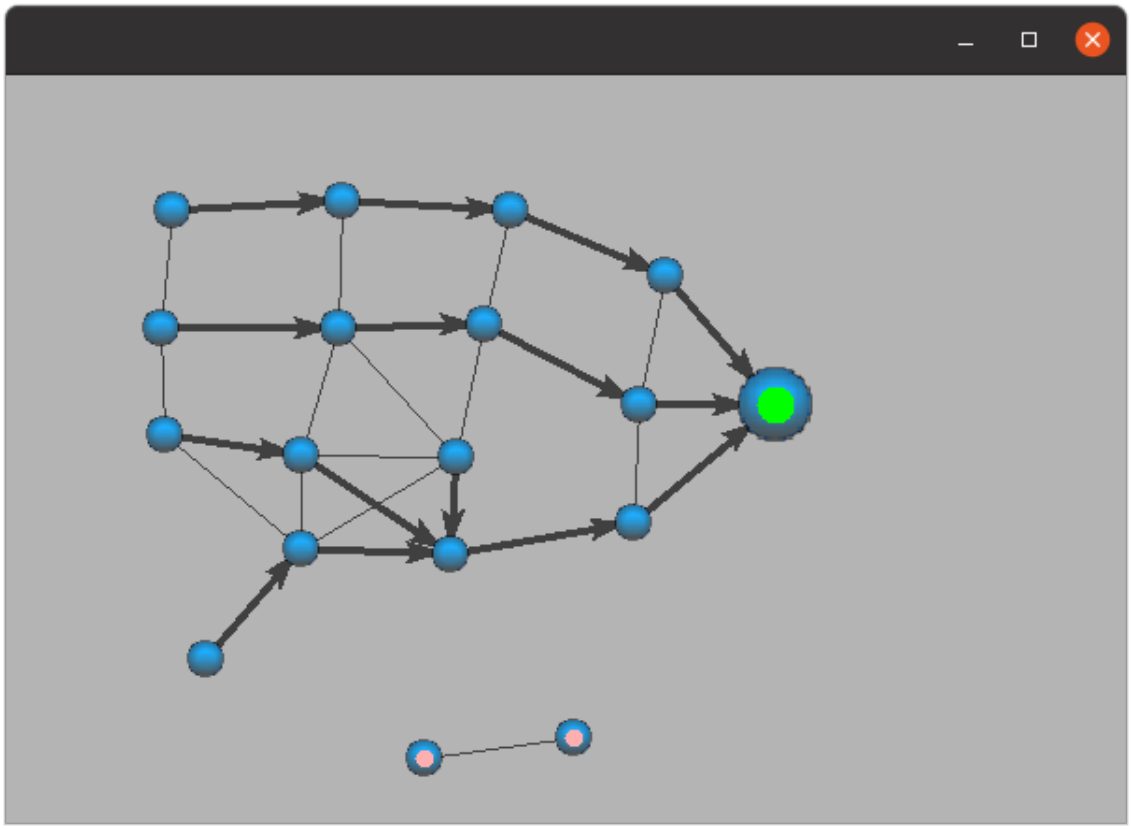

FIGURE 9 – Les sommets isolés passent à Rose

5 Limites

Pour le moment, chaque noeud connaît le nombre de noeuds dans le graphe. Cet argument doit être changé manuellement dans le code si nous prévoyons d'utiliser un nombre de noeuds différent à chaque fois. Si un mauvais nombre de noeuds est utilisé des noeuds étant pas déconnectés pourraient être considérés comme tel et vice versa.
Dans le futur, une bonne solution pourrait être d'avoir une fonction comptant le nombre de noeuds, ou alors permettre à l'utilisateur de changer le nombre depuis la console avant l'exécution de notre algorithme.

6 Conclusion

Ce projet montre les changements du graphe lorsque la topologie est modifiée. Si lors des changements, un noeud change de chemin vers la destination, le noeud devient un "menteur". Il s'agit alors de détecter les menteurs, de les afficher et réparer les erreurs pour avoir des chemins optimaux vers la racine après tels changements. Nous avons envisager l'ajout de nœuds. Comme à la suppression d'arête, une addition d'arête peut changer la distance (le nombre de saut pour arriver à la destination) dans le graphe.