Aix-Marseille Université
Master Informatique
Algorithmique Distribuée
2014

TP 3 : Élection dans les graphes arbitraires

1 Utilisation du simulateur *JBotSim*

Nous allons utiliser cette semaine aussi la bibliothèque “JBotSim" qui permet de simuler des algorithmes distribués sur des réseaux quelconques. La bibliothèque JBotSim est disponible au lien suivant :

`http://jbotsim.sourceforge.net/`

La documentation de la bibliothèque est disponible au lien suivant :

`http://jbotsim.sourceforge.net/javadoc/`

Exercice

1. Télécharger le fichier "SPT.zip" au lien suivant :

   `http://pageperso.lif.univ-mrs.fr/~arnaud.labourel/AD2014/SPT.zip`

   Ce fichier contient un programme implémentant la construction d'un arbre couvrant avec un initiateur unique. Vous devez compiler le programme java et l'exécuter avec la ligne commande suivante :

   `java -cp. : jbotsim.jar SPTstart`

   Il suffit de cliquer sur la fenêtre pour créer des nœuds et construire le réseau. Ensuite, il suffit de cliquer sur le bouton “Start" pour exécuter l construction de l'arbre couvrant du réseau que vous avez créé.
2. Dans le programme SPTstart il y a un seul initiateur dans l'algorithme (le nœud ayant l'identifiant égal à 10). Que se passe-t-il s'il y a plusieurs initiateurs ? Changez le programme écrit dans “SPTnode.java" de telle sorte qu'un cinquième des nœuds soit initiateur (par exemple en changeant la condition d'initiateur par ID % 5 == 0). Exécuter le programme modifié sur un graphe de 20 nœuds et observer l'exécution de l'algorithme. L'algorithme devrait construire une forêt couvrantes composée de 4 arbres.

2 Implémentation de l'algorithme YO-YO pour l'élection

On rappelle que l'algorithme YO-YO permet de résoudre l'élection dans les réseaux arbitraires munis d'UID. La première phase de l'algorithme convertit le réseau en un réseau orienté en orientant toutes les arêtes du plus petit identifiant vers le plus grand. Le réseau devient donc un DAG (graphe dirigé acyclique) ayant plusieurs *sources* (appelé sources en anglais) qui sont les nœuds ayant aucun voisins entrant (aucune flèche allant vers le nœud) et *puits* (appelé sinks en anglais) qui sont les nœuds ayant aucun voisins sortants (aucune flèche sortant du nœud). Les nœuds n'étant ni des puits, ni des sources sont appelés des nœuds *internes*. Les nœuds sources sont les candidats pour l'élection. L'algorithme enchaîne des phases appelées YO-YO pour éliminer des candidats jusqu'à qu'il n'en reste qu'un qui devient élu. Une phase de YO-YO commence par l'envoi par les nœuds sources d'un message <YO-, UID> à tous leurs voisins sortants. Ces messages sont propagés par les nœuds internes jusqu'aux puits. Chaque puits répond <-YO, UID, yes> au nœud dont l'IUD était le plus petit reçu parmi ceux reçus par le puits. Chaque puits envoie un message <-YO, UID, no> aux autres nœuds. Le nœud qui reçoit un message “no” change l'orientation du lien par lequel il a reçu le message. Par conséquent, le nœud n'est plus une source. Quand il reste plus qu'une source, celle-ci devient le Leader. Un pseudo-code de l'algorithme est donné dans la page suivante. Afin de rendre

plus rapide les phases de YO-YO suivante, on procède à l'élimination des liens et des nœuds inutiles. On utilise les deux règles suivantes : les puits ayant qu'un seul voisin entrant sont enlevés et lors d'une phase si le même nœud reçoit le même UID de plusieurs voisins entrants, il enlève tous les liens correspondants sauf un.

Exercice

1. Implémenter l'algorithme YO-YO en utilisant JBotSim.
   (a) Créer une classe *YoYoNode* qui hérite de la classe nœud et qui implémente les interfaces ClockListener and MessageListener. Cette classe devra implémenter l'algorithme YO-YO.
   (b) On va considérer des réseaux dont les nœuds ont des identifiants uniques (UID). Il faut donc assigner à chaque nœud créé un identifiant unique. Pour cela, on va utiliser un compteur en variable globale.

   ```
   public static int counter = 0;
   ```

   Quand un nœud est construit, on va fixer son UID à la valeur courante du compteur et incrémenter celui-ci.
   (c) On utilisera la méthode `void setColor(java.lang.String color)` de `Node` pour colorier les sources en rouge ("red") et les puits en en vert ("green"). Les liens ayant été enlevé par l'algorithme devront être colorié en blanc ("white").
   (d) Dans le fichier contenant la fonction `main`, ajouter un bouton "Start" contrôler l'exécution de l'algorithme (comme dans l'exemple de l'arbre couvrant). Each round of the algorithm should start on clicking the start button. At end of a round, each node sets the variable "Done" to true (to signify that it has finished the round). On clicking the start button, the variable "Done" is set to false, for each node.
2. Lancer le programme sur un graphe quelconque. Observer l'execution de votre algorithme et vérifier que son comportement est correct (la configuration finale devrait contenir qu'un seul sommet colorié en rouge le leader).

Pseudo-code for the YO-YO Algorithm

Each node has the following variables :
ID : (a unique identifier)
State ∈ { UNKNOWN, SOURCE, SINK, INTERNAL, DELETED, LEADER }
IN-LINKS : A list of incoming links.
OUT-LINKS : A list of outgoing links.
Initially the state of a node is UNKNOWN and the lists of incoming and outgoing links are empty.

Code for (State==UNKNOWN) :

```
Send(ID) to each neighbor;
Receive messages from all neighbors;
```

Upon receiving a message M on link l :

```
if (M.ID > my_ID) then
   Add link l to OUT-LINKS;
if (M.ID < my_ID) then
   Add link l to IN-LINKS;
if (size-of(IN-LINKS) == #neighbors) then
```

```
   Change State to SINK;
if (size-of(OUT-LINKS) == #neighbors) then
   Change State to SOURCE;
```

Code for (State==SOURCE) :
In each round k,

```
Send <YO-, ID, k> on each outgoing link;
Receive messages from outgoing links;
```

Upon receiving a message M on link l :

```
if (M is a "no" message) then
   Reverse the link l (move this link to IN-LINKS);
if (size-of(IN-LINKS) > 0 & size-of(OUT-LINKS) == 0) then
   Change State to SINK;
if (size-of(IN-LINKS) > 0 & size-of(OUT-LINKS) > 0) then
   Change State to INTERNAL;
if (size-of(IN-LINKS) == 0 & size-of(OUT-LINKS) == 0) then
   Change State to LEADER;
```

Code for (State==SINK) :
In each round k,

```
if (size-of(IN-LINKS) == 1) then
   Delete the last link and change State to DELETED;
```

Upon receiving messages on all incoming links :

```
MIN = minimum of all the ID's received.
Send <-YO, "yes"> to the link from which MIN is received;
Send <-YO, "no"> to all other incoming links;
if (received the same ID from 2 incoming links) then
   Delete one of these 2 links;
```

Code for (State==INTERNAL) :
In each round k,

```
if (size-of(OUT-LINKS) == 0) then
   Change State to SINK;
```

Upon receiving $< YO-, x, k >$ messages on all incoming links :

```
MIN = minimum of all the ID's received.
Send <YO-, MIN, k> message on each outgoing link;
if (received the same ID from 2 incoming links) then
    Delete one of these 2 links;
Send <-YO, "no"> to all incoming links except from which MIN is received;
Reverse the links on which you sent <-YO, "no"> message;
```

Upon receiving $< -YO$,"yes"> messages on all outgoing links :

```
Send <-YO, "yes"> on incoming links;
```

Upon receiving $< -YO$,"no"> messages on link l :

```
Send <-YO, "no"> on incoming links;
Reverse the link l (move this link to IN-LINKS);
```

(Note : For deleting a link, a node must communicate with the node on the other end of the link, by sending a <delete-link> message.)