Blain Hugo
4A ILC

ESIREM

Compte Rendu TP

Compte rendu des TPs de Systèmes distribués

Table des matières

I. Introduction : ... 3

II. TP n°2 : Implémentation d'algorithme de base ... 4

1. Calcul d'un arbre recouvrant en utilisant un parcours en profondeur ... 4

A . Rappel des règles pour cet algorithme : ... 4

B. Mise en place de l'algorithme : ... 4

C. Implémentation ... 6

4. Exemple ... 7

2. Calcul de l'ordre du graphe ... 8

A. Principe ... 8

B. Mise en place ... 9

C. Exemple ... 9

III. TP n°3 : Election d'un leader dans un arbre ... 10

A. Rappels sur les règles de réécriture de l'élection d'un leader dans un arbre ... 10

B. Mise en place ... 11

C. Exemple ... 11

IV. TP n°4 : Etoile ouverte ... 13

1. Arbre recouvrant ... 13

A. Rappel des règles de réécriture ... 13

B. Mise en place de l'algorithlme ... 13

C. Exemple ... 14

2. Election d'un leader ... 16

A. Rappel des règles de réécriture ... 16

B. Mise en place ... 16

C. Exemple ... 17

V. TP n°5 : Etoile fermée ... 18

1. Arbre recouvrant ... 18

A. Rappel des règles de réécriture ... 18

B. Mise en œuvre ... 18

C. Exemple ... 19

2. Election d'un leader dans un arbre ... 20

A. Rappel des règles ... 20

B. Mise en place ... 20

C. Exemple ... 21

3. Détection de la terminaison locale ... 22

B. Election leader ... 23

Conclusion ... 24

I. Introduction :

Au cours de ces travaux pratiques nous étudierons et manipulerons différents algorithmes de systèmes distribués vus en cours. Pour ceci nous utiliserons le langage JAVA et une librairie externe « ViSiDia.jar ». Nous verrons les algorithmes d'arbre recouvrant, d'élection d'un leader dans un arbre et du calcul d'ordre dans un graphe, ceci avec trois différents types de synchronisations : handshake, étoile ouverte et étoile fermée.

Dans ce compte rendu une partie du code sera décrite, pour voir tous les programmes, suivez le lien GitHub suivant :

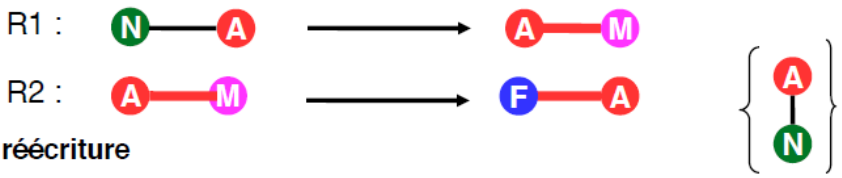
https://github.com/HugoBlain/Systeme_Distribues_TP_ViSiDia

II. TP n°2 : Implémentation d'algorithme de base

Au cours de ce second TP nous allons écrire deux algorithmes basiques à l'aide de l'API ViSiDia, disponible sur le site http://visidia.labri.fr et de l'exemple de code source fourni au premier TP. Nous commencerons par celui d'un arbre recouvrant puis nous en verrons un autre permettant de calculer l'ordre d'un graphe.

1. Calcul d'un arbre recouvrant en utilisant un parcours en profondeur

A . Rappel des règles pour cet algorithme :



Règles de réécriture

- Si un sommet est étiqueté N et que son voisin est étiqueté A, alors le sommet passe à l'état A, son voisin passe à l'état M et l'arête entre ces deux sommets est marquée.
- Si un sommet est étiqueté A est connecté a un voisin étiqueté M par une arête marquée (et **ce sommet A n'a aucun voisin N**), alors le sommet A passe à l'état F et son voisin passe à l'état A (l'arête reste marquée).

A noter que nous risquons d'avoir du mal à vérifier si un lien est marqué. Pour contourner cette difficulté, lors de l'application de la première règle, le nœud qui passe à l'état « A » va enregistrer le port sur lequel se situe son père grâce à la variable *neighborDoor.* Ainsi, lors de l'application de la seconde règle, le nœud peut vérifier que le voisin avec lequel la synchronisation est établie est bien situé sur le port où se trouve son père.

B. Mise en place de l'algorithme :

Comme vu dans l'introduction, nous allons écrire le code de notre algorithme en JAVA avant de l'importer dans ViSiDia. Nous utiliserons notamment des classes propres à ViSiDia qui nous permettront de coder le comportement voulu pour chaque sommet du graphe.

Pour l'algorithme de notre arbre recouvrant nous utiliserons une synchronisation locale entre deux nœuds, dans ViSiDia appelée *LC0_Algorithm.* Chaque nœud envoie un entier à ses voisins, cet entier vaut 0 pour tous les voisins à l'exception de celui que le nœud choisit pour la synchronisation, dans ce cas l'entier envoyé est strictement positif. Si deux nœuds adjacents se choisissent mutuellement, la synchronisation a lieu. Le nœud ayant envoyé l'entier le plus grand est chargé d'appliquer les règles de réécriture si la configuration le permet. A noter que si les entiers sont égaux alors la synchronisation échoue et que plusieurs synchronisations peuvent avoir lieu en même temps si elle n'impliquent pas les mêmes nœuds.

La classe *LC0_Algorithm* compte 4 méthodes à implémenter :

- `getDescription()` : renvoie sous forme de *String* la une description de l'algorithme mis en place.
- `beforeStart()` : code exécuté par le nœud (tous) au début avant le lancement des synchronisations.
- `onStarCenter()` : code exécuté par le nœud chargé d'appliquer les règles de réécriture.
- `clone`() : renvoie un nouvel objet de la classe.

Au début notre code ressemble donc à ceci :

```java
package defaut;

import visidia.simulation.process.algorithm.LC0_Algorithm;
import visidia.simulation.process.edgestate.MarkedState;

public class SpanningTree_TP2 extends LC0_Algorithm {

    @Override
    public String getDescription() {
        // TODO Auto-generated method stub
        return null;
    }


    @Override
    protected void beforeStart() {
        // TODO Auto-generated method stub

    }

    @Override
    protected void onStarCenter() {
        // TODO Auto-generated method stub

    }

    @Override
    public Object clone() {
        // TODO Auto-generated method stub
        return null;
    }

```

Figure 1 – Base de notre programme

C. Implémentation

Pour des raisons évidentes je ne décrirai pas, ni ici ni dans le reste de ce compte rendu les méthodes *getDescription()* et *clone()*, pour plus de détails le code est disponible sur le dépôt GitHub (https://github.com/HugoBlain/Systeme_Distribues_TP_ViSiDia).

Pour appliquer la deuxième règle nous devons pouvoir vérifier que le nœud n'a plus aucun voisin à l'état « N ». Pour vérifier ceci, nous devons faire en sorte que chaque nœud retienne l'état de ses voisins dans une table, chaque nœud aura donc en plus dans ses attributs un tableau contenant une liste de l'état de ses voisins. Etant donné qu'un nœud ne peux connaitre que l'état du voisin avec lequel il ses synchronise, au départ tous les nœuds concidereront que leurs voisins sont à l'état « N » et à chaque fois qu'une synchronisation aura lieu, la table du nœud sera mise à jour.

```java
@Override
protected void beforeStart() {
    // chaque noeud sauvegarde son état dans la variable "label"
    //   remarque: tous les noeuds sont à l'état "N" au début sauf un à "A"
    setLocalProperty("label", vertex.getLabel());
    // on initialise une valeur à -1 pour retenir plus tard le port sur lequel se trouve le père
    setLocalProperty("portDuPere", -1);
    // on considère qu'au départ tous les voisins sont à l'état "N" (pas moyen de savoir ou est le "A")
    this.etatsVoisin = new String[vertex.getDegree()];
    for(int i = 0; i<vertex.getDegree(); i++) {
        this.etatsVoisin[i] = "N";
    }
}
```

Figure 2 – beforeStart()

```java
@Override
protected void onStarCenter() {

    // mise à jour de la table des états des voisins
    this.etatsVoisin[neighborDoor] = getNeighborProperty("label").toString();

    // 1ere regle
    // on verifie si le neoud qui applique la règle est bien un noeud à l'etat "N" et son voisin à "A"
    if (getLocalProperty("label").equals("N") && getNeighborProperty("label").equals("A")){
        // le noeud qui applique la regle passe à l'etat "A" et son voisin à "M"
        setLocalProperty("label", "A");
        setNeighborProperty("label", "M");
        // le noeud qui applique la regle sauvegarde le port sur lequel se trouve son pere
        setLocalProperty("portDuPere", neighborDoor);
        // on marque l'arete
        setDoorState(new MarkedState(true), neighborDoor);
        // mise à jour de la table des états des voisins
        this.etatsVoisin[neighborDoor] = getNeighborProperty("label").toString();
    }
```

Figure 3 – onStarCenter() partie 1

```
// 2eme regle
// on verifie si le neoud qui applique la règle est bien un noeud à l'etat "A" et son voisin à "M"
if (getLocalProperty("label").equals("A") && getNeighborProperty("label").equals("M")){
    // on verifie que l'arete est bien marquée entre les deux noeuds
    // cad, que le voisin à l'etat "M" se situe bien sur le port du père du noeud "A"
    if(neighborDoor == (int) getLocalProperty("portDuPere")) {
        // on verifie si le noeud à l'etat "A" qui applique la regle n'a plus aucun voisin à l'etat "N"
        boolean voisinN = false;
        for(int i=0; i<vertex.getDegree(); i++) {
            if(this.etatsVoisin[i].equals("N"))
                voisinN = true;
        }
        // si aucun voisin à l'état "N"
        if (!voisinN) {
            // le noeuds changent d'état
            setLocalProperty("label", "F");
            setNeighborProperty("label", "A");
        }
    }
}
```

Figure 4 - onStarCenter() partie 2

4. Exemple

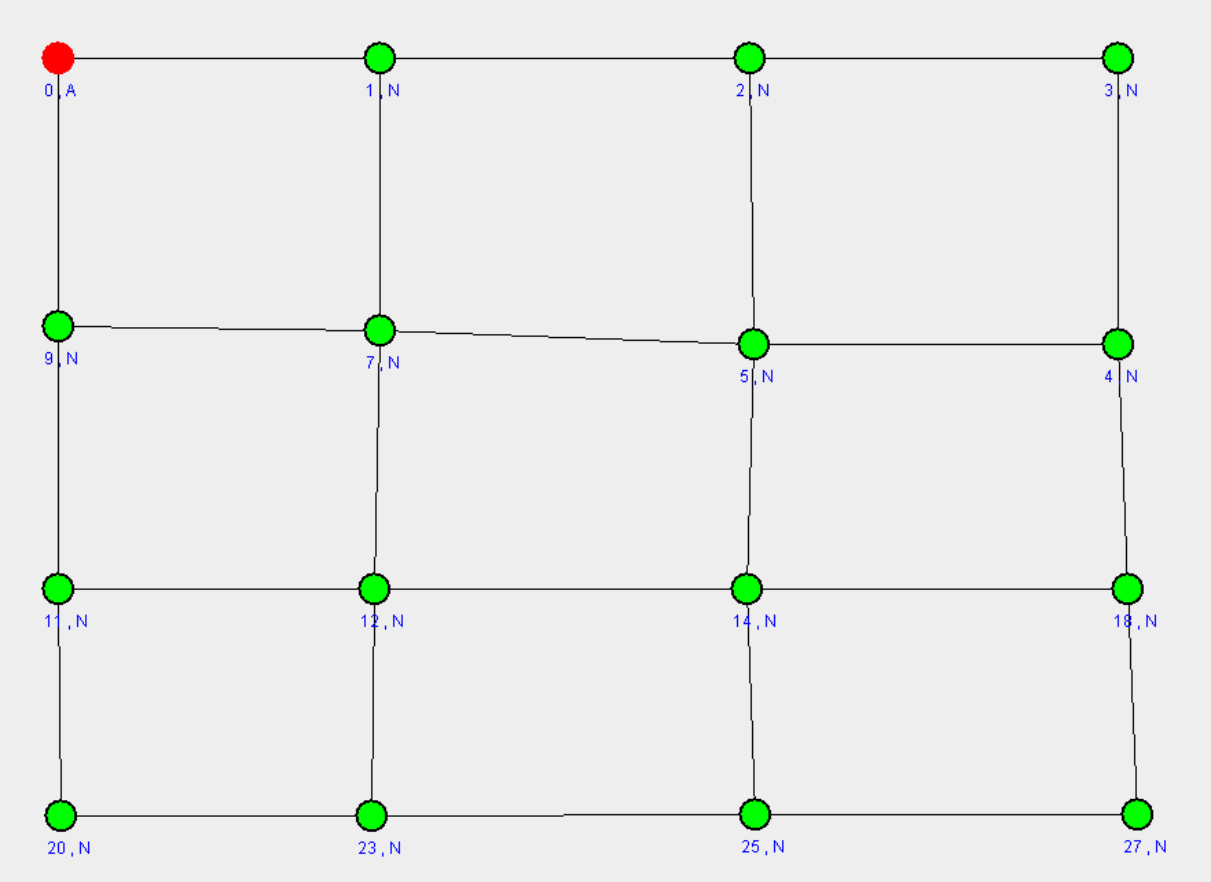


Figure 5 - Graphe de départ

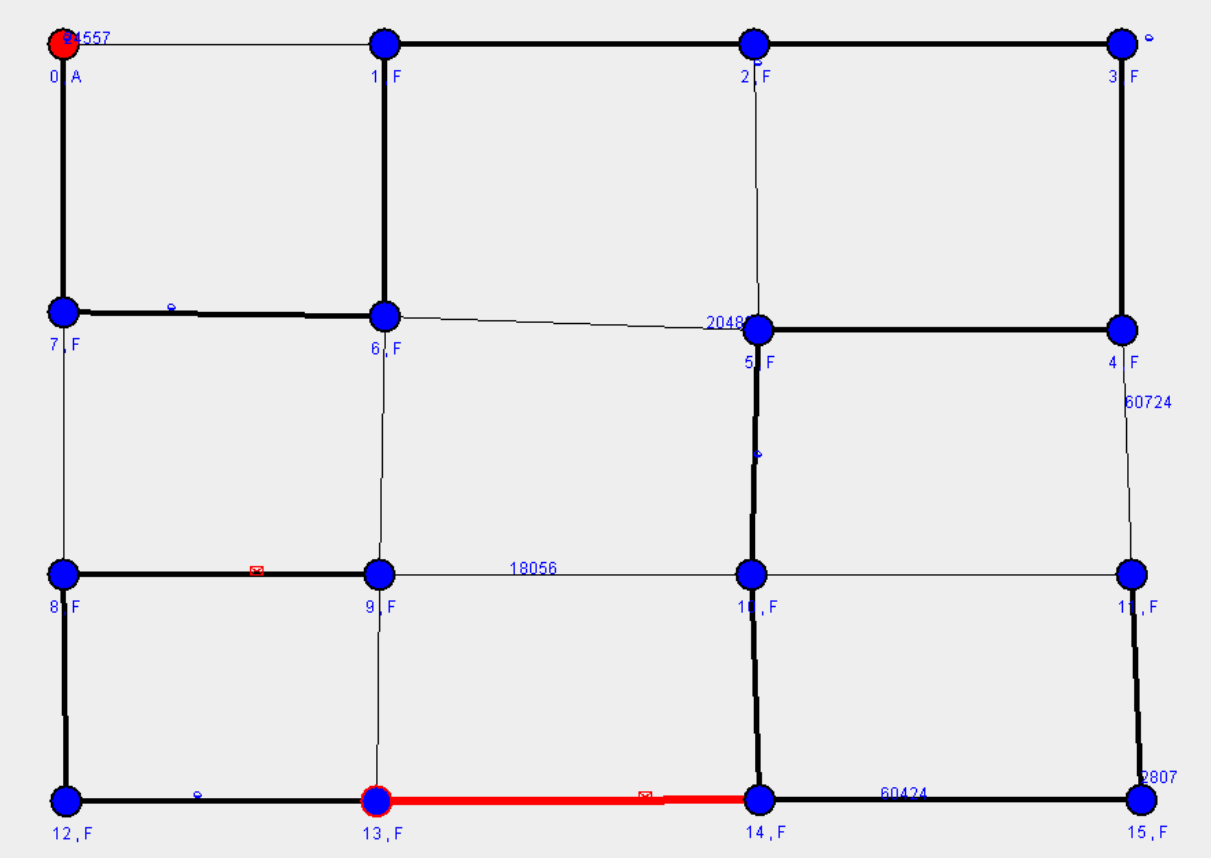


Figure 6 - Arbre recouvrant

2. Calcul de l'ordre du graphe

A. Principe

En nous inspirant du programme précédant nous allons en écrire un nouveau permettant de calculer l'ordre d'un graphe. Pour rappel, l'ordre d'un graphe est le nombre de sommet que compte ce graphe.

Puisque l'arbre recouvrant passe par tous les sommets sans faire de cycle, cela nous est très utile. A partir du précédant algorithme, on ajoute simplement un compteur à chacun des nœuds, ce compteur vaut à la base 1 (chaque nœud se compte lui-même). Ce compteur sera utilisé à chaque fois que la deuxième règle est appliqué : une fois qu'un nœud applique cette règle, il passe à l'état « F » et envoie son compteur au nœud auquel il s'est synchronisé et l'additionne au compteur de celui-ci. Ce qui fait qu'une fois que tous les nœuds sont passés à l'état « M » (sauf un dernier à « A »), l'arbre est parcourus, la deuxième règle va s'appliquer et les compteurs vont commencer à remonter petit à petit jusqu'au nœud initialement à l'état « A » qui pourra ainsi savoir le nombre de sommet dans le graphe.

Pour afficher ce compteur et le nombre totale de graphe, on utilise la méthode *putProperty().*

B. Mise en place

- Dans la méthode *beforeStart(),* on ajoute une variable compteur à 1 avec la méthode *setLocalProperty().*
- Dans la méthode *onStarCenter(),* dans la partie où on applique la seconde règle, on ajoute le code permettant d'ajouter au compteur du nœud voisin qui passe à l'état « A » la valeur du compteur du nœud qui applique la règle.
- On affiche le compteur de chaque nœud avec la méthode *putProperty()*.

```
// si aucun voisin à l'état "N"
if (!voisinN) {
    // le noeuds changent d'état
    setLocalProperty("label", "F");
    setNeighborProperty("label", "A");
    // on envoie le compteur au voisin pour qu'il l'ajoute au sien
    int compteurLocal = (int) getLocalProperty("compteur");
    int compteurVoisin = (int) getNeighborProperty("compteur");
    setNeighborProperty("compteur", compteurLocal + compteurVoisin);
}
```

C. Exemple

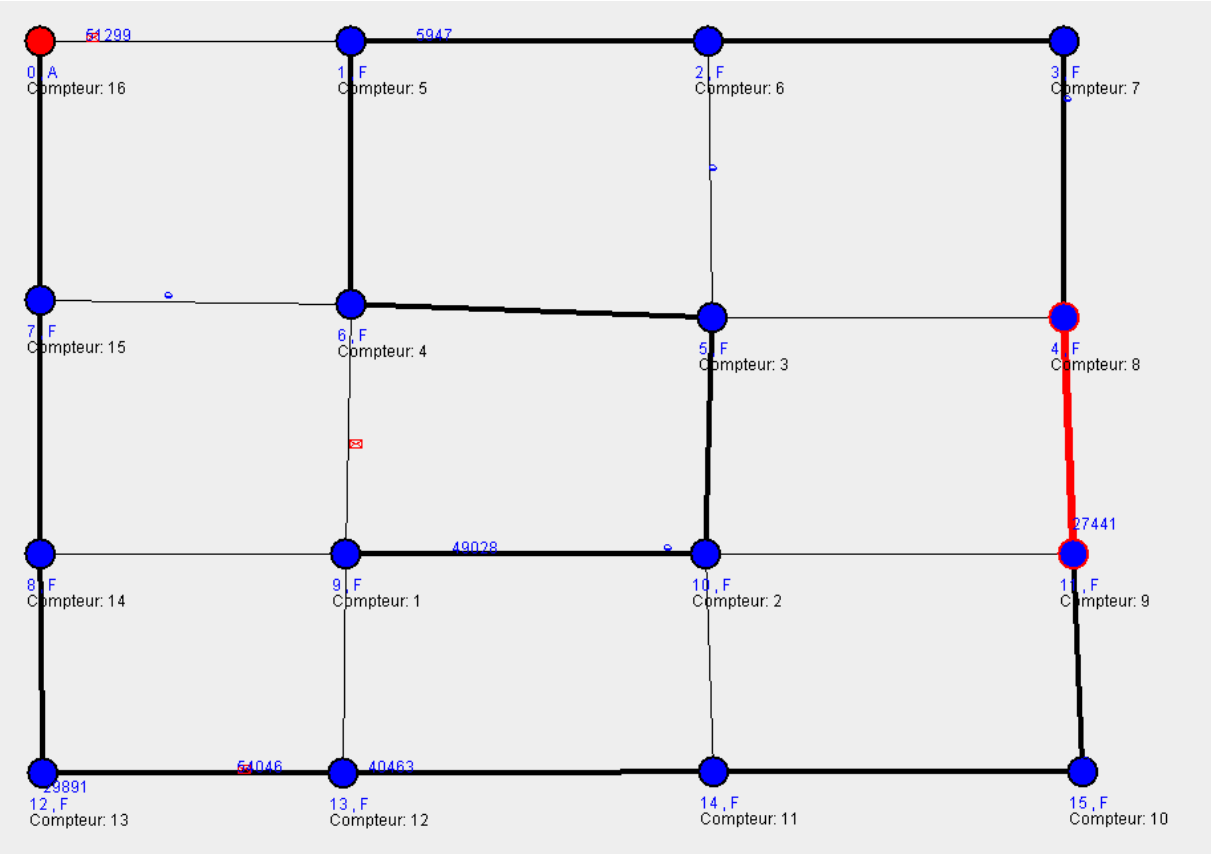


Figure 7 - Calcul ordre du graphe

III. TP n°3 : Election d'un leader dans un arbre

Le but de ce TP est d'écrire un algorithme permettant d'élire un leader dans un arbre. Pour ceci on utilisera une synchronisation locale de type *handshake* comme dans le TP précédant, nous utiliserons encore une fois la classe *LC0_Algorithm.*

A. Rappels sur les règles de réécriture de l'élection d'un leader dans un arbre

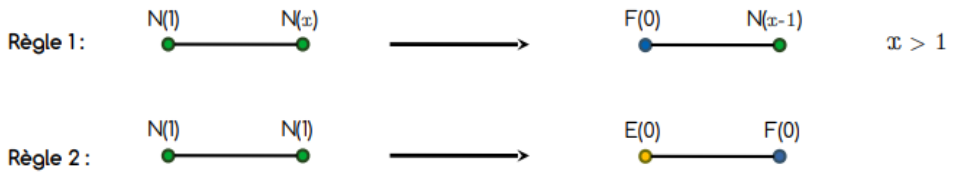
Elagage progressif :

- Au départ, tous les nœuds sont dans un état neutre, « N »
- Au fil de l'algorithme, les nœuds passeront soit à l'état non élu, soit à l'état élu
- Les états non élu et élu sont finaux (lorsqu'un nœud passe dans l'un de ces états, il y reste jusqu'à la fin de l'algorithme)
- Lorsque l'algorithme se termine, tous les nœuds sont à l'état non élu, sauf un et un seul qui est à l'état élu

On appelle une « feuille » un nœud qui n'a plus qu'un seul voisin à l'état « N ». Progressivement ces feuilles vont être éliminées et vont passer à l'état « F », non élu, ce qui fait que de nouveaux nœuds vont à leur tour devenir des feuille et ainsi de suite. On continue avec ce fonctionnement jusqu'à ce qu'il ne reste plus que deux feuilles, dans ce cas c'est le nœud qui applique la règle lors de la synchronisation qui est élu leader.

Version utilisant le handshake

- Au départ tous les nœuds sont dans un état neutre N
- Chaque nœud dispose d'un compteur de voisins



Comme expliqué juste en haut, x désigne le nombre de voisin à l'état « N » que compte un nœud. Lorsqu'on applique la première règle, on élimine un nœud qui n'a qu'un seul voisin à l'état « N », une feuille, et donc immanquablement le nombre de voisin à l'état « N » du nœud qu'il reste diminue de 1.

B. Mise en place

```java
@Override
protected void beforeStart() {
    // chaque noeud sauvegarde son état dans la variable "label"
    //   remarque: tous les noeuds sont à "N" au début
    setLocalProperty("label", vertex.getLabel());
    // chaque noeud sauvegarde son nombre de voisin
    setLocalProperty("nbrVoisinN", vertex.getDegree());
    // on affiche
    putProperty("Affichage", "Voisin N = "+getLocalProperty("nbrVoisinN"), SimulationConstants.PropertyStatus.DISPLAYED);
}
```

Figure 8 - Méthode beforeStart()

```java
// en cas de synchro
@Override
protected void onStarCenter() {
    // on commence par checker si les deux noeuds sont bien des N
    if (getLocalProperty("label").equals("N") && getNeighborProperty("label").equals("N")){
        // on recupère le nombre de voisin à N des deux noeuds
        int nbrVoisinLocal = (int) getLocalProperty("nbrVoisinN");
        int nbrVoisinVoisin = (int) getNeighborProperty("nbrVoisinN");
        // cas de figure 1, le plus courant :
        //     N(1) -- N(x) --> F(0) -- N(x-1)
        if (nbrVoisinLocal == 1 && nbrVoisinVoisin > 1 ) {
            setLocalProperty("label", "F");
            setLocalProperty("nbrVoisinN", 0);
            setNeighborProperty("nbrVoisinN", nbrVoisinVoisin-1);
        }
        // cas de figure 2, quand il ne reste plus que deux noeud "N" :
        //     N(1) -- N(1) --> E(0) -- F(0)
        else if (nbrVoisinLocal == 1 && nbrVoisinVoisin == 1) {
            setLocalProperty("label", "E");
            setLocalProperty("nbrVoisinN", 0);
            setNeighborProperty("label", "F");
            setNeighborProperty("nbrVoisinN", 0);
        }
        // Affichage: on met a jour l'affichage
        if(getLocalProperty("label").equals("F")) {
            putProperty("Affichage", "", SimulationConstants.PropertyStatus.DISPLAYED);
        }
        else if (getLocalProperty("label").equals("E")) {
            putProperty("Affichage", "LEADER", SimulationConstants.PropertyStatus.DISPLAYED);
        }
        else {
            putProperty("Affichage", "Voisin N = "+getLocalProperty("nbrVoisinN"), SimulationConstants.PropertyStatus.DISPLAYED);
        }
    }
}
```

Figure 9 - Méthode onStarCenter()

C. Exemple

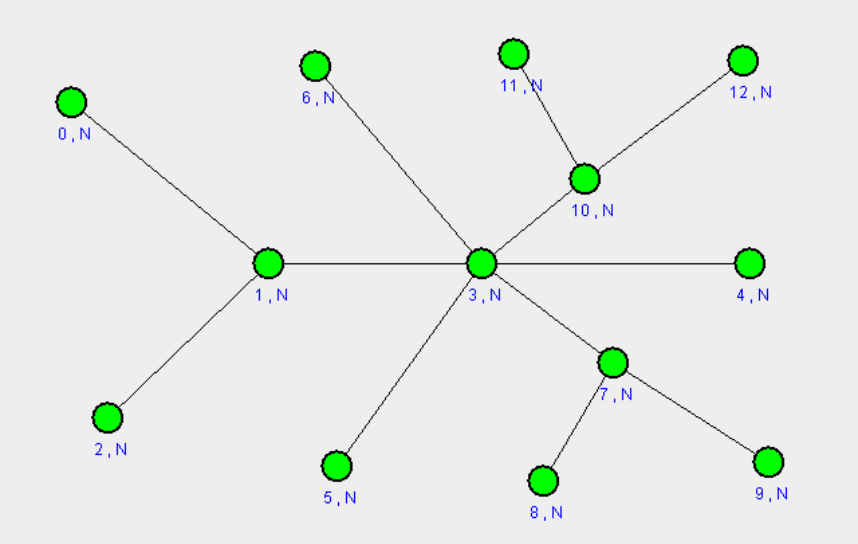


Figure 10 - Graphe de base

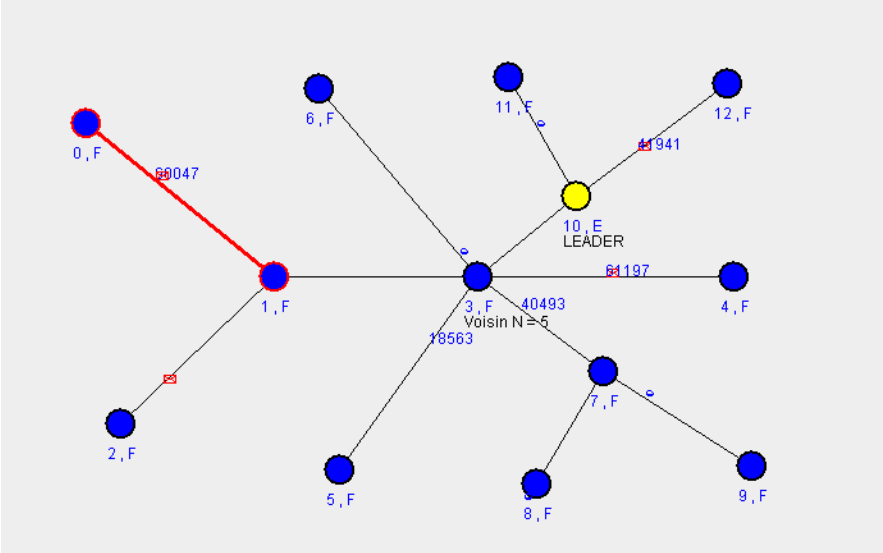


Figure 11 - Election Leader

Comme on peut le voir, l'algorithme s'est bien déroulé :

- Tous les nœuds sont à l'état « N » au départ
- On affiche bien le nombre de voisin à l'état « N » de chacun grâce à la méthode *putProperty()*
- Enfin il n'y a plus aucun autre nœud à l'état « N » lorsque le leader est élu.

IV. TP n°4 : Etoile ouverte

Le but de ce TP est d'écrire des algorithmes qui s'appuient sur une synchronisation locale de type étoile ouverte, nous utiliserons donc cette fois la classe de ViSiDia *LC1_Algorithm.*

Pour rappel, lorsqu'une synchronisation de type ouverte est établie, le centre de l'étoile est chargé d'appliquer les règles de réécriture. Ce dernier peut modifier :

- Son état
- Les états des arêtes qui le lient à ses voisins
- Attention : Le nœud ne peut pas modifier l'état de ses voisins

1. Arbre recouvrant

A. Rappel des règles de réécriture

Comme pour l'algorithme d'arbre recouvrant du TP n°2, au départ, tous les sommets du graphes doivent être à l'état « N » sauf un à l'état « A », celui qui initialise le calcul.

Règles de réécriture

- Si le centre de l'étoile est étiqueté N et qu'il possède au moins un voisin étiqueté A, il en choisit un (appelons **X** ce voisin)
- Le centre change son étiquette en A
- Le centre marque l'arête qui le lie à **X**

B. Mise en place de l'algorithlme

Tout d'abord comme on la dit au début de ce TP, nous n'utiliserons plus la classe *LC0_Algorithm,* mais à la place la *LC1_Algorithm.* Elle compte les 4 mêmes méthodes avec lesquelles nous travaillons : *getDescription(), beforeStart(), onStarCenter()* et *clone().*

Comme nous allons devoir vérifier si un nœud a des voisins à l'état « A » ou non, nous allons utiliser une nouvelle méthode appartenant à cette classe, getActiveDoors(). Cette méthode renvoie une *ArrayList* contenant les numéros de ports sur lesquels les voisins d'un nœud sont connectés.

```java
@Override
protected void beforeStart() {
    // chaque noeud sauvegarde son état dans la variable "label"
    //   remarque: tous les noeuds sont à "N" au début sauf un à "A" qui initiera le calcul
    setLocalProperty("label", vertex.getLabel());
}
```

```java
@Override
protected void onStarCenter() {
    // si le centre de l'étoile est "N"
    if (getLocalProperty("label").equals("N")) {
        // on check tous les voisins du noeud et on en choisit un des "A" si il en a (on stocke son numéro de port)
        int portVoisinA = -1;
        for (int i = 0; i < getActiveDoors().size(); i++) {
            int numPort = getActiveDoors().get(i);
            if (getNeighborProperty(numPort, "label").equals("A")) {
                portVoisinA = numPort;
            }
        }
        // si on a bien au moins un voisin "A"
        if (portVoisinA != -1) {
            // le noeud passe aussi à "A"
            setLocalProperty("label", "A");
            // on marque l'arete vers ce noeud
            setDoorState(new MarkedState(true), portVoisinA);
        }
    }
}
```

C. Exemple

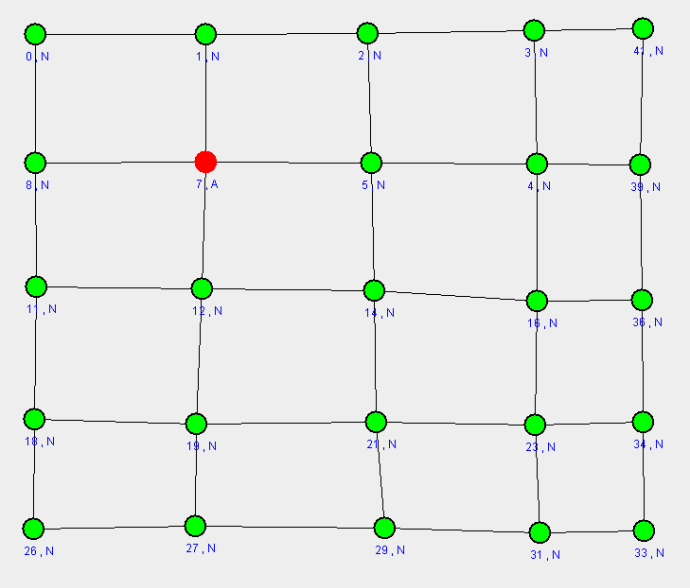


Figure 12 - Graphe de base

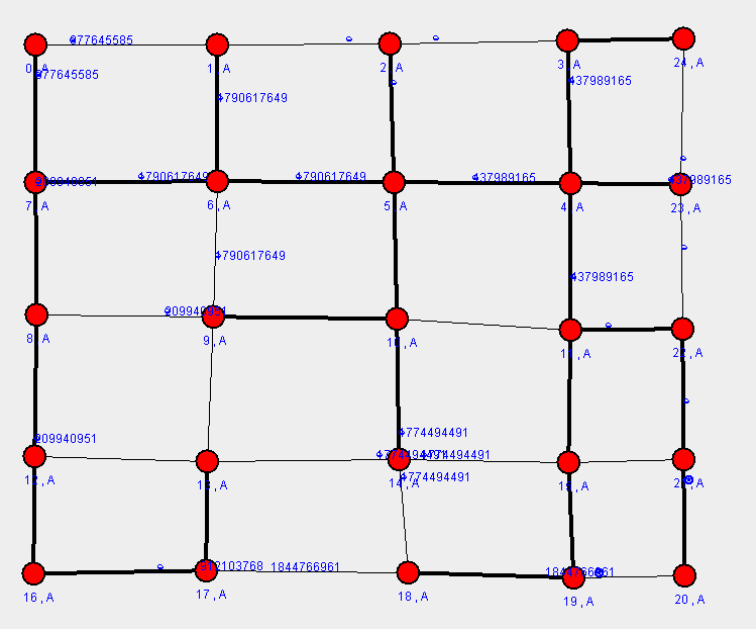


Figure 13 - Arbre recouvrant étoile ouverte

On peut noter qu'avec cette méthode utilisant la synchronisation de type étoile ouverte, nous obtenons l'arbre recouvrant beaucoup plus vite qu'avec la première (handshake).

2. Election d'un leader

A. Rappel des règles de réécriture

Pour élire un leader en utilisant une étoile ouverte :

- Si le centre de l'étoile est à l'état « N » et qu'il ne possède qu'un seul voisin « N » alors c'est une « feuille » et ce centre passe à l'état « F ». Il est éliminé.
- Si le centre est à l'état « N » et qu'il n'a aucun autre voisin aussi à l'état « N » alors ce centre est le dernier non éliminé, il passe à l'état « E », il est élu leader.

B. Mise en place

```
@Override
protected void beforeStart() {
    // chaque noeud sauvegarde son état dans la variable "label"
    //   remarque: tous les noeuds sont à "N" au début
    setLocalProperty("label", vertex.getLabel());
}
```

```
@Override
protected void onStarCenter() {
    // si le centre de l'étoile est "N"
    if (getLocalProperty("label").equals("N")) {
        // on compte son nombre de voisin à l'état "N"
        int nbrVoisinN = 0;
        for (int i=0; i<getActiveDoors().size(); i++) {
            int numPort = getActiveDoors().get(i);
            if (getNeighborProperty(numPort, "label").equals("N")) {
                nbrVoisinN ++;
            }
        }
        // si il le noeud n'a qu'un seul voisin "N", c'est une feuille, on l'élimine
        if(nbrVoisinN == 1) {
            setLocalProperty("label", "F");
        }
        // si il n'en a plus, il a gagné, c'est l'élu
        else if (nbrVoisinN == 0) {
            setLocalProperty("label", "E");
        }
    }
}
```

C. Exemple

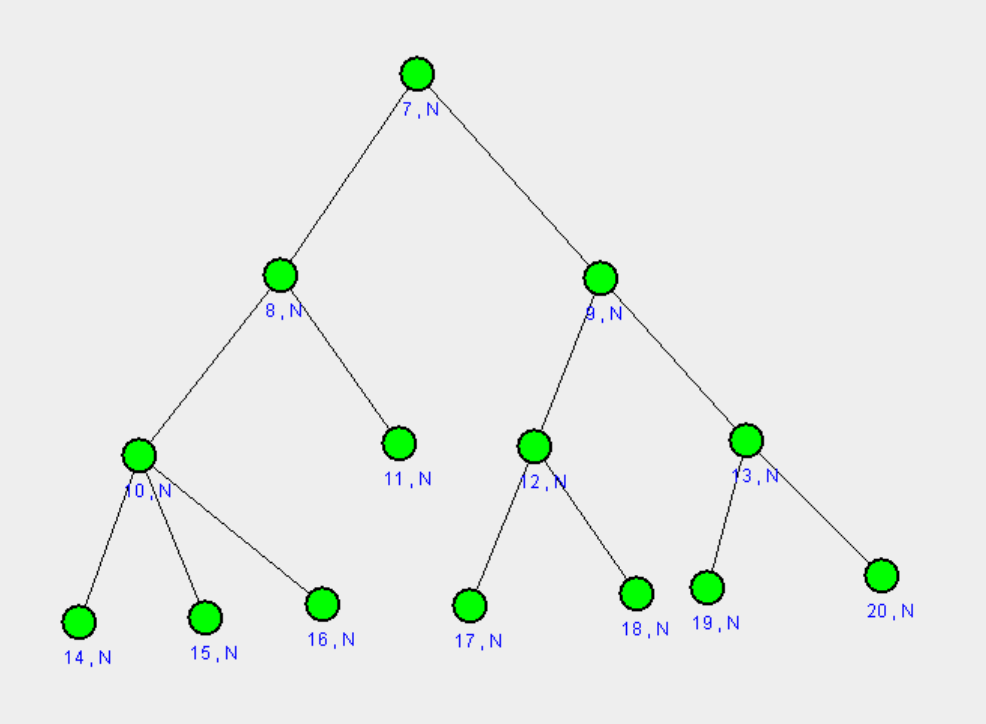


Figure 14 - Graphe de base

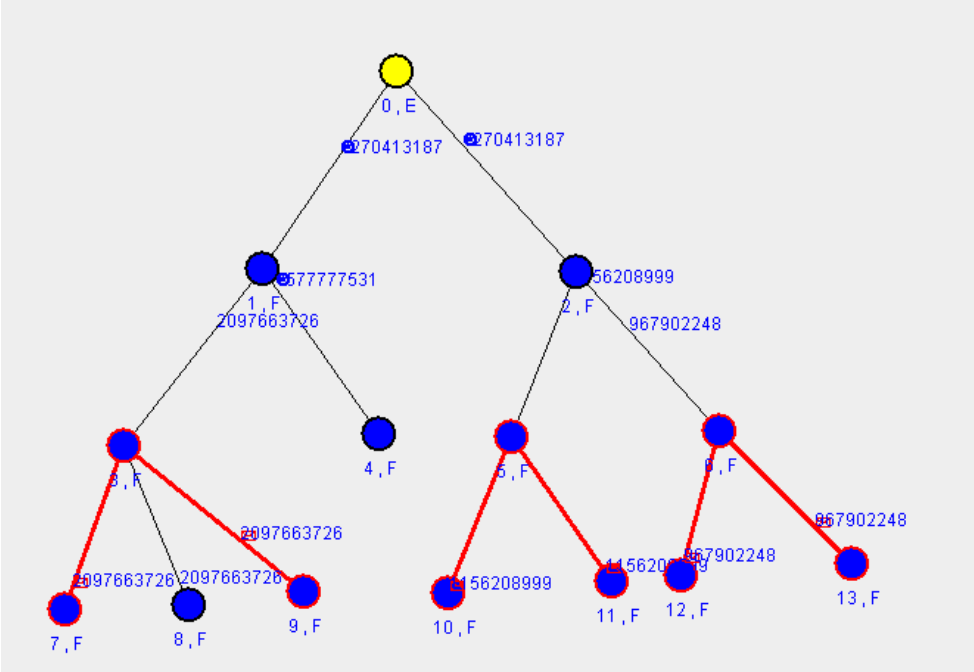


Figure 15 - Exemple élection leader étoile ouverte

Voici un exemple de résultat. Comme pour l'algorithme d'arbre recouvrant de la première partie du TP, celui-ci aussi est beaucoup plus rapide avec la synchronisation en étoile ouverte.

V. TP n°5 : Etoile fermée

Le but de ce TP est d'écrire des algorithmes qui s'appuient sur une synchronisation locale de type étoile fermée, on utilisera pour cela la classe *LC2_Algorithm.*

Pour rappel, lorsqu'une synchronisation de type étoile fermée est établie, le centre de l'étoile est chargé de l'application des règles de réécriture. Ce dernier peut modifier :

- Son état
- Les états des arêtes qui le lient à ses voisins
- Les états de ses voisins

1. Arbre recouvrant

A. Rappel des règles de réécriture

Comme toujours on supposera qu'au départ tous les nœuds sont dans un état « N » sauf un à l'état « A » qui initiera le calcul.

- Si le centre de l'étoile est étiqueté A
  - Il réétiquette tous ses voisins N en A
  - Il marque les arêtes respectives qui le lie à ces voisins (nouvellement étiquetés A)

B. Mise en œuvre

```
@Override
protected void beforeStart() {
    // chaque noeud sauvegarde son état dans la variable "label"
    //   remarque: tous les noeuds sont à l'état "N" au début sauf un à "A"
    setLocalProperty("label", vertex.getLabel());
}
```

```java
@Override
protected void onStarCenter() {
    // on vérifie si le noeud est à l'état "A"
    if (getLocalProperty("label").equals("A")){
        // on check l'état de tous ses voisin
        for (int i=0; i<getActiveDoors().size(); i++) {
            int numPort = getActiveDoors().get(i);
            // si le i ème voisin est à l'état "N"
            if (getNeighborProperty(numPort, "label").equals("N")) {
                // il passe à l'état "A"
                setNeighborProperty(numPort, "label", "A");
                // on marque l'arete
                setDoorState(new MarkedState(true), numPort);
            }
        }
    }
}
```

C. Exemple

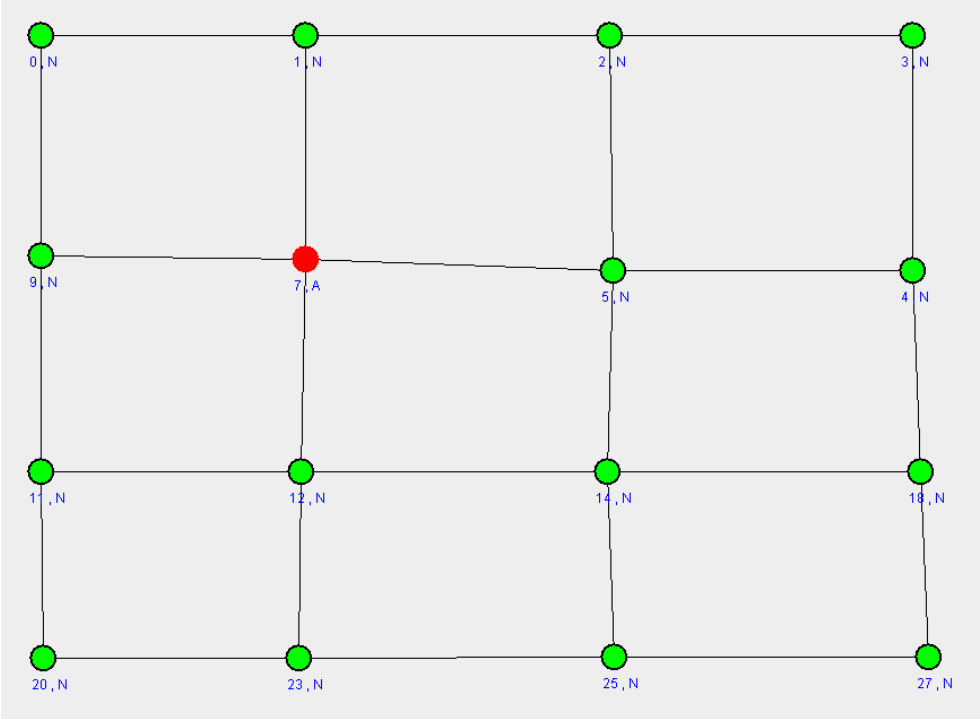


Figure 16 - Graphe de base

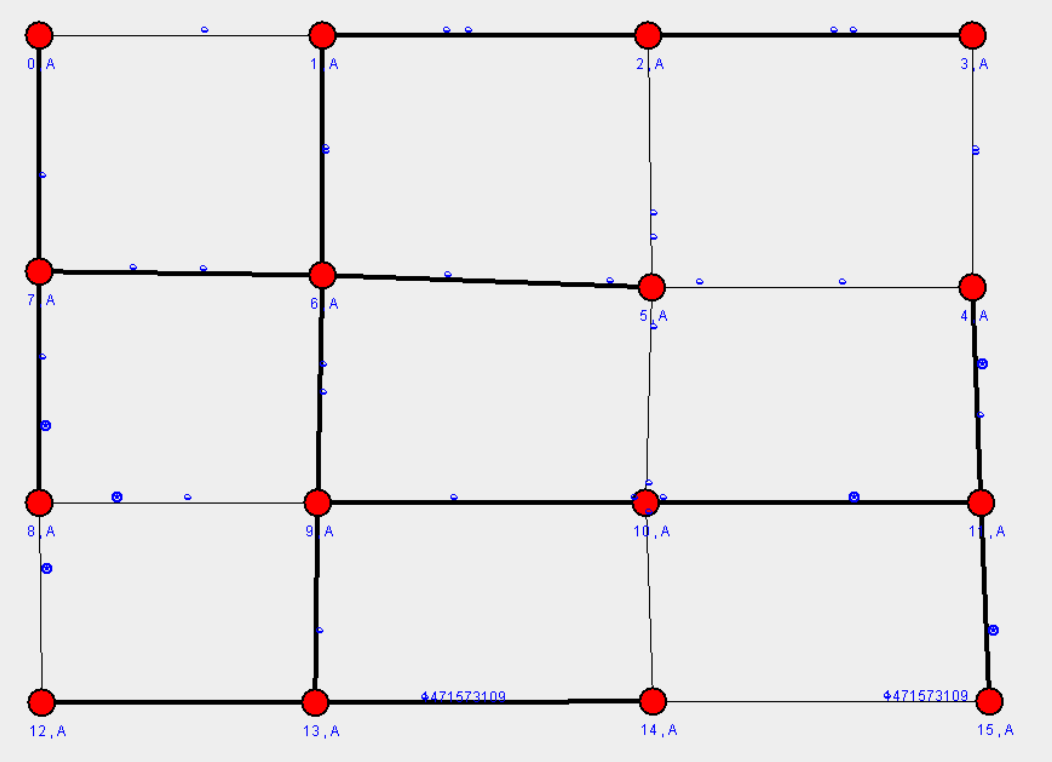


Figure 17 - Résultat arbre recouvrant étoile fermée

2. Election d'un leader dans un arbre

A. Rappel des règles

On suppose que chaque nœud connait le nombre de voisins actifs qu'il possède.

- Si le centre est à l'état « N », il demande à tous ses voisins aussi à l'état « N » et qui ont un seul voisin actif de passer à l'état « F » (le nœud est éliminé).
- Si le centre est à l'état « N » et qu'il possède aucun voisin actif alors il est élu, il passe à l'état « E ».

B. Mise en place

Pour cette algorithme nous devons pouvoir être en capacité de savoir combien de voisin à l'état « N » compte un nœud. Pour ceci nous mettons en place en plus un compteur, mis à jour à chaque synchronisation.

```java
@Override
protected void beforeStart() {
    // chaque noeud sauvegarde son état dans la variable "label"
    //   remarque: tous les noeuds sont à l'état "N" au début
    setLocalProperty("label", vertex.getLabel());
    // on met en place un compteur de voisin N
    setLocalProperty("nbrVoisinN", vertex.getDegree());
}
```

```java
@Override
protected void onStarCenter() {

    // si le centre de l'étoile est "N"
    if (getLocalProperty("label").equals("N")) {
        // true si le noeud à au moins un noeud voisin à "N"
        boolean voisin = false;
        // on check l'état de ses voisins
        for (int i=0; i<getActiveDoors().size(); i++) {
            int numPort = getActiveDoors().get(i);
            // si le voisin est à l'état "N"
            if (getNeighborProperty(numPort, "label").equals("N")) {
                // on a trouvé un voisin à "N"
                voisin = true;
                // si ce voisin en question n'a qu'un seul voisin "N"
                if ((int) getNeighborProperty(numPort, "nbrVoisinN") == 1) {
                    // il pass à l'état "F", il n'est pas élu
                    setNeighborProperty(numPort, "label", "F");
                }
            }
        }
        // si le noeud n'a aucun voisin "N"
        if(!voisin) {
            // le noeud est élu et passe à "E"
            setLocalProperty("label", "E");
        }
    }
```

C. Exemple

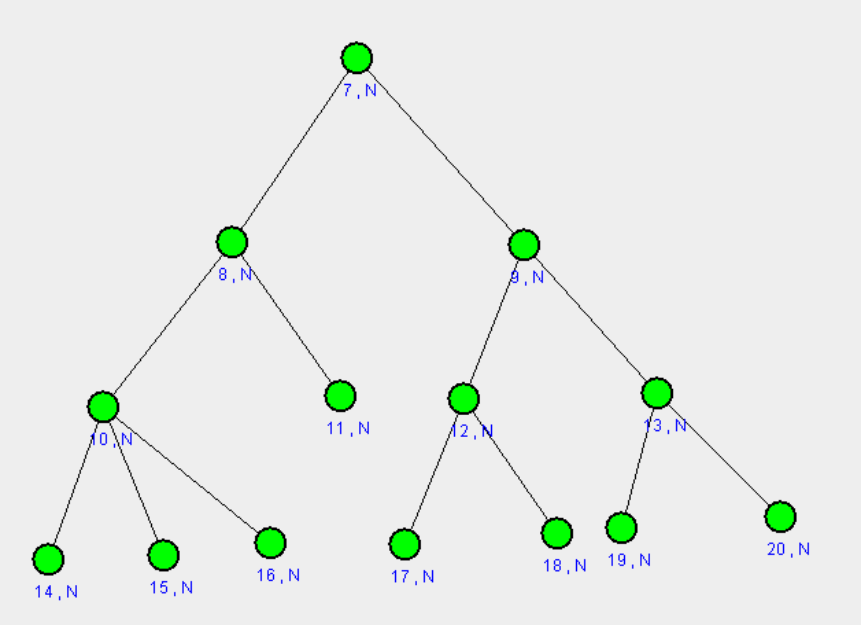


Figure 18 - Graphe de base

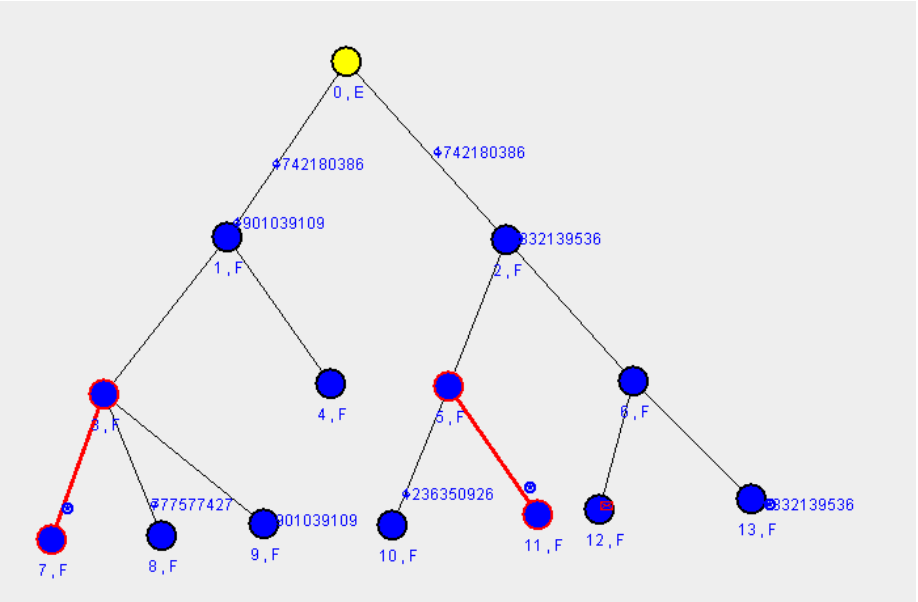


Figure 19 - Résultat élection leader étoile fermée

3. Détection de la terminaison locale

A. Pour l'arbre recouvrant

Dans l'algorithme de l'arbre recouvrant, un nœud peut s'arrêter si plus aucun de ses voisin est à l'état « N ». Pour vérifier ceci on ajoute simplement une variable de type *Boolean*, à chaque fois qu'un nœud est au centre de l'étoile, si aucun voisin « N » n'est détecté alors on appelle la méthode *localTermination*() et le nœud s'arrête. Si tous les nœuds sont

arrêtés alors la simulation prend fin. Plus aucun message n'est envoyé, la simulation est terminée.

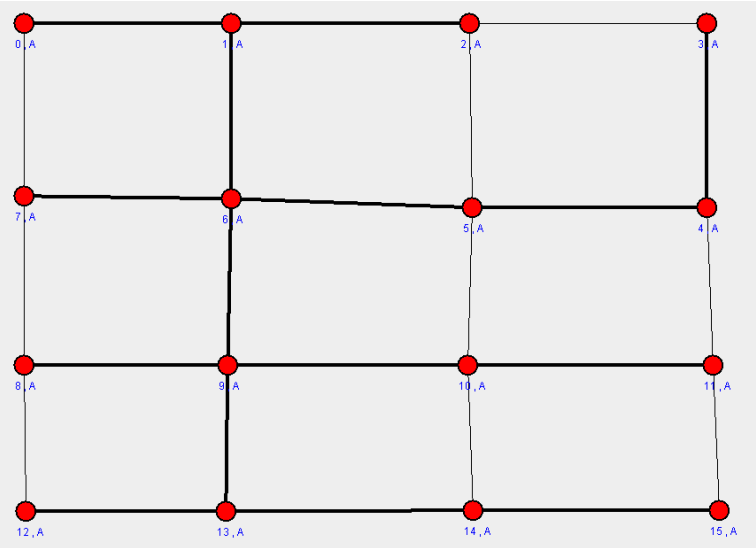


Figure 20 - simulation terminée

B. Election leader

Dans le cas de l'algorithme d'élection d'un leader, un nœud peut s'arrêter lorsqu'il passe à l'état « F » ou « E », c'est-à-dire non élu ou élu. Le centre de l'étoile peut demander à son voisin de changer d'état pour passer à l'état « F » mais ne peux pas lui demander de s'arrêter. Le nœud à l'état « F » s'arrêtera seulement quand il sera centre de l'étoile et qu'il appliquera les règles.

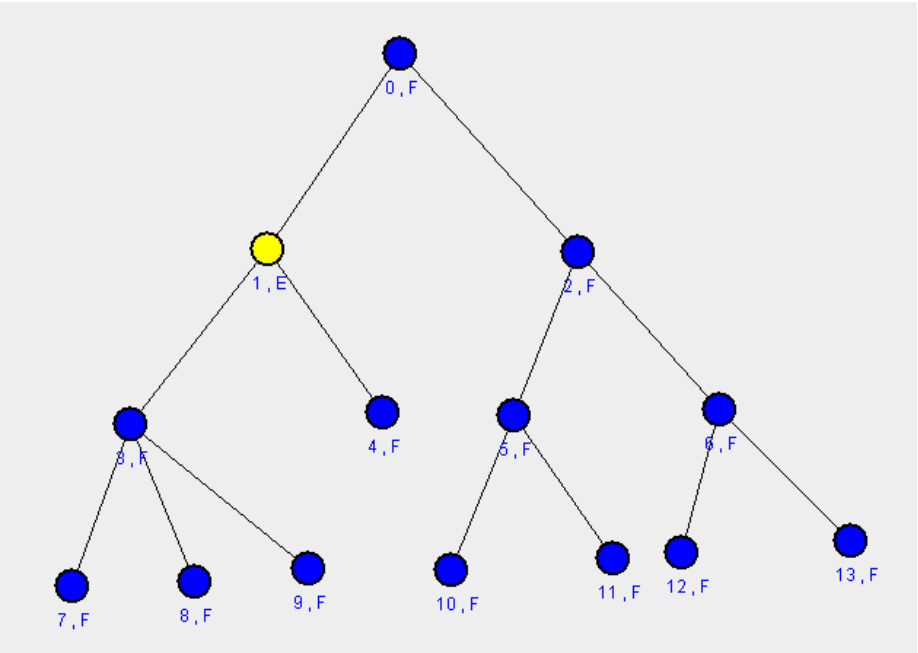


Figure 21- Simulation terminée

Conclusion

Au cours de ces travaux pratiques nous avons pu étudier et manipuler différents algorithmes et différents types de synchronisation. Cela nous a permis de nous familiariser encore un peu plus avec les notions vues en cours et aussi découvrir l'outils « ViSiDia ».