Université de Bourgogne

TP Systèmes distribués

Auteurs :
Benjamin MILHET
Marcelo LOPES

Professeurs :
M. ABDOU
M. TOSSA

benjamin_milhet@etu.u-bourgogne.fr
marcelo_lopes-castanheira@etu.u-bourgogne.fr

2022 - 2023

Table des matières

Introduction

Lors de ce travail pratique, nous étudierons et exploiterons différents algorithmes de systèmes distribués que nous avons vus en classe. Pour cela nous utiliserons le langage java avec le logiciel ViSiDiA. Le logiciel ViSiDiA permet de visualiser la simulation d'algorithmes distribués, à l'aide de règles de réécriture de graphes, d'agents mobiles ou de capteurs mobiles. Nous verrons lors de ce rapport, l'algorithme d'arbre recouvrant, d'élection d'un leader dans un arbre ou encore du calcul d'ordre dans un graphe et ceci avec plusieurs type de synchronisation comme étoile ouverte, étoile fermée ou encore le handshake.

Lien du dépôt Github

TP1 - Découverte de l'outil Visidia

2.1 Première simulation

Pour notre première simulation, nous allons établir nos règles de réécriture grâce à l'interface visuelle du logiciel Visidia.

2.2 Premier algorithme : calcul d'un arbre recouvrant

Nous allons maintenant créer notre première algorithme de calcul d'un arbre recouvrant en Java. Notre programme hérite de la classe "LC0_Algorithm" et nous heritons donc de 4 fonctions obligatoires qui sont :

- getClone() : Retourne une instance de l'objet en cours
- getDescription() : Retourne la description de mon algorithme avec les différentes règles.
- beforeStart() : Fonction permettant d'initialiser nos noeuds, elle sera appeler une seul fois à l'initialisation de notre simulation.
- onStarCenter() : Algorithme appelé lorsque notre noeud est le centre de l'action.

TP2 - Implémentation d'algorithme de base

Dans ce TP nous allons effectuer un algorithme qui va nous permettre d'effectuer un arbre recouvrant en utilisant un parcours en profondeur puis dans un second temps nous allons modifier cet algorithme pour qu'il puisse nous retourner l'ordre du graphe. Un arbre recouvrant est un sous-graphe d'un graphe qui couvre tous les sommets sans former de cycle. Aussi un parcours en profondeur est une méthode pour explorer un graphe en visitant tous les sommets de manière systématique, en commençant par un sommet de départ et en explorant le plus loin possible avant de revenir en arrière.

3.1 Calcul d'un arbre recouvrant en utilisant un parcours en profondeur

3.1.1 Règle de réécriture de l'élection d'un leader dans un arbre

- Règle 1 : N—A —> A—M
- Règle 2 : A—M —> F—A
- Règle interdite : A—N

  Aussi 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" peut 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.

3.1.2 Algorithme

3.1.3 Fonction beforeStart()

```
setLocalProperty("label", vertex.getLabel());
setLocalProperty("father", this.etat);

this.voisins = new String[getArity()];

for (int i = 0 ; i < getArity() ; i++) {
    this.voisins[i] = "N";
}
```

Dans un premier temps nous allons créer deux attributs privés qui nous permettront de connaître le port de notre père ainsi que d'un tableau nous permettant de connaître les états de nos voisins à un instant T.

Par la suite dans la méthode "beforeStart()" nous allons définir 2 propriété locale, une pour connaître l'état de notre nœud et l'autre pour déterminer le port de notre père qui est pour l'instant à -1. Enfin, la boucle for initialise un tableau voisin avec la valeur "N" pour chaque voisin du sommet actuellement traité. Pour connaître le nombre de voisin que nous avons, nous utilisons la méthode "getArity()".

3.1.4 Fonction onStarCenter()

```
this.voisins[neighborDoor] = getNeighborProperty("label").toString();

if (getLocalProperty("label").equals("N") && getNeighborProperty("label").equals("A")) {
    setNeighborProperty("label", "M");
    setLocalProperty("label", "A");
    setLocalProperty("father", neighborDoor);
    setDoorState(new MarkedState(true), neighborDoor);
    this.voisins[neighborDoor] = "M";
} else if (getLocalProperty("label").equals("A") && getNeighborProperty("label").equals("M")
     && neighborDoor == (int) getLocalProperty("father") && this.getNbVoisin("N") == 0) {
    setNeighborProperty("label", "A");
    setLocalProperty("label", "F");
    this.voisins[neighborDoor] = "A";

}
```

Ensuite avec la méthode onStarCenter() nous effectuons les 2 règles pour avoir notre arbre recouvrant.

La ligne this.voisins[neighborDoor] = getNeighborProperty("label").toString(), met à jour la propriété « voisins » pour le voisin de la porte neighborDoor.

Les deux premières conditions vérifient respectivement si la première règle est respectée ou si c'est la seconde est respecté avec le contexte interdit. Si la première condition est respecté alors nous appliquons règles et nous mettons a jour la propriété « father » à jour pour savoir le port de notre père. Aussi nous rajoutons l'état de notre voisin dans le tableau « voisins ». Si la deuxième condition est vérifiée alors nous appliquons la seconde règle et nous mettons l'état du voisin a jour dans le tableau « voisins ».

La méthode "getNbVoisin(String lettre)" est une méthode qui retourne le nombre de voisins qui contienne l'état « lettre ».

3.2 Calcul de l'ordre du graphe

Pour connaître l'ordre de notre graphe il suffit de reprendre la code précédent et de rajouter une propriété local qui correspond a un compteur et qui s'affiche pour chaque nœud grâce la méthode "putProperty()". Lorsque nous passons un nœud a l'état F, le nœud envoie son compteur au nœud auquel il s'est synchronisé et l'additionne au compteur de celui-ci. Et donc nous obtenons le nombre l'ordre du graphe sur la second nœud qui est synchronisé au nœud de départ.

3.3 Rendu final

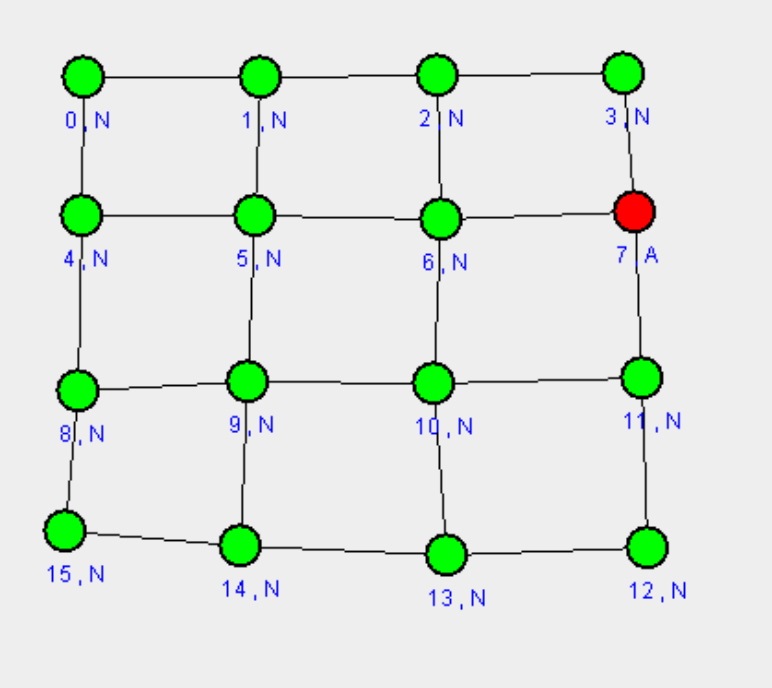

FIGURE 3.1 – Rendu de notre graphe 4x4 avant la simulation

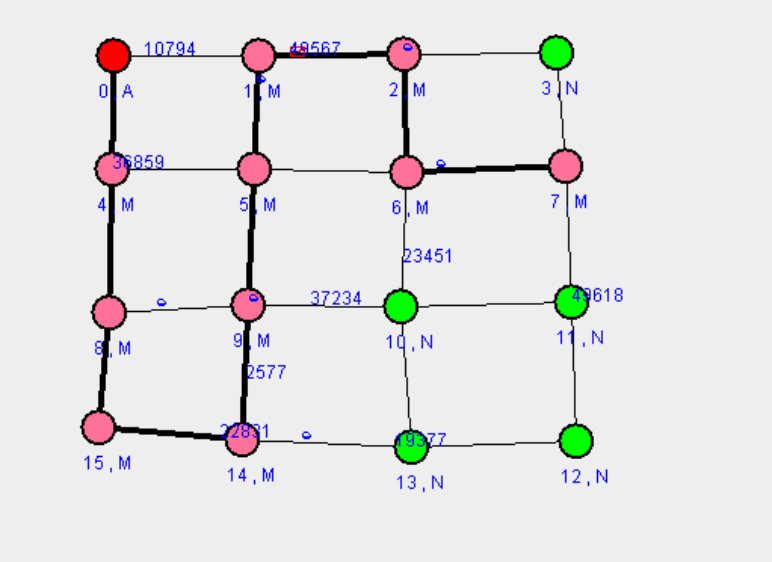

FIGURE 3.2 – Rendu de notre graphe 4x4 pendant la simulation

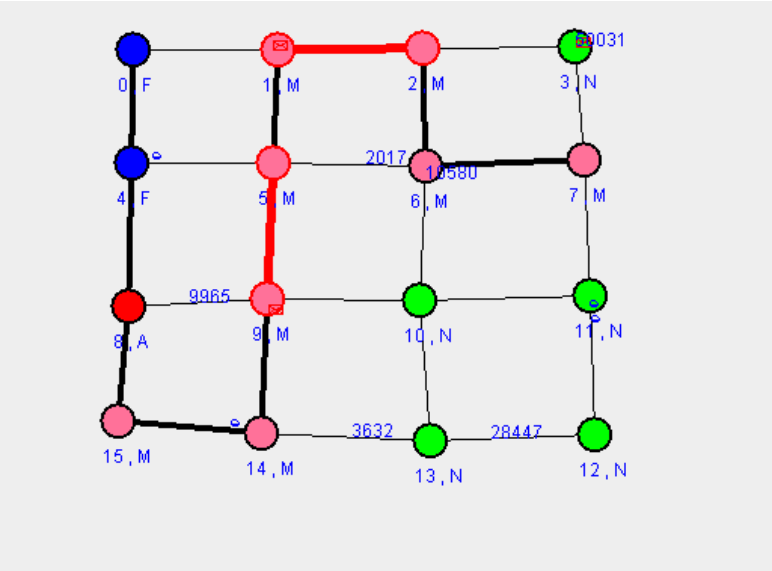

FIGURE 3.3 – Rendu de notre graphe 4x4 pendant la simulation

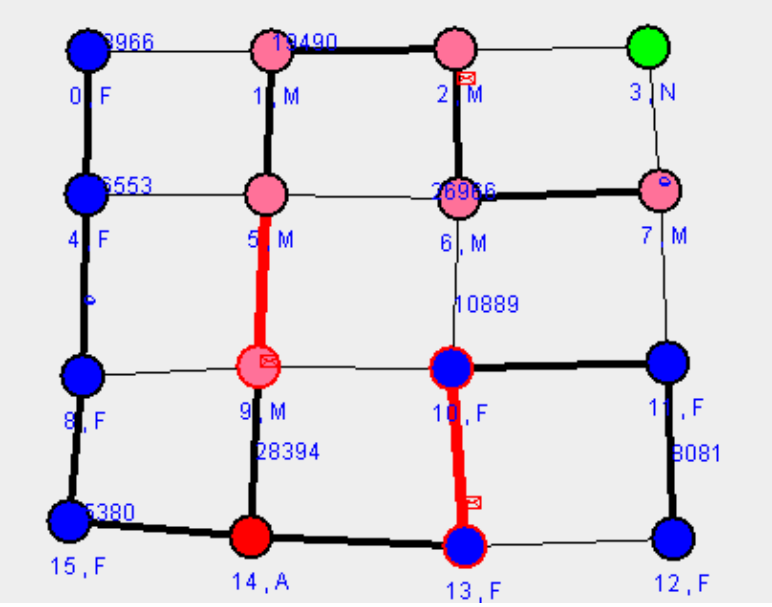

FIGURE 3.4 – Rendu de notre graphe 4x4 pendant la simulation

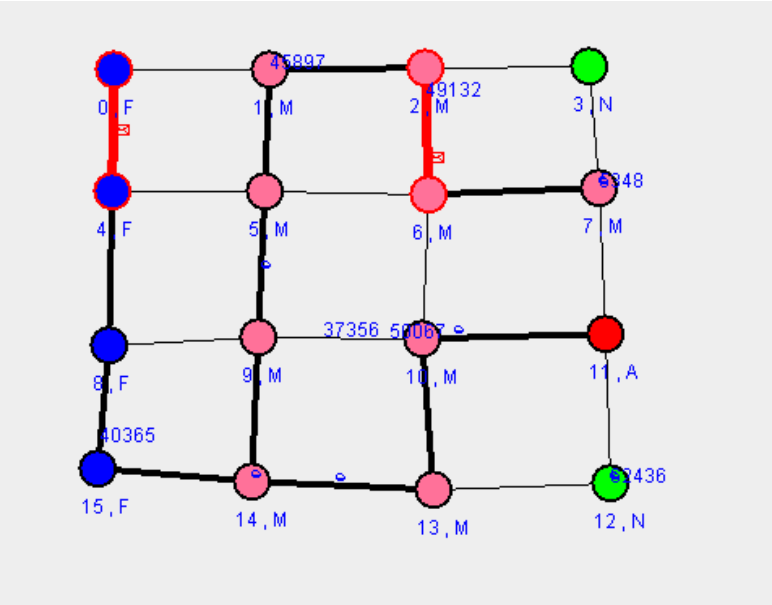

FIGURE 3.5 – Rendu de notre graphe 4x4 pendant la simulation

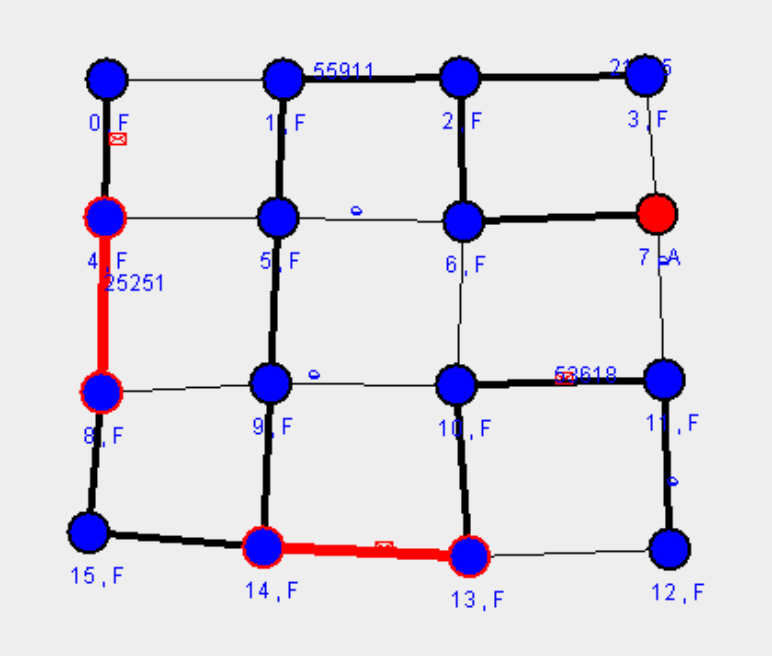

FIGURE 3.6 – Rendu final de notre graphe 4x4 avec notre algorithme d'arbre recouvrant

TP3 - Élection d'un leader dans un arbre

4.1 Règle de réécriture de l'élection d'un leader dans un arbre

— Tous les sommets de départ sont à l'état neutre N
— Règle 1 : N(1)—N(x) —> F(0)—N(x-1)
— Règle 2 : N(1)—N(1) —> E(0)—F(0)
— avec x > 1

Pour qu'un noeud soit élu leader, il faut que 2 noeuds voisins N n'est chacun que des voisins feuilles pour pouvoir élire le dernier noeud qui sera le leader de notre arbre.

4.2 Algorithme

L'objectif de cet algorithme est d'élire un leader parmi l'ensemble des points. A la fin de la simulation de notre algorithme d'élection d'un leader dans un arbre, nous aurons l'ensemble de nos noeuds qui seront à l'état F (Feuille) sauf un, et un seul qui sera à l'état E et qui correspondra à notre leader.

4.2.1 Fonction beforeStart()

```
setLocalProperty("label", vertex.getLabel());
setLocalProperty("x", getArity());

putProperty("Affichage", getArity(), SimulationConstants.PropertyStatus.DISPLAYED);
```

On commence par initialiser 2 données qui est le nom de label et le nombre de voisin ayant un label N. Avec la fonction "putProperty", on affiche le nombre de voisin N à coté de noeud en question.

4.2.2 Fonction onStarCenter()

```
if (getLocalProperty("label").equals("N") && getNeighborProperty("label").equals("N")) {
           int nbVoisinsLocal = (int) getLocalProperty("x");
           int nbVoisinsNeighbor = (int) getNeighborProperty("x");

           if (nbVoisinsLocal == 1 && nbVoisinsNeighbor > 1) {
               setLocalProperty("label", "F");
               setLocalProperty("x", 0);

               setNeighborProperty("label", "N");
               setNeighborProperty("x", nbVoisinsNeighbor - 1);

               nbVoisinsLocal = (int) getLocalProperty("x");
               putProperty("Affichage", nbVoisinsLocal, SimulationConstants.PropertyStatus.DISPLAYED);

           } else if (nbVoisinsLocal == 1 && nbVoisinsNeighbor == 1) {
               setLocalProperty("label", "E");
               setLocalProperty("x", 0);

               setNeighborProperty("label", "F");
               setNeighborProperty("x", 0);

               nbVoisinsLocal = (int) getLocalProperty("x");
               putProperty("Affichage", nbVoisinsLocal, SimulationConstants.PropertyStatus.DISPLAYED);

           }
       }
```

On commence par regarder que les noeuds de la liaison soit bien des noeuds "N". On commence par traiter les noeuds qui sont des sommets de l'arbre et qui traite avec des noeuds voisins qui ont plusieurs sommets. On est donc dans le cas d'une feuille, on change alors la valeurs de l'étiquette en "F". Si le noeud local a un seul voisin et que ce voisin à lui aussi un seul voisin, alors nous sommes à la dernière itération et nous pouvons élire le leader de notre arbre.

4.3 Rendu final

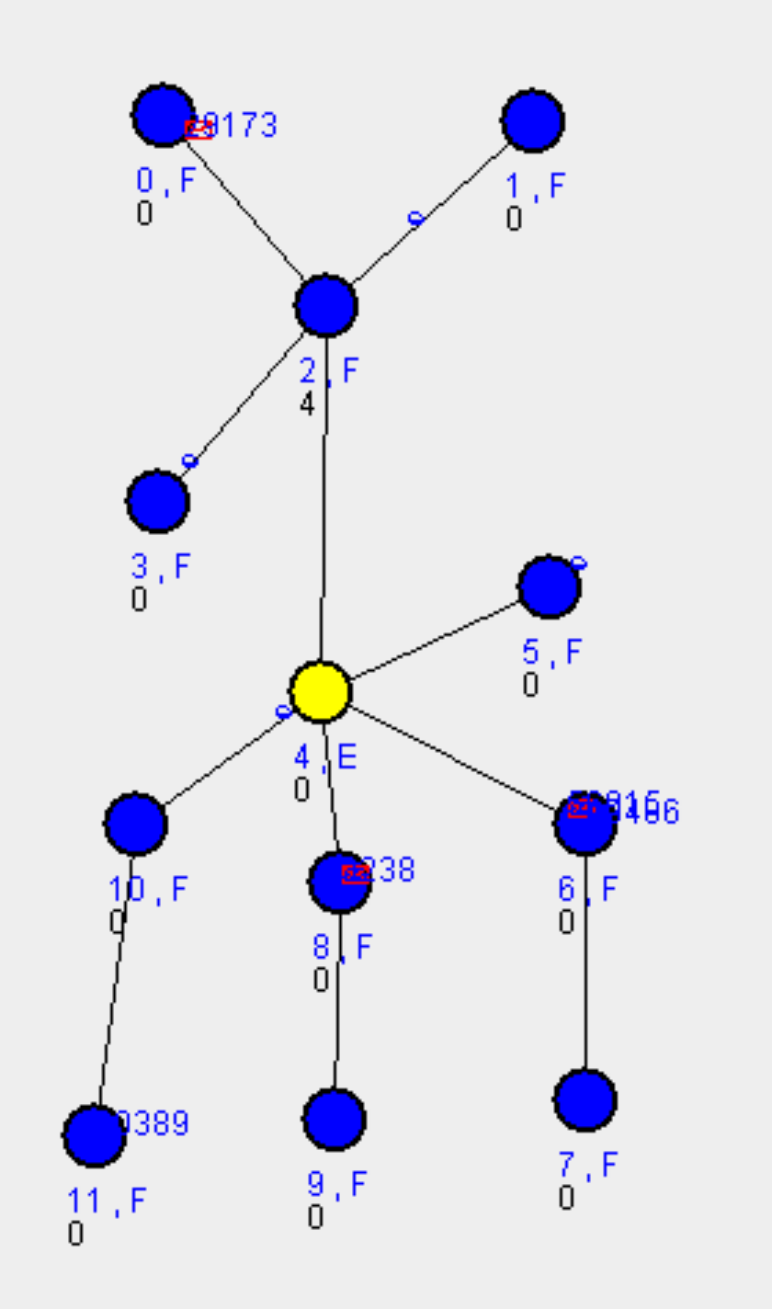

FIGURE 4.1 – Rendu final de l'élection d'un leader dans un arbre

On remarque que bien tous les noeuds sont à l'état "F" final sauf un qui est à l'état "E", ce qui correspond au leader de notre arbre.

TP4 - Etoiles ouvertes

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 centre n'a pas la possibilité de modifier les états de ses voisins.

5.1 Arbre recouvrant

Pour effectuer cette règle nous avons crée une méthode "getVoisinA()" qui retourne un voisin quelconque de notre nœud qui est à l'état A. Pour faire cette méthode nous avons utilisé la méthode "getActiveDoors()" qui renvoie une ArrayList contenant les numéros de ports sur lesquels les voisins d'un nœud sont connectés. Cette méthode provient de la classe "LC1_Algorithm" d'où le fait que nous héritons de cette classe.

Par la suite il suffit dans un premier temps de définir la propriété local « label » (dans la méthode "beforeStart()") avec l'état de notre nœud, puis par la suite dans la méthode "onStarCenter()" de regarder si notre nœud est à l'état N puis si cela est le cas nous marquons la synchronisation si notre nœud a comme voisin un nœud à l'état A.

On peut remarquer qu'avec cette algorithme qui utilise la synchronisation de type étoile ouverte, nous obtenons l'arbre recouvrant beaucoup plus vite qu'avec celle vue lors du TP2.

5.2 Election d'un leader dans un arbre

Avant de présenter le programme voici les règles de calcul que nous devons respecter :

- Si le centre de l'étoile est à l'état "N" et qu'il ne possède qu'un seul voisin "N" et ce centre passe à l'état "F". Car dans ce cas il représente une feuille de notre arbre.

- Si le centre est à l'état "N" et qu'il n'a aucun autre voisin aussi à l'état "N" alors ce centre devient le leader et passe à l'état "E".

5.2.1 Algorithme

5.2.1.1 Fonction beforeStart()

```
setLocalProperty("label", vertex.getLabel());
```

5.2.1.2 Fonction onStarCenter()

```
@Override
protected void onStarCenter() {
    if (getLocalProperty("label").equals("N")) {
        int port = getVoisinA();

        if (port != -1) {
            setLocalProperty("label", "A");
            setDoorState(new MarkedState(true), port);
        }
    }
}

private int getVoisinA(){
    int res = -1;

    for (int i = 0 ; i < getActiveDoors().size() ; i++) {
        int tmp = getActiveDoors().get(i);

        if (getNeighborProperty(tmp, "label").equals("A")) {
            res = tmp;
        }
    }

    return res;
}
```

Comme pour les autres programmes dans la méthode "beforeStart()" le nœud va sauvegarder son état dans le variable "label". Par la suite dans la méthode "onStarCenter()" nous allons regarder tout d'abord si notre nœud est à l'état « N » puis pour vérifier que notre nœud n'a qu'un seul voisin à l'état "N" nous allons compter le nombre de nœud avec une boucle for en s'aidant de la méthode "getActiveDoors()" qui permet de retourner un tableau contenant les ports des voisins de notre nœud. Si le nœud n'a qu'un seul voisin à l'état « N » alors c'est une feuille est donc nous changeons son état pour le passer à l'état "F" sinon s'il n'a aucun voisin à l'état "N" alors c'est le leader et nous passons son "label" à l'état "E".

5.3 Rendu final

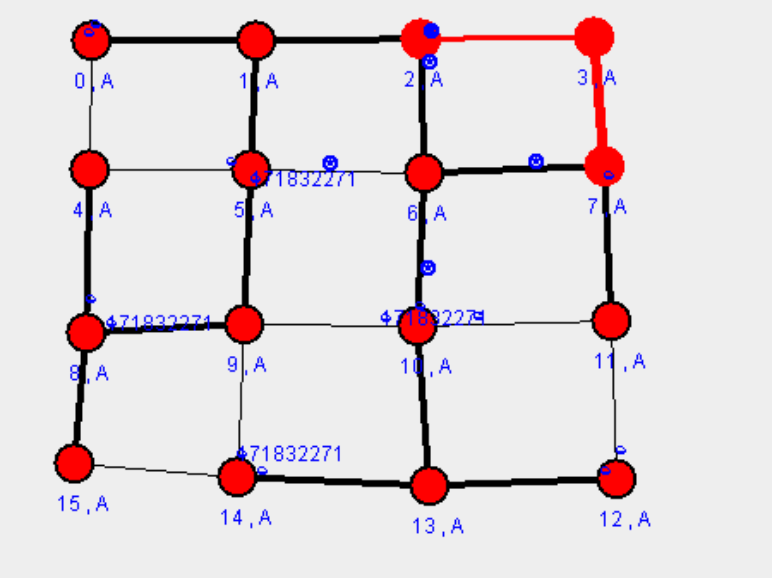

FIGURE 5.1 – Rendu final de notre graphe 4x4 avec notre algorithme d'étoile ouverte

TP5 - Etoiles fermées

6.1 Arbre recouvrant

Le but de ce TP est d'écrire des algorithmes qui s'appuient sur une synchronisation locale de type étoile fermée. 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

6.1.1 Règles

Au départ, tous les sommets du graphes doivent être à l'état "N" sauf un seul qui sera à l'état "A" qui initiera le calcul.

Si le centre de l'étoile est à l'état "A" et qu'il possède des voisins à l'état "N", il demande à ces derniers de passer à l'état "A" et marque les liens qui le connectent à ces voisins.

Afin de pouvoir connaître les ports sur lesquels les voisins sont connectés, nous avons utilisé la fonction "getActiveDoors()".

6.1.2 Algorithme

6.1.2.1 Fonction beforeStart()

```
setLocalProperty("label", vertex.getLabel());
setLocalProperty("isDone", false);
```

Dans cette fonction, nous initialisons l'ensemble des noeuds avec leur étiquette respective. La variable "isDone" nous est utile dans la détection de la terminaison locale afin d'arrêter un noeud plus tard dans le programme.

6.1.2.2 Fonction onStarCenter()

```
if ((boolean) getLocalProperty("isDone")) localTermination();

    if (getLocalProperty("label").equals("A")) {

        for (int i = 0 ; i < getActiveDoors().size() ; i++) {
            int tmp = getActiveDoors().get(i);

            if (getNeighborProperty(tmp, "label").equals("N")) {
                setNeighborProperty(tmp,"label", "A");
                setDoorState(new MarkedState(true), tmp);
            }

            setLocalProperty("isDone", true);
        }

    }
```

On commence par vérifier si la variable "isDone" est à True ou non. Si oui, on arrête le noeud avec la fonction "localTermination()". Ensuite, il nous suffit de vérifier si le noeud local est à l'état "A". On parcours l'ensemble des voisins du noeud à l'état "A", pour tous ses voisins à l'état "N", on change son état à l'état "A" et on marque l'arrête. Dès que l'on tombe une fois sur un noeud à l'état "A", nous changeons la valeur de la variable "isdone" à True parce que nous n'aurons plus besoin de ce noeud pour la suite du programme. Au début, nous avions directement appelé la fonction "localTermination()" au lieu d'utiliser une variable temporaire, cependant cela posait problème et l'algorithme s'arrêtait avant la fin.

6.1.3 Rendu final

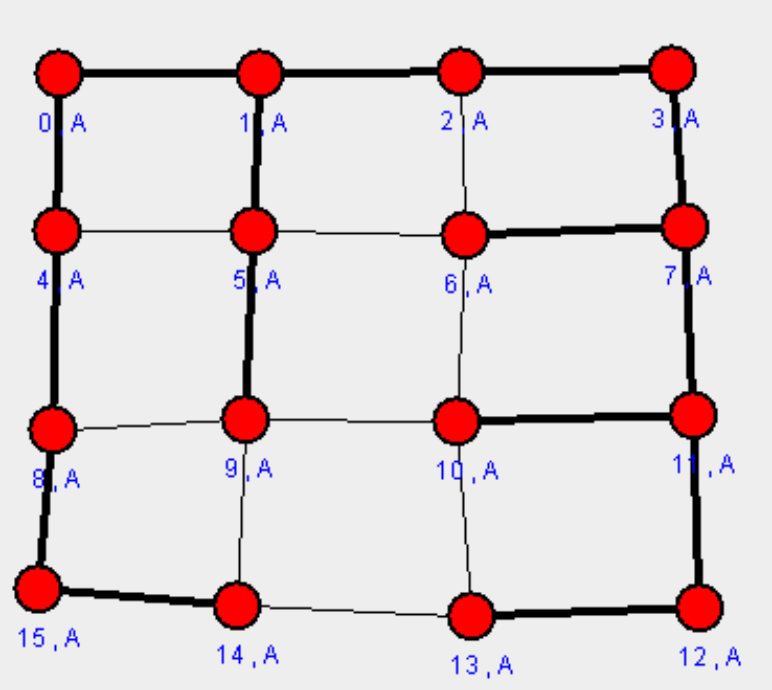

FIGURE 6.1 – Rendu final du calcul d'un arbre couvrant par l'utilisation de l'étoile fermée

On observe bien qu'une fois notre algorithme terminé, tous les noeuds du graphe sont à l'état "A" et nous obtenons un arbre avec les arrêtes marquées.

6.2 Election d'un leader dans un arbre

6.2.1 Algorithme

6.2.1.1 Fonction beforeStart()

```
setLocalProperty("label", vertex.getLabel());
setLocalProperty("x", getArity());
setLocalProperty("isDone", false);
```

Dans cette fonction, nous initialisons l'ensemble des noeuds avec leur étiquette respective. La variable "isDone" nous est utile dans la détection de la terminaison locale afin d'arrêter un noeud plus tard dans le programme. Nous ajoutons en plus une variable "x" qui contient le nombre de voisin du noeud.

6.2.1.2 Fonction onStarCenter()

```
if ((boolean) getLocalProperty("isDone")) localTermination();

    if (getLocalProperty("label").equals("N")) {

        for (int i = 0 ; i < getActiveDoors().size() ; i++) {
            int tmp = getActiveDoors().get(i);

            if (getNeighborProperty(tmp, "label").equals("N")) {
                if ((int) getNeighborProperty(tmp, "x") == 1){
                    setNeighborProperty(tmp, "label", "F");
                    setLocalProperty("x", (int) getLocalProperty("x")-1);
                    setNeighborProperty(tmp, "isDone", true);
                }
            }
            if ((int) getLocalProperty("x") == 0) {
                setLocalProperty("label", "E");
                setLocalProperty("isDone", true);
            }
        }
    }
```

Dans cette algorithme, on commence par regarder que le noeud local est bien un noeud à l'état "N". Ensuite, nous parcourons l'ensemble des ses voisins et pour chaque voisin à l'état "N", nous regardons la valeur de "x", qui correspond a son nombre de voisin à l'état "N". S'il n'a qu'un seul voisin à l'état "N", alors c'est une feuille et nous changeons son état à "F". Si le noeud local ne possède aucun voisin "N", alors nous en déduisons que c'est le leader de notre arbre et nous changeons son état à "E".

6.2.2 Rendu final

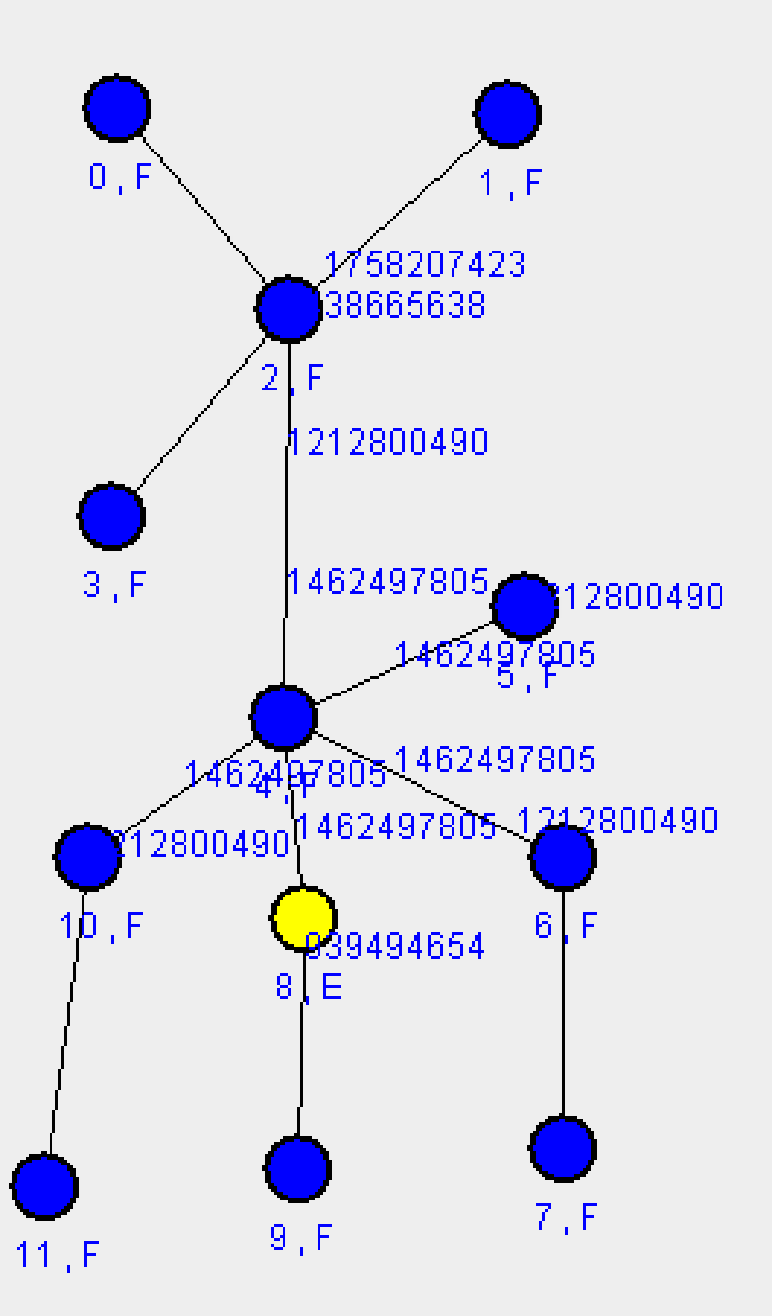

FIGURE 6.2 – Rendu final de l'élection d'un leader dans un arbre

6.3 Détection de la terminaison locale

La détection local s'effectue grâce à mes variables "isDone" stocker dans le local property de chacun des noeuds. Dès que cette variable passe a True, nous appelons la fonction "localTermination()" qui permet d'arrêter un noeud.

Conclusion

Pour conclure, grâce à ces travaux pratiques nous avons pu manipuler et mieux comprendre différents algorithmes et différents types de synchronisation. Aussi, ces travaux pratique nous, on permet de découvrir le logiciel ViSiDia.