Blain Hugo ESIREM

4A ILC

# Compte Rendu TP

Compte rendu des TPs de Systèmes distribués

# Table des matières

# Introduction :

Dans ces deux travaux pratiques nous utiliserons la librairie « ViSiDia » et le langage de programmation JAVA afin de simuler et programmer des algorithmes élémentaires ou applicables à la gestion distribuée des ressources. Nous verrons notamment l'algorithme de 3-Coloration d'un anneau, les notions de stabilité et de porté de terminaison, un algorithme auto-stabilisant et enfin comment pallier la perte d'information locale.

Les codes sources sont disponibles sur le dépôt suivant :

https://github.com/HugoBlain/TP_Gestion_Distribuee_Des_Ressources

# TP n°1 : Allocation des ressources & gestion de la terminaison/stabilité

## 1. Allocation distribuée des ressources, 3-Coloration d'un anneau

Soit un anneau composé d'au moins trois nœuds. On souhaite attribuer des couleurs (parmi 3 possibles) de telle sorte que deux nœuds voisins aient des couleurs différentes. La coloration des sommets est quelque chose qui s'apparente à l'allocation des ressources et trouve de nombreuses applications en réseaux ou gestion des systèmes.

### A. Règles de réécriture

Pour réaliser la 3-Coloration, on peut procéder comme suit :

- Si un nœud a la même couleur que ses deux voisins, alors il choisit une couleur différente.
- Si un nœud a la même couleur qu'un seul de ses voisins, il choisit une autre couleur en évitant, celle de son second voisin.

### B. Mise en place

Pour cet exercice j'ai décidé d'utiliser une synchronisation de type étoile ouverte et donc la classe *LC1_Algorithm* de ViSiDia. En effet, nous devons pouvoir vérifier l'état de tous les voisins lors d'une synchronisation et pas seulement de celui avec lequel elle a lieux mais nous n'avons pas besoin de modifier leur état.

Au lancement de l'algorithme, j'ai fait le choix d'attribuer aléatoirement à chacun des sommets une couleur parmi les 3 possibles. La couleur du nœud est stockée dans une *localProperty* nommée *code Couleur* et peut prendre une valeur de 0 à 2. Aussi, pour que visuellement le nœud change de couleur durant la simulation, on associe chacun de ces codes-couleurs à un état, c'est-à-dire que quand le code couleur d'un nœud change, son état aussi. A noter qu'on aurait aussi pu commencer avec aucun des nœuds avec une couleur, c'est juste plus long.

| Code couleur | Etat | Couleur |
|---|---|---|
| 0 | « N » | Vert |
| 1 | « A » | Rouge |
| 2 | « F » | Bleu |

```
void changerCouleur(int codeCouleur) {
    setLocalProperty("codeCouleur", codeCouleur);
    switch(codeCouleur) {
        // vert
        case 0:
            setLocalProperty("label", "N");
            break;
        // rouge
        case 1:
            setLocalProperty("label", "A");
            break;
        // bleu
        case 2:
            setLocalProperty("label", "F");
            break;
        default:
            System.out.println("Erreur switch");
    }
}
```

Puisque nous allons devoir de nombreuses fois changer la couleur d'un nœud, j'ai choisi de créer une fonction à cet effet pour simplifier le code. Change le code couleur et l'état du nœud en fonction de l'entier qu'on lui passe en paramètre.

```
@Override
protected void beforeStart() {
    // on tire une couleur aléatoire pour chacun des noeuds au départ
    int aleatoire = new Random().nextInt(3);
    changerCouleur(aleatoire);
}
```

Comme vu juste avant, au lancement, chacun des nœuds se voit attribuer une couleur au hasard parmi les trois possibles.

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

    // ports des voisins
    int port0 = getActiveDoors().get(0);
    int port1 = getActiveDoors().get(1);

    Object etat = getLocalProperty("label");

    // règle n°1 si deux voisins de la même couleur que le noeud
    if (getNeighborProperty(port0, "label").equals(etat) && getNeighborProperty(port1, "label").equals(etat)) {
        // on change la couleur avec une des deux autres restantes
        int couleurActuelle = (int) getLocalProperty("codeCouleur");
        couleurActuelle += 1;
        if(couleurActuelle == 3) {
            couleurActuelle = 0;
        }
        changerCouleur(couleurActuelle);
    }

    // règle n°2 partie 1: le voisin 0 a la même couleur
    else if (getNeighborProperty(port0, "label").equals(etat)) {
        int codeVoisinPareil = (int) getNeighborProperty(port0, "codeCouleur");
        int codeVoisinDifferent = (int) getNeighborProperty(port1, "codeCouleur");
        changerCouleur(3-codeVoisinPareil-codeVoisinDifferent);
    }
    // règle n°2 partie 2: le voisin 1 a la même couleur
    else if (getNeighborProperty(port1, "label").equals(etat)) {
        int codeVoisinPareil = (int) getNeighborProperty(port1, "codeCouleur");
        int codeVoisinDifferent = (int) getNeighborProperty(port0, "codeCouleur");
        changerCouleur(3-codeVoisinPareil-codeVoisinDifferent);
    }

}
```

Enfin dans la méthode *onStarCenter()*, on applique les règles de réécriture. A noter que puisque qu'on travaille avec une synchronisation en étoile ouverte et avec un graphe en anneau, chacun des nœuds possède deux voisins et on peut identifier leurs ports avec la méthode *getActiveDoors().*

Si le nœud est de la même couleur que ses deux voisins alors son code couleur est simplement incrémenté de 1 pour passer à la couleur suivante (retour à 0 si on dépasse 2).

Sinon si il est de la même couleur qu'un seul de ses voisins, il doit changer mais en évitant la couleur de son autre voisin. Il ne lui reste plus qu'une seule possibilité parmi les couleurs possibles. Pour la déduire le code de la couleur à affecter on utilise celui les couleurs à éviter en effectuant une soustraction. La situation impose que les 3 nœuds, celui au centre de l'étoile et ses deux voisins, doivent être de couleurs différentes. La somme des 3 doit faire 3 et nous connaissons le code des deux voisins, on peut donc calculer le code couleur que doit utiliser le nœud au centre.

Exemple :

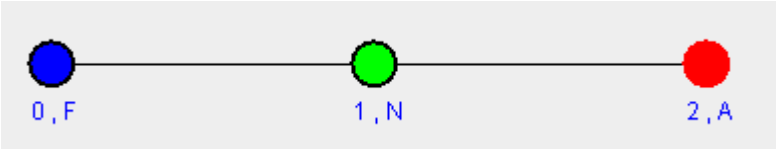


*Figure 1 - Exemple calcul code couleur*

Ici, si le nœud du centre est le nœud du centre de l'étoile, son code doit être :

3 – code bleu – code rouge ⇔ 3 – 2 – 1 = 0 ⇔ 0 qui est bien le code du vert

**C. Exemple**

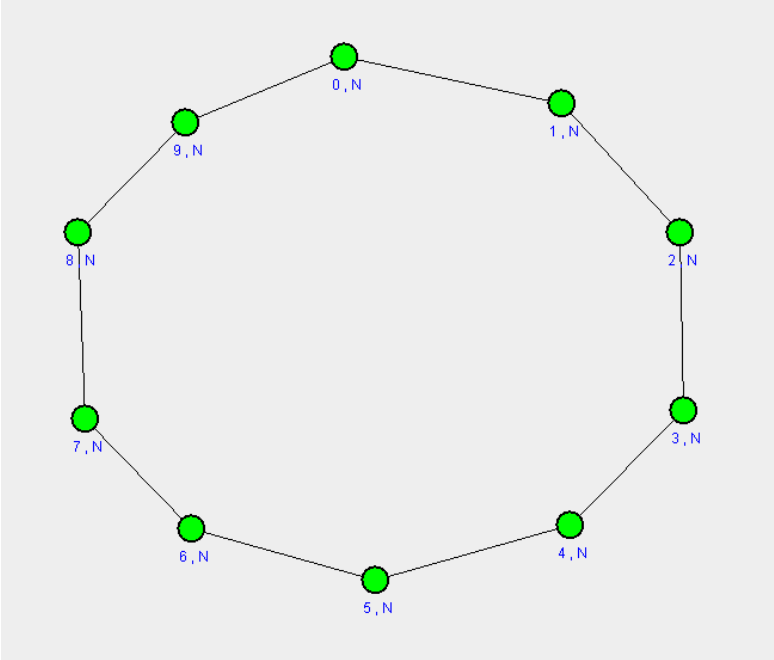


*Figure 2 - Graphe de base*

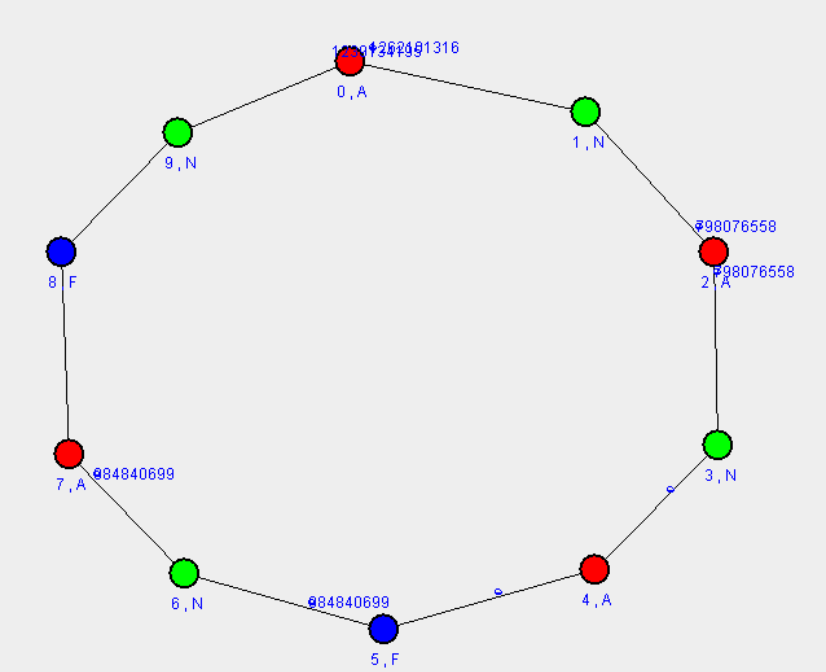


*Figure 3 - Coloration anneau*

## 2. Portée de la terminaison ou de la stabilité, algorithme SSP

Des serveurs se mettent à jour en récupérant des données de leurs voisins. Cette récupération prend fin au bout de n synchronisation n est fixé aléatoirement pour chaque nœud. Lorsqu'un nœud est à jour, il passe à l'état « U ».

### A. Algorithme SSP

L'algorithme SSP utilise deux variables : un booléen que l'on a appelé « termine » et un compteur « a ». Au départ, chaque nœud initialise « termine » à *false* et « a » à -1. Lorsque la condition de terminaison est satisfaite, « termine » passe à *true* et « a » est mis à jour. La nouvelle valeur de « a » est égale au minimum des « a » des voisin +1.

Dans notre cas, on considère qu'un nœud a terminé lorsque le serveur qu'il représente est à jour, c'est-à-dire lorsqu'il a effectué ses n synchronisation. Aussi on peut décrire « a » comme la distance à laquelle le nœud peut affirmer que ses voisins et lui-même sont stables. C'est pour cela qu'on commence avec a = -1, au début le nœud n'a lui-même pas terminé, il ne peut donc même pas affirmer que les nœuds sont stables à une distance 0 (distance 0 signifie qu'il se désigne lui-même).

**B. Mise en place de l'algorithme**

```
public class Portee_Terminaison_TP1 extends LC1_Algorithm {

    // tous les serveurs tire au sort un nombre aléatoire de syncho à faire pour considerer qu'il a terminé
    private int nbr_synchr_a_faire =  1 + new Random().nextInt(5);
    // savoir si le noeud à terminer
    private boolean termine = false;
```

On déclare deux attributs dans chaque nœud, un entier « nbr_synchro_a_faire » qui correspond au nombre de synchronisation que doit faire le nœud pour considérer qu'il a terminé. Cette variable est en fait le n dont nous parlions plus haut et prend une valeur aléatoire entre 1 et 5. Il y a aussi le booléen qui marque si le nœud a terminé ou non, de base il est à *false*.

```
@Override
protected void beforeStart() {
    // distance à laquelle on peut garantir que les serveurs ont terminé
    setLocalProperty("a", -1);
    // affichage
    putProperty("affichageA", "a = "+getLocalProperty("a"), SimulationConstants.PropertyStatus.DISPLAYED);
}
```

```
@Override
protected void onStarCenter() {

    // si le noeud n'a pas terminé
    if(!this.termine) {
        // compteur synchro mis à jour
        this.nbr_synchr_a_faire --;
        // check si le noeud à terminé
        if(this.nbr_synchr_a_faire == 0) {
            this.termine = true;
            setLocalProperty("label", "U");
        }
    }

    // si le noeud a terminé
    else {
        // la variable a prend la valeur la plus petite parmi la sienne et celle de ses voisins
        int mini = (int) getLocalProperty("a");
        for (int i=0; i<getActiveDoors().size(); i++) {
            int numPort = getActiveDoors().get(i);
            if ((int) getNeighborProperty(numPort, "a") < mini) {
                mini = (int) getNeighborProperty(numPort, "a");
            }
        }
        setLocalProperty("a", mini+1);
        // affichage
        putProperty("affichageA", "a = "+getLocalProperty("a"), SimulationConstants.PropertyStatus.DISPLAYED);
    }

}
```

Lors de chaque synchronisation on applique différentes règles selon si le nœud a terminé ou non. Si ce n'est pas le cas, on décrémente le compteur des synchronisations qu'il lui reste à faire. Et sinon, si le nœud a terminé, la variable « a » se met à jour.

## C. Exemple

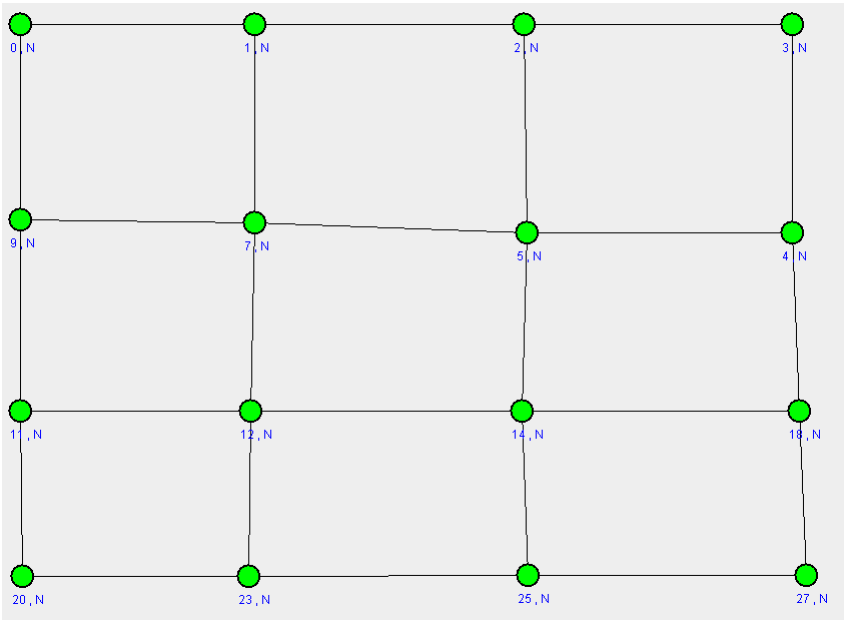


*Figure 4 - Graphe de base*

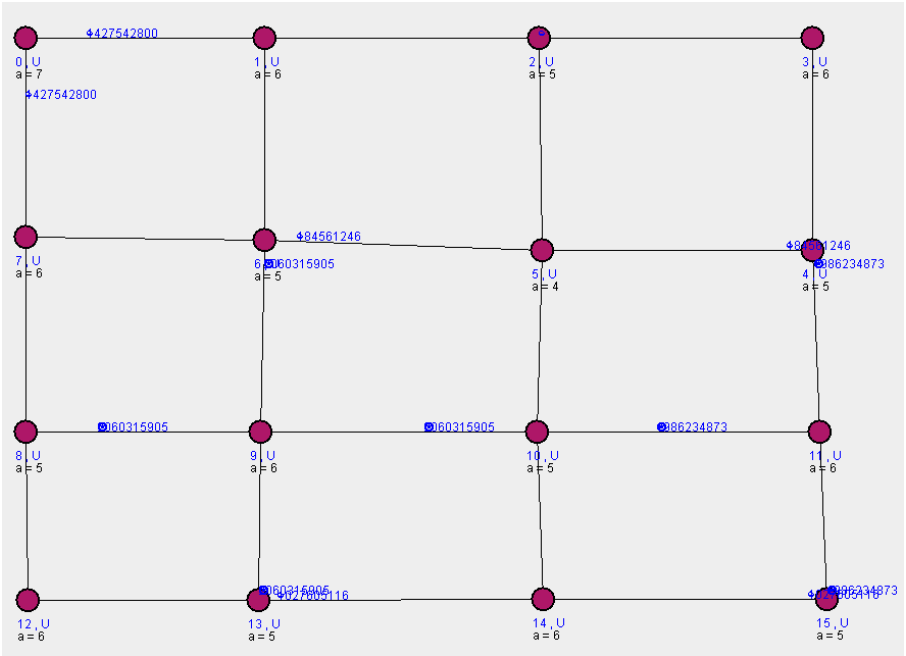


*Figure 5 - Graphe après lancement*

Tel qu'il est actuellement, le programme ne s'arrête pas et le compteur « a » des nœuds peut augmenter à l'infini. Dans l'idéal il faudrait que le nœud s'arrête quand son compteur dépasse le diamètre du graphe cependant, cette valeur est compliquée à calculer.

**2) A votre avis, que ce passerait-il si chaque nœud décidait d'exécuter *localTermination* quand son compteur « a » vaut 5 ?**

Dans un tel cas, de manière générale tous les nœuds s'arrêteraient petit à petit quand il arrive à 5. Cependant cela dépend du graphe et de l'ordre aléatoire des synchronisations, il est possible qu'un nœud soit isolé si son voisin seul passe à 5, alors ce nœud restera à 4.

**3) Que ce passerait-il si un nœud exécutait *globalTermination()* lorsque son compteur passe à 5 ?**

Dans ce cas-là, toujours selon la forme du graphe et du nombre de nœud, les autres nœuds auraient une valeur « a » inférieur à 5 car le celui qui exécute cette méthode est le premier et donc ne laisse pas le temps aux autres d'augmenter.

# TP n°2 : Algorithme auto-stabilisant & pallier la perte d'une information locale

## 1. Algorithme auto-stabilisant

Un algorithme est dit auto-stabilisant si, quel que soit son état initial, il garantit l'atteinte d'un état donné en un nombre d'étapes finies. Ce type d'algorithme se termine correctement malgré les perturbations qui pourraient survenir dans le système.

### A. Description

Soit un anneau composé de n nœuds ou processeurs. On définit un algorithme de changement d'état lorsqu'un nœud possède le jeton décrit comme suit :

- Soit k un entier supérieur ou égale à 0, tel que k > n
- Initialisation : chaque nœud a un état (σ), dont le numéro est choisi aléatoirement entre 0 et k-1
- Possession du jeton :
    - Le nœud 0 (N0) a le jeton si σ0 = σn-1
    - Ni a le jeton si σi est différent de σi-1 avec i > 0
- Calcul :
    - Pour N0 : si σ0 = σn-1 alors σ0 ← σ0 + 1%k
    - Pour Ni : si σi différent σi-1 alors σi ← σi-1

### B. Mise en œuvre

Pour cet algorithme k sera égal à 7, c'est-à-dire que pour l'utiliser il faudra que le graphe soit un anneau avec 6 nœuds au maximum (k > n). Aussi, étant donné que le nœud N0 a un comportement différent des autres, il faudra faire en sorte de pouvoir le différencier. Pour simplifier, au début, tous les nœuds de notre anneau seront à l'état « N » sauf un qui sera à l'état « A », celui-ci sera le nœud N0.

Nous utilisons pour cet algorithme une synchronisation en étoile ouverte, lorsqu'un nœud est centre de l'étoile on doit pouvoir consulter la valeur du sigma de son voisin précèdent dans l'anneau. Pour consulter la valeur on utilise la méthode getNeighborProperty, spécifiant le numéro de port du bon voisin, c'est-à-dire le voisin i-1 pour le nœud i.

Nous sommes dans une configuration en anneau, chaque nœud à 2 voisins, pour être sur que le voisin sur le port 0 est le voisin précèdent il faut construire son anneau dans l'ordre, nœud par nœud. Ainsi, chaque nœud aura le nœud précèdent sur le port 0 et le nœud suivant sur le port 1. Attention néanmoins lorsqu'on ajoute le dernier nœud, on le relit au tout premier, ce qui fait que pour le premier nœud de notre anneau, le nœud précèdent est exceptionnellement sur le port numéro 1.

Sur ViSiDia, on accède à la liste des ports utilisés d'un nœud avec la méthode *getActiveDoors*. Les ports renseignés dans cette liste sont ajoutés au fur et qu'on créer des liens entre le nœuds, d'où l'importance de créer l'anneau dans l'ordre. Pour faciliter les choses j'aurai laissé un graphe déjà fait avec le code sur le dépôt GitHub.

Dans cet algorithme :

- Pour N0 : si σ0 = σn-1 alors σ0 ← σ0 + 1%k
- Pour Ni : si σi différent σi-1 alors σi ← σi-1

Ce qui fait que tous les nœuds vont prendre la valeur du précédent sauf N0 et donc s'aligner sur sa valeur sigma. Une fois que tous les nœuds sont à la même valeur que N0, c'est-à-dire que le dernier vient de passer à cette valeur alors N0 va pouvoir évoluer. Il prend sa valeur actuelle +1 et le tout modulo k. Ce qui fait que tant que sigma+1 < k, la valeur de sigma de l'ensemble des nœuds continue d'augmenter petit à petit mais que des que sigma+1 = k alors on retombe à 0. Les valeurs de sigma des nœuds de l'anneau vont donc osciller entre 0 et k.

```java
public class Algo_AutoStabilisant_GD_TP2 extends LC1_Algorithm {

    private int k = 7;

    @Override
    public String getDescription() {
        return "Algorithme de changement d'état lorsqu'un noeud possède le jeton --> Auto stabilisant\n"
                + "--> doit etre utiliser avec un anneau de maximum 9 noeuds";
    }

    @Override
    protected void beforeStart() {
        // chaque noeud sauvegarde son état dans la variable "label"
        //   remarque: tous les noeuds sont à "N" sauf 1 à l'état "A" pour signifier que c'est le N0
        setLocalProperty("label", vertex.getLabel());
        // au début on tire au sort une valeur pour sigma entre 0 et k-1
        setLocalProperty("sigma", new Random().nextInt(k));
        // affichage
        putProperty("affichageSigma", "sigma = "+getLocalProperty("sigma"), SimulationConstants.PropertyStatus.DISPLAYED);
    }
```

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

    // port du noeud précedent dans l'anneau ------------------------------------------------
    int numPort;
    // si c'est N0
    if(getLocalProperty("label").equals("A")){
        numPort = getActiveDoors().get(1);
    }
    // sinon si c'est un autre noeud
    else {
        numPort = getActiveDoors().get(0);
    }

    // vérification possession du jeton + calcule si oui ------------------------------------

    // si c'est N0
    if(getLocalProperty("label").equals("A")){
        // jeton si sigma 0 = sigma n-1
        if(getLocalProperty("sigma").equals(getNeighborProperty(numPort, "sigma"))) {
            // si oui alors calcul du nouveau sigma
            int temp = ((int) getLocalProperty("sigma") + 1) % k;
            // affectation de la nouvelle valeur
            setLocalProperty("sigma", temp);
        }
    }
    // sinon si c'est un autre noeud
    else {
        // jeton si sigma i est différent de sigma i-1
        if(! getLocalProperty("sigma").equals(getNeighborProperty(numPort, "sigma"))) {
            // si oui alors sigma i prend la valeur de sigma i-1
            setLocalProperty("sigma", getNeighborProperty(numPort, "sigma"));
        }
    }

    // affichage ----------------------------------------------------------------------------

    putProperty("affichageSigma", "sigma = "+getLocalProperty("sigma"), SimulationConstants.PropertyStatus.DISPLAYED);
```

## C. Exemple

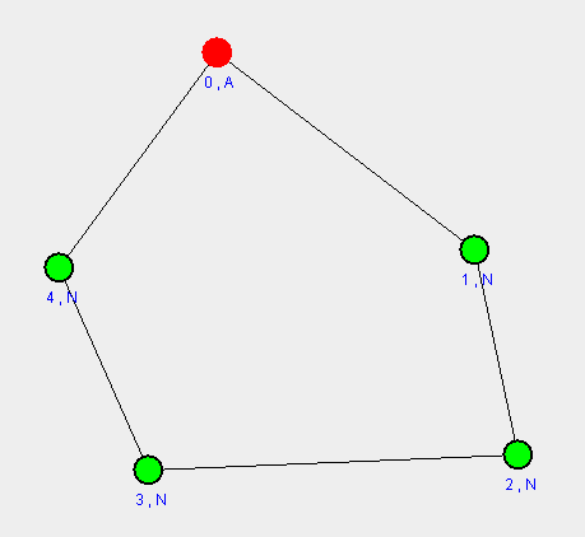


*Figure 6 - Anneau de base*

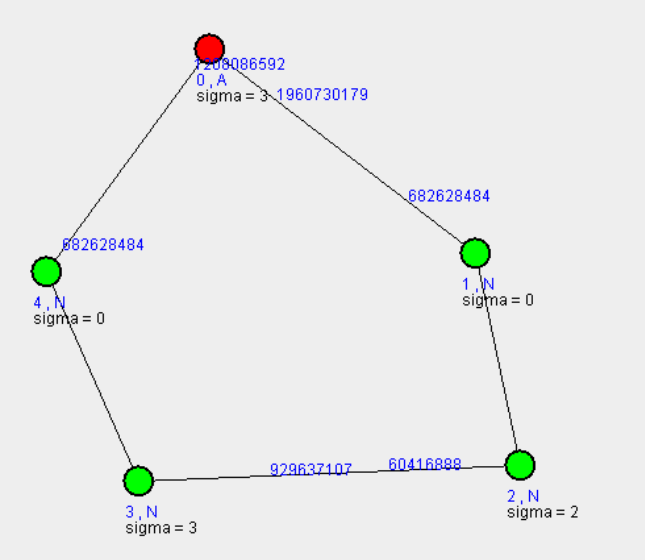


*Figure 7 - Lancement de l'algorithme, tous les nœuds tirent au hasard une valeur pour leur sigma*

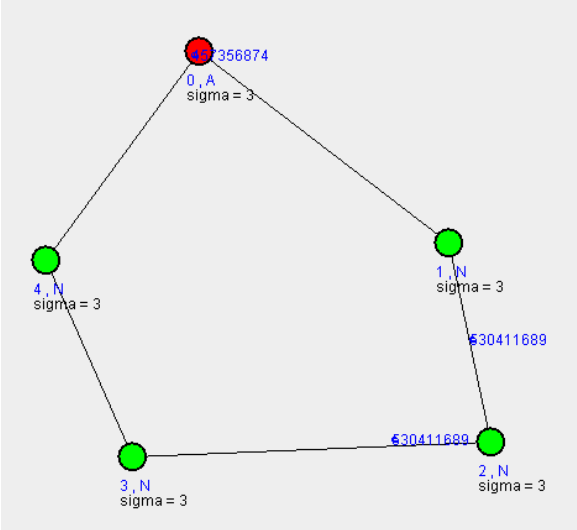


*Figure 8 - Tous les nœuds se sont alignés sur la valeur de N0, la prochaine fois que N0 sera le centre de l'étoile la valeur de son sigma évoluera*

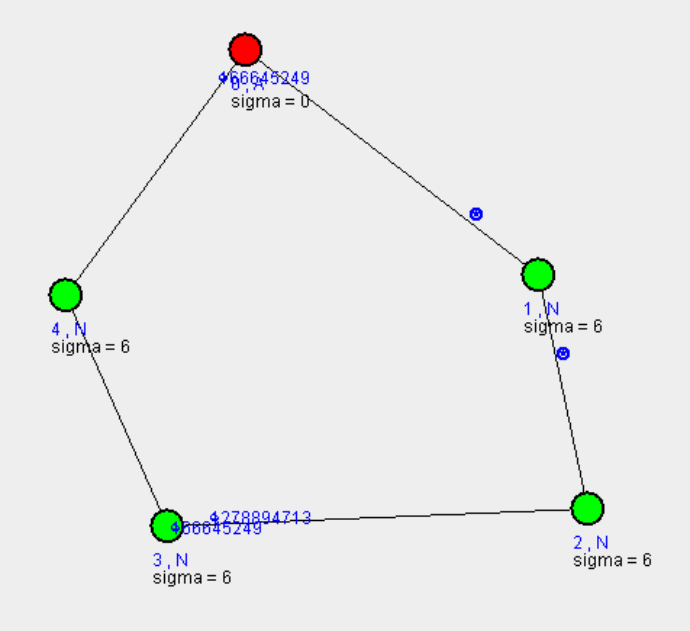


*Figure 9 - Tous les nœuds étaient à 6, k=7 alors le nœud N0 repasse à 0 et les autres suivront*

## 2. Pallier la perte d'une information locale

Dans cet exercice, nous supposerons que le graphe d'étude est connexe et que chaque nœuds a un degré supérieur ou égal à 2. Au départ, un et un seul nœud a une étiquette « A », tous les autres ont une étiquette « N ». Voir le graphe ci-dessous comme exemple :

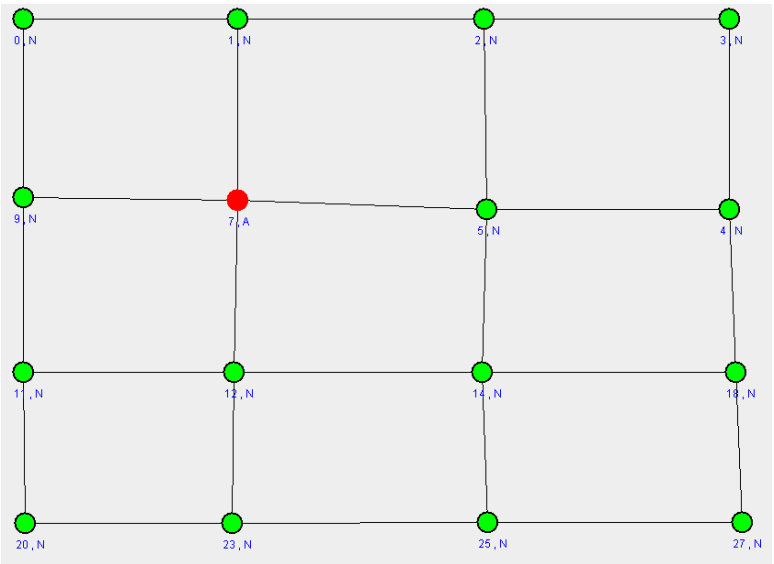


*Figure 10 - Graphe connexe, chaque a un degré supérieur ou égale à 2 et un seul nœud est à l'état "A"*

## A. Règles

Chaque nœud enregistre les identifiants et les étiquettes de ses voisins, à l'exception de celui qui se trouve sur le port 0. Au bout de 7 synchronisations, le nœud ayant pour étiquette « A » s'arrête.

```
private int nbrSynchronisation = 0;
private int [] idVoisin = new int[vertex.getDegree()-1];
private String [] etiquetteVoisin= new String[vertex.getDegree()-1];

@Override
public String getDescription() {
    return "Algorithme test";
}

@Override
protected void beforeStart() {
    // chaque noeud sauvegarde son état dans la variable "label" (1 seul à "A" et tout le reste à "N")
    setLocalProperty("label", vertex.getLabel());
    // sauvegarde son ID
    setLocalProperty("id", vertex.getId());
    // affichage
    putProperty("affichage", "nbr synchro= "+this.nbrSynchronisation, SimulationConstants.PropertyStatus.DISPLAYED);

}
```

```
@Override
protected void onStarCenter() {

    // nombre de synchronisation augmente
    this.nbrSynchronisation++;
    putProperty("affichage", "nbr synchro= "+this.nbrSynchronisation, SimulationConstants.PropertyStatus.DISPLAYED);

    // on sauvegarde l'état l'id des voisins
    for (int i=1; i<getActiveDoors().size(); i++) {
        int numPort = getActiveDoors().get(i);
        this.etiquetteVoisin[i-1] = (String) getNeighborProperty(numPort, "label");
        this.idVoisin[i-1] = (int) getNeighborProperty(numPort, "id");
    }

    // si c'est le noeud à l'état "A" et que les 7 synchronisations ont été atteintes
    if(getLocalProperty("label").equals("A") && this.nbrSynchronisation == 7) {
        // le noeud s'arrete
        localTermination();
    }

}
```

### 1) Comment peut-on détecter cet arrêt (au niveau des noeuds voisins) ?

Pour détecter l'arrêt d'un nœud on peut appeler la fonction *vertex.getDegree()* et comparer le résultat à celui du dernier appel, si le nœud arrêté fait partie des voisins alors on devrait obtenir un résultat inférieur de 1 lors du second appel de cette fonction.

**2) Faites en sorte que l'étiquette du noeud arrêté soit connue par son voisin qui l'ignorait (car "A" était sur le port 0 de ce voisin).**

Pour faire en sorte que l'état du nœud arrêté soit connu par le nœud voisin qui l'ignorait, on peut passer par un voisin tier. C'est-à-dire, si un nœud A s'arrête et que le nœud B ne connait pas son étiquette car il était sur son port 0 alors il peut passer par un voisin commun C, qui lui connait peut être l'étiquette du nœud arrêté. Remarque dans une grille il faut peut être passer par un voisin interposé pour atteindre le voisin qui aurait l'information.

**3) La récupération de l'information à la question 2 est-elle garantie ? Si oui, pourquoi ? Sinon expliquez les raisons qui pourraient empêcher cette récupération (ou décrivez un scénario rendant cette récupération impossible).**

Non, la méthode de récupération de l'information décrite dans la question deux n'est pas garantie. En effet, pour connaitre l'étiquette du nœud arrêté le principe était de demander à un autre nœud qui aurait peut être l'information mais il est possible que aucun nœud ne l'ait si par malheur le nœud arrêté se trouve sur le port 0 de tous ses voisins.

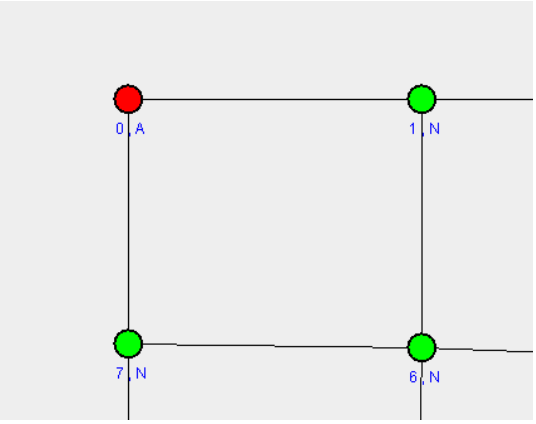


*Figure 11 - exemple cas impossible*

Si le nœud 0 s'arrête et que le nœud 9 souhaite connaitre son étiquette il peut avoir l'info auprès du nœud 1. Mais le nœud 1 à plusieurs voisins, il se peut que le nœud arrêté soit sur son port 0 et donc l'information sera perdue définitivement.