
UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ
UNIVERSITÉ DE BOURGOGNE
ESIREM
ÉCOLE SUPÉRIEURE D'INGÉNIEURS
NUMÉRIQUE ET MATÉRIAUX

Contents

Introduction

In this TP we will experiment with different types of algorithms in distributed systems seen in class during this TP. We are using in this TP JAVA programming language and the library ViSiDia. It also observed that we will be using three types of synchronizations- handshake, open star “étoile ouverte”, and closed star “étoile fermée” we’ll also see the spanning tree algorithm and the election of a leader in a tree algorithm plus computation of the order in a graph.

Here is a link to my github:

https://github.com/ChristianHasbani/TP_Systeme_Distribues

TP2: Implementation of the basic algorithm

1. Computing a spanning tree using the depth-first search

a) How the algorithm works

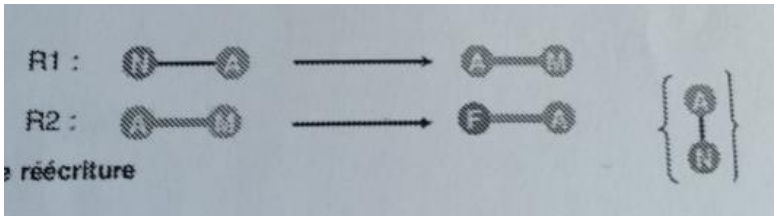

Figure 1-Rules for the algorithm

Rewriting rules:

- If a vertex is labeled N and its neighbor is labeled A, then the vertex goes to a state A, its neighbors goes to state M and the edge between these two vertices is marked.
- If a vertex labeled A is connected to a neighbor labeled M by a marked edge (and co vertex A has no neighbor N), then vertex A transitions to state F and its neighbor transitions to state A (the edge remains marked).

When we apply the first rule we need to mark the link between the local node and the neighbor node so we get around it in this TP using through the node which passes to the state “A” will record the port on which its parent is located thanks to the neighborDoor variable. Thus, during the application of the second rule, the node can verify that the neighbor with which synchronization is established is indeed located on the port where its parent is located.

b) The algorithm

As seen in the introduction, we will write the code of our algorithm in JAVA before importing it into VISIDia. We will use classes specific to VISIDia which will allow us to code the desired behavior for each vertex of the graph.

For our spanning tree algorithm, we will use a local synchronization between two nodes, in ViSiDia called LCD Algorithm. Each node sends an integer to its neighbors, this integer is worth 0 for all the neighbors except for the one that the node chooses for synchronization, in this case the integer sent is strictly positive. If two adjacent nodes choose each other, synchronization takes place. The node that sent the largest integer is responsible for applying the rewrite rules if the configuration allows it. Note that if the integers are equal then the synchronization fails and that several synchronizations can take place at the same time if they do not involve the same node.

The class LC0_Algorithm implements 4 main methods:

1. “getDescription” Method: returns String description of the algorithm we are writing.
2. “beforeStart” Method: code run by all nodes at the start before launching of synchronizations.
3. “onStarCenter” Method: code run by the node responsible for applying the rules of rewriting.
4. “clone” Method: returns a new object of the class.

c) Implementation of the algorithm in ViSiDia

To apply the second rule, we must be able to verify that the node no longer has any neighbors in state “N”. To verify this, we must ensure that each node retains the state of its neighbors in a table, so each node will also have in its attributes an array containing a list of the state of its neighbors called “neighborStates”. Since a node can only know the state of the neighbor it is synchronizing with, initially all nodes will assume that their neighbors are in state "N" and each time a synchronization occurs, the node table will be updated.

```java
@Override
protected void beforeStart() {
    setLocalProperty("label", vertex.getLabel()); //Set the label property for each node
    setLocalProperty("parent", -1); //Set the parent port to -1 to remember later the port that is the start of all the neighbors
    this.neighborStates = new String[vertex.getDegree()]; // get the number of neighbor nodes
    for(int i = 0; i < vertex.getDegree(); i++){
        this.neighborStates[i]="N"; //Set all neighbor states to N by default
    }
}
```

Figure 2-beforeStart Method

```java
@Override
protected void onStarCenter() {
    this.neighborStates[neighborDoor] = getNeighborProperty("label").toString(); //Update the neighbor states for the node rewriting the rules
    R1();
    R2();
}
```

Figure 3-onStarCenter Method

```
// Function for rule 1
private void R1() {
    //Check if rule 1 applies for those 2 nodes
    //Check if the local node has a label "N" and the neighbor node has the label "A"
   if (getLocalProperty("label").equals("N") && getNeighborProperty("label").equals("A")) {
       setLocalProperty("label", "A");// Change the local node's label to A
       setNeighborProperty("label", "M"); // The neighbor's to M
       setLocalProperty("parent",neighborDoor); //The node that applies the rule saved the port on which its parent is located to mark the link
       setDoorState(new MarkedState(true), neighborDoor); // Mark the link between those the local and neighbor node
       this.neighborStates[neighborDoor] = getNeighborProperty("label").toString(); // Update the neighbors states array
   }
}
```

Figure 4-R1 Method used in the onStarCenter Method.

```
// Function for rule 2
private void R2() {
    //Verify that rule 2 applies for this node and its neighbor
    // Check if the local node has the label "A" and the neighbor node has the label "M"
    if(getLocalProperty("label").equals("A") && getNeighborProperty("label").equals("M")){
        if(neighborDoor == (int) getLocalProperty("parent")){ // Check if the link is marked
            if(!neighborsExist()){ //There isn't any neighbors with the state "N"
                setLocalProperty("label","F"); //Change the label of the local node
                setNeighborProperty("label","A"); // Chagne the label of the neighbor node
            }
        }
    }
}
```

Figure 5-R2 Method used in the onStarCenter Method.

```
// Function to check if the nodes still has any neighbor nodes with the state "N"
private boolean neighborsExist(){
    boolean result = false;
    for(int i = 0; i<vertex.getDegree();i++){
        if(this.neighborStates[i].equals("N")){
            result = true;
        }
    }
    return result;
}
```

Figure 6-neighborExist Method to check if the node has more neighbors with the label "N"

We create the following graph in ViSiDia.

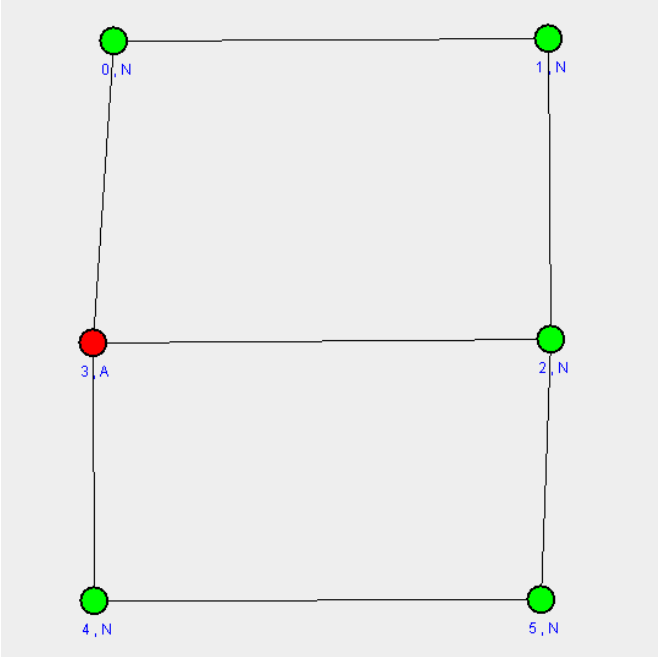

Figure 7-Graph before applying the algorithm.

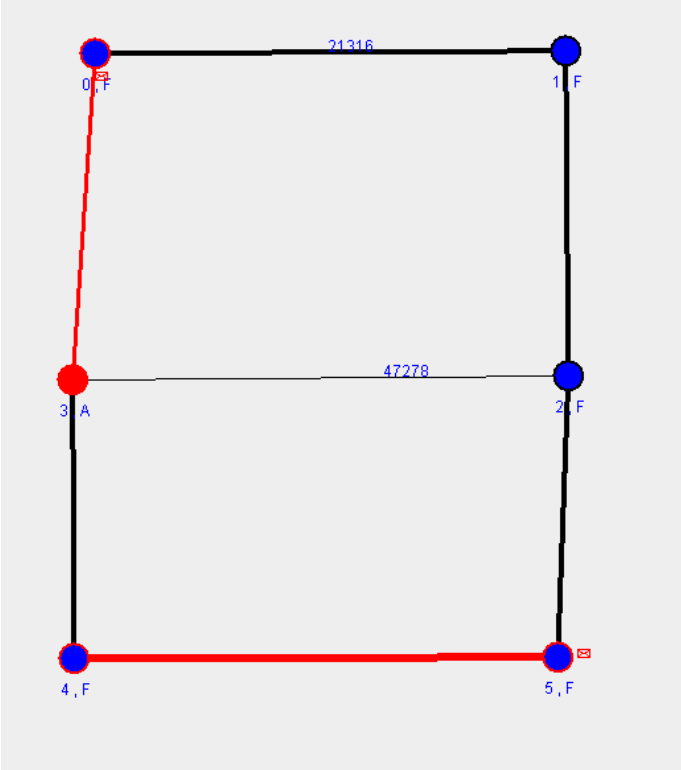

Figure 8-Graph after using the spanning tree algorithm

2. Calculating the order of the graph

a) Concept

Like the code we just written in the previous exercise we need to implement an extra feature to calculate the order the graph.

Since the spanning tree goes through all the vertices without cycling, this is very useful for us. From the preceding algorithm, we simply add a counter to each of the nodes, this counter is worth at base 1 (each node counts itself). This counter will be used each time the second rule is applied: once a node applies this rule, it goes into the “F” state and sends its counter to the node it synchronized to and adds it up on the counter of it. This means that once all the nodes have passed to the "M" state (except a last one to "A"), the tree is traversed, the second rule will apply, and the counters will start to go up little by little until the node initially in state A» which will thus be able to know the number of vertices in the graph.

b) Implementing the algorithm

We add a property of counter using the setLocalProperty method and display the counter using the putProperty method.

```
@Override
protected void beforeStart() {
    setLocalProperty("label", vertex.getLabel()); //Set the label property for each node
    setLocalProperty("parent", -1); //Set the parent port to -1 to remember later the port that is the start of all the neighbors
    this.neighborStates = new String[vertex.getDegree()]; // get the number of neighbor nodes
    for(int i = 0; i < vertex.getDegree(); i++){
        this.neighborStates[i]="N"; //Set all neighbor states to N by default
    }
    setLocalProperty("counter",1);// Add a counter property
    putProperty("Affichage",getLocalProperty("counter").toString(), SimulationConstants.PropertyStatus.DISPLAYED); // Display the counter
}
```

Figure 9-beforeStart method after adding the counter

We will also add the counter when the second rule is applied and redisplay the result when we are done

```
// Function for rule 2
private void R2() {
    //Verify that rule 2 applies for this node and its neighbor
    // Check if the local node has the label "A" and the neighbor node has the label "M"
    if(getLocalProperty("label").equals("A") && getNeighborProperty("label").equals("M")){
        if(neighborDoor == (int) getLocalProperty("parent")){ // Check if the link is marked
            if(!neighborsExist()){ //There isn't any neighbors with the state "N"
                setLocalProperty("label","F"); //Change the label of the local node
                setNeighborProperty("label","A"); // Chagne the label of the neighbor node
                setNeighborProperty("counter", Integer.parseInt(getLocalProperty("counter").toString()) +
                 Integer.parseInt(getNeighborProperty("counter").toString())); // Update the counter
            }
        }
    }
    putProperty("Affichage",getLocalProperty("counter").toString(), SimulationConstants.PropertyStatus.DISPLAYED);
}
```

Figure 10-Rule 2 method that is used in the onStarCenter method.

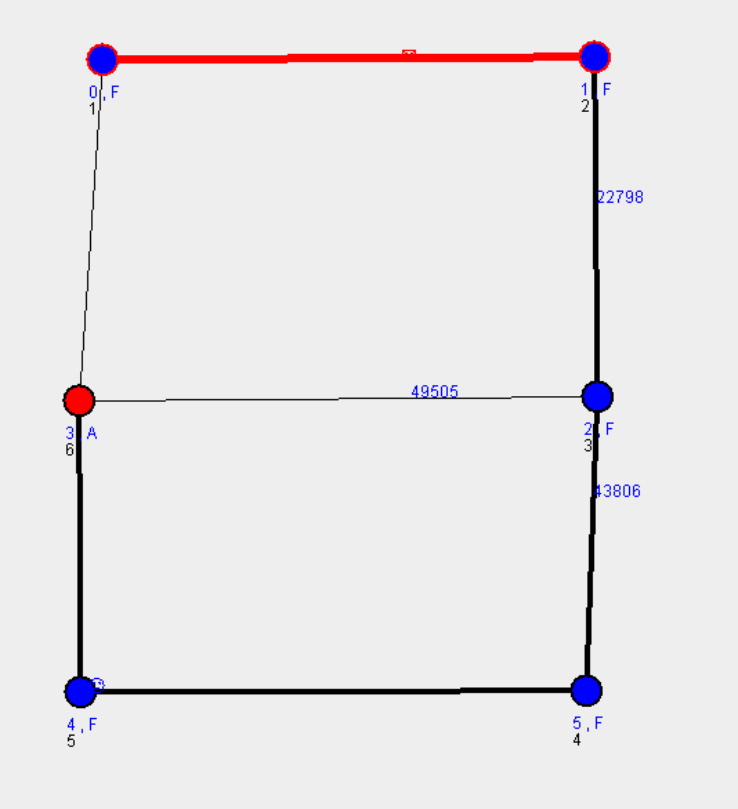

Figure 11-Graph after implementing the algorithm showing the counter in "black"

TP3: Electing a leader

The goal of this practical work is to write an algorithm allowing to elect a leader in a tree. For this we will use a local handshake type synchronization as in the previous lab, we will once again use the LC0_Algorithm class.

1. Concept of electing a leader in a tree

Progressive pruning:

- Initially, all nodes are in a neutral state, “N”
- As the algorithm progresses, nodes will transition to either the unelected state or the elected state.
- The unelected and elected states are final (when a node enters one of these states, it stays there until the end of the algorithm)
- When the algorithm ends, all nodes are in the state not elected except one and only one who is in elected status.

We call a “leaf” a node which has only one neighbor left in state “N”. Gradually these leaves will be eliminated and will pass to the “F” state, not elected, which means that new nodes will in turn become leaves and so on. We continue with this operation until there are only two leaves left, in this case it is the node which applies the rule during synchronization which is elected leader.

Version using the handshake

- Initially all nodes are in a neutral state “N”
- Each node has a neighbor counter

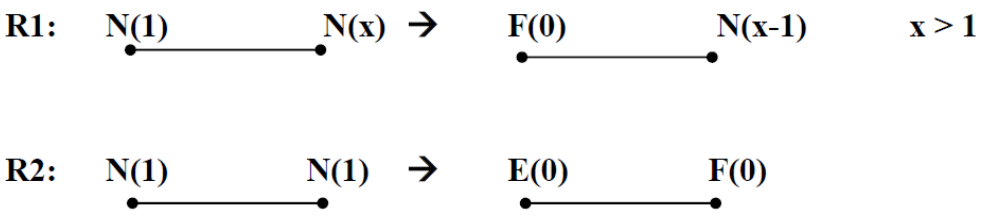

Figure 12-Rules for electing a leader

As explained just above, x designates the number of neighbors in the state “N” that a node has. When we apply the first rule, we eliminate a node that has only one neighbor in state "N", a leaf, and therefore inevitably the number of neighbors in state "N" of the node that it remains decreases by 1.

2. Implementing the algorithm

```
@Override
protected void beforeStart() {
    setLocalProperty("label", vertex.getLabel()); // label for each node
    setLocalProperty("nbNeighbors", vertex.getDegree()); // number of neighbor nodes
    putProperty("Affichage", "Neighbor N = " +getLocalProperty("nbNeighbors"),SimulationConstants.PropertyStatus.DISPLAYED); // display
}
```

Figure 13- beforeStart Method

```
@Override
protected void onStarCenter() {
 if(getLocalProperty("label").equals("N") && getNeighborProperty("label").equals("N")){ // Check if local and neighbor nodes are both in state "N"
        int nbNeighborsLocal = (int) getLocalProperty("nbNeighbors"); // Get the number of neighbor nodes of the local node
        int nbNeighborsNeighbor = (int) getNeighborProperty("nbNeighbors"); // Get the number of neighbor nodes of the neighbor node
        if(nbNeighborsLocal == 1 && nbNeighborsNeighbor > 1){ // Check 1st rule of our algorithm applies
            setLocalProperty("label","F"); // Change local node label to "F" and number of neighbors to 0
            setLocalProperty("nbNeighbors",0);
            setNeighborProperty("nbNeighbors",nbNeighborsNeighbor - 1); // Decrease the number of neighbors of the neighbor node by 1
        }
        else if(nbNeighborsLocal == 1 && nbNeighborsNeighbor == 1){ // Check if the 2nd rule applies
            setLocalProperty("label","E"); // Change local node label to "E" and number of neighbors to 0
            setLocalProperty("nbNeighbors",0);
            setNeighborProperty("label","F");// Change the neighbor node label to "F" and number of neihgbors to 0
            setNeighborProperty("nbNeighbors",0);
        }
        display();
    }
}
```

Figure 14- onStarCenter Method

```
// Method to display on the graph
private void display(){
    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","Neighbor N = " + getLocalProperty("nbNeighbors"),
            SimulationConstants.PropertyStatus.DISPLAYED);
        }
}
```

Figure 15- display method called in the onStarCenter method

3. Using the algorithm in ViSiDia

This algorithm works by initializing all nodes with the state “N” then displaying the label each neighbor “N” with the “putProperty” method and finally when there are no nodes with the label “N” it’s elected as the leader.

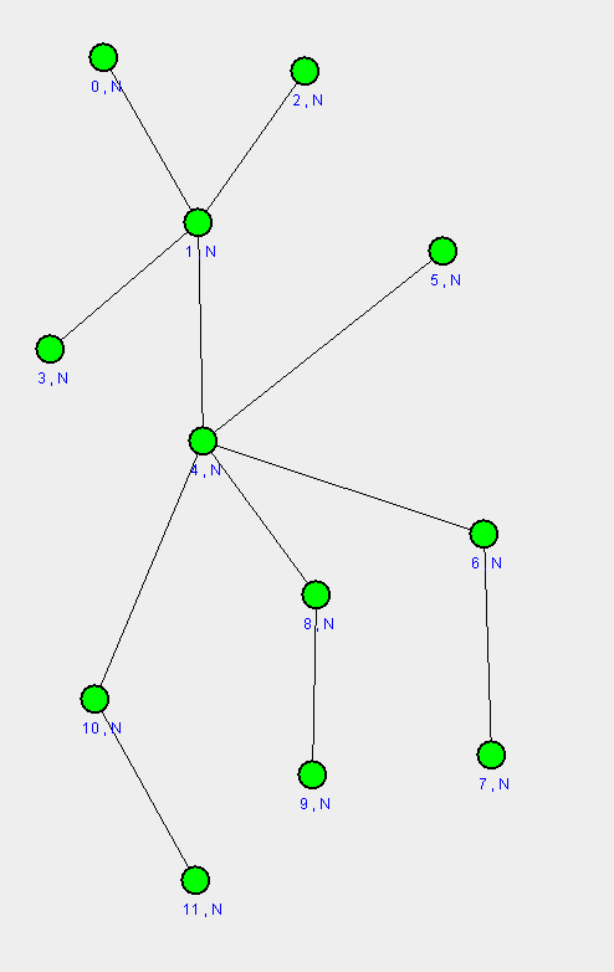

Figure 16- Graph before running the algorithm

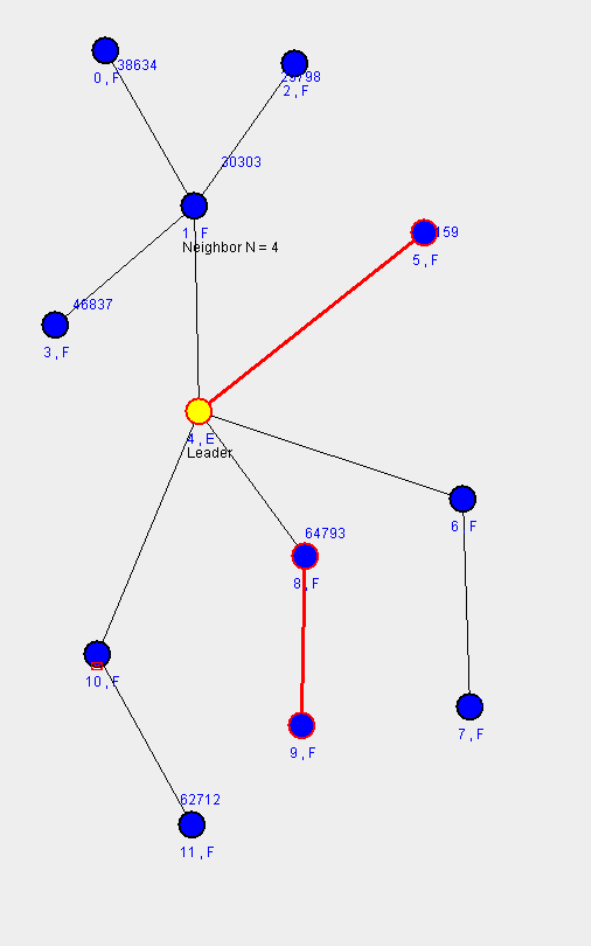

Figure 17- Graph after running the algorithm

TP4: Open Star

In this TP, we will write an algorithm that uses local synchronization of the type open star “étoile ouverte”, which uses in ViSiDia the LC1_Algorithm.

As a reminder, when an open star synchronization is established, the star center is responsible for applying the rewrite rules. The latter can modify:

- Its state
- The states of the edges that link it to its neighbors
- Warning: The node cannot modify the state of its neighbors

1. Spanning tree

a) Rewriting rules for the open start method

As for the spanning tree algorithm of TP2, at the start, all the vertices of the graph must be in state “N” except one in state A”, the one that initializes the calculation.

Rewrite rules:

- If the center of the star is labeled N and it has at least one neighbor labeled “A”, he chooses one (let's call this neighbor “X”)
- The center changes its label to “A”
- The center marks the edge that connects it to “X”

b) Implementing the algorithm

Similar to LC0_Algorithm the LC1_Algorithm contains 4 main method “getDescription”, “beforeStart”, “onStarCenter”, “clone”.

As we will have to check if a node has neighbors in the as state or not, we will use a new method belonging to this class, “getActiveDoors”. This method returns an ArrayList containing the port numbers on which a node's neighbors are connected.

c) Using the algorithm in ViSiDia

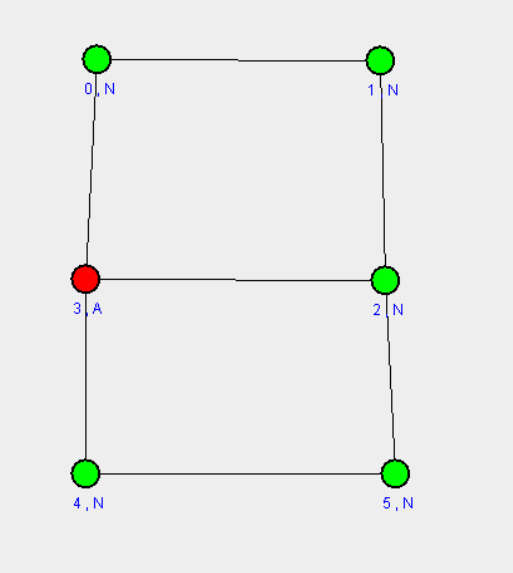

Figure 18- Graph before using the algorithm

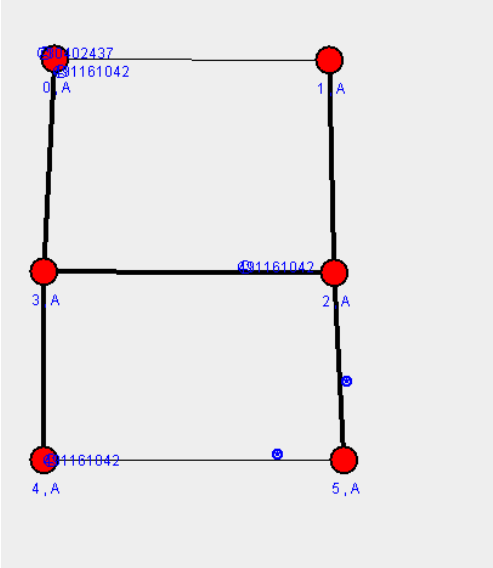

Figure 19- Graph after using the algorithm

2. Election of a leader

a) Rewriting rules for electing a leader

To elect a leader using an open star:

- If the center of the star is in state “N” and it has only one neighbor “N” then it is a “leaf”, and this center goes to state “F”. He is eliminated.
- If the center is in the "N" state and it has no other neighbor also in the "N" state, then this center is the last one not eliminated, he passes to the “E” state, he is elected leader.

b) Implementing the algorithm

```
@Override
protected void beforeStart(){
    setLocalProperty("label",vertex.getLabel());// label for each node
}
```

Figure 20- beforeStart Method

```java
@Override
protected void onStarCenter(){
   if(getLocalProperty("label").equals("N")){ // Check if the local node has the label "N"
    int nbNeighbors = countNbNeighbors(); // Get the number of neighbors

    if(nbNeighbors == 1){ // Check if the number of neighbors with the label "N" is 1
        setLocalProperty("label","F"); // This node failed to be a leader its a follower node
    }

    else if(nbNeighbors == 0){ // Check if the number of neighbors with tthe label "N" is 0
        setLocalProperty("label","E"); // This node is the Leader
    }

   }
}
```

Figure 21- onStarCenter Method

```java
//Method to count number of neighbors of each node
private int countNbNeighbors(){
    int result = 0;
    for(int i = 0; i< getActiveDoors().size(); i++){ // loop through all the edges of the local node using the getActiveDoors method
        int numPort = getActiveDoors().get(i);
        if(getNeighborProperty(numPort,"label").equals("N")){
            result++; // for each neighbor of with the label "N" increment the number of neighbors
        }
    }
    return result; // return the result containing the number of neigbors of the local node with the label "N"
}
```

Figure 22-countNeighbors method used in the onStarCenter method

c) Applying the algorithm in ViSiDia

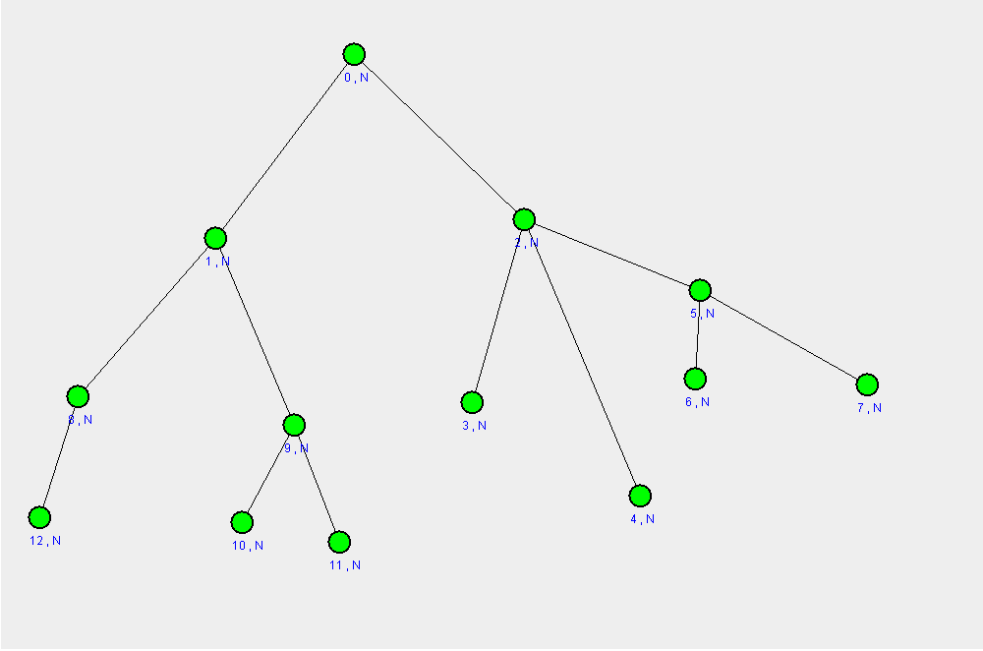

Figure 23- Graph before using the algorithm

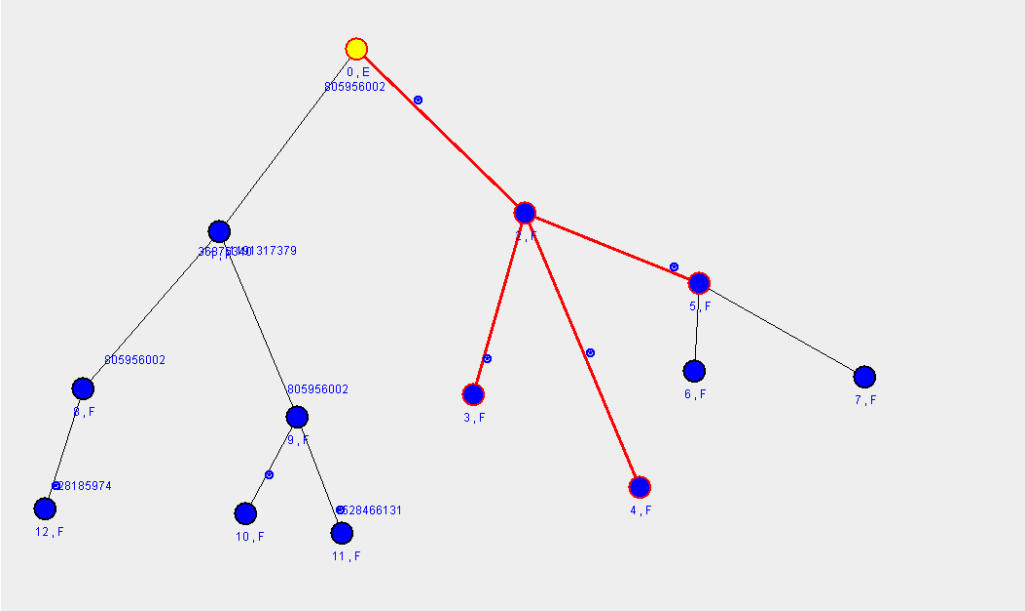

Figure 24- Graph after using the algorithm

TP5: Closed star

The goal of this TP is to write algorithms which are based on a local synchronization of closed star type, we will use for that the class LC2_Algorithm.

As a reminder, when a closed star type synchronization is established, the center of the star is responsible for applying the rewrite rules. The latter can modify:

- Its state
- The states of the edges that link it to its neighbors
- The states of its neighbors

1. Spanning tree

a) Rewriting rules for closed star

As always, it will be assumed that at the start all the nodes are in a state “N” except one in the state “A” which will initiate the calculation.

If the center of the star is labeled A It relabels all its neighbors N to A

It marks the respective edges that bind it to these neighbors (newly labeled A)

b) Implementing the algorithm

```java
@Override
protected void beforeStart(){
    setLocalProperty("label",vertex.getLabel());// label for each node
}
```

Figure 25- beforeStart method

```java
@Override
protected void onStarCenter(){
   if(getLocalProperty("label").equals("A")){/// Check if the local node has label = "A"
    for(int i=0; i<getActiveDoors().size(); i++){ // Get all the edges using the getActiveDoors method
        int numPort = getActiveDoors().get(i); // Get the port number of each edge
        if(getNeighborProperty(numPort,"label").equals("N")){ // Check if the neighbor node has the state "N"
            setNeighborProperty(numPort,"label", "A"); //if the condition is verified then change the neighbor's label to "A"
            setDoorState(new MarkedState(true),numPort); // Mark the edge between the local node and the neighbor node
        }
    }
   }
}
```

Figure 26- onStarCenter method

c) Using the algorithm in ViSiDia

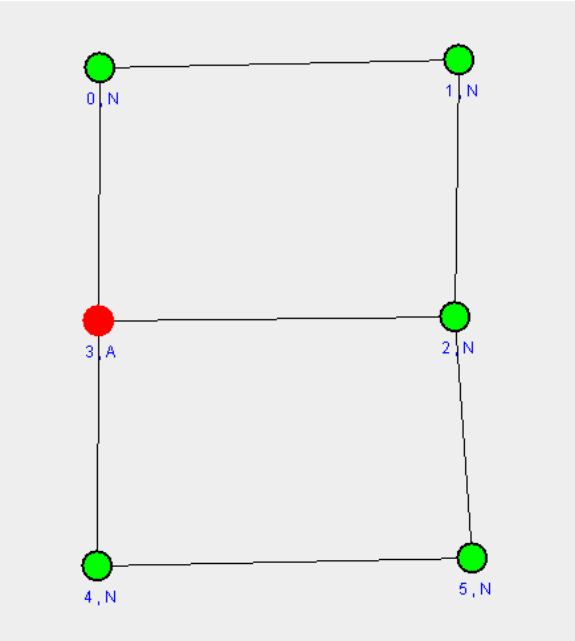

Figure 27-Graph before using the algorithm

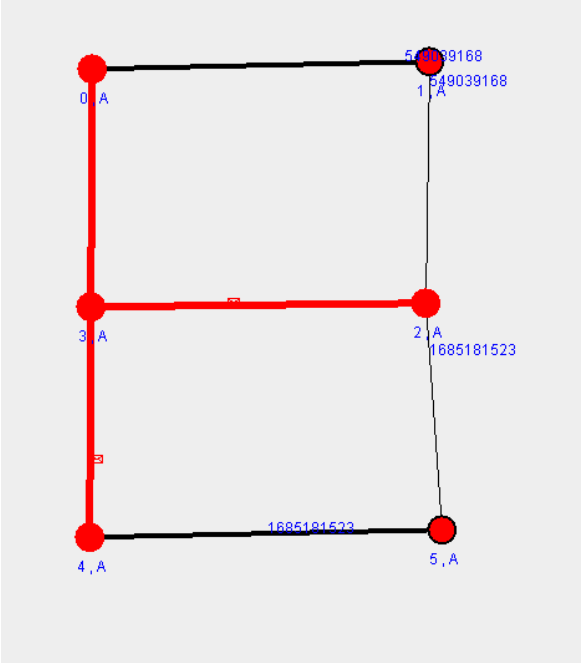

Figure 28- Graph after using the algorithm

2. Election of a leader

a) Rewriting rules

It is assumed that each node knows the number of active neighbors it has. If the center is in the “N” state, it asks all its neighbors also in the “N” state and which have only one active neighbor to move to the “F” state (the node is eliminated). If the center is in state "N" and it has no active neighbor then it is elected, it goes to state "E".

b) Implementation

For this algorithm we must be able to know how many neighbors in state “N” a node has. For this we also set up a counter, updated at each synchronization and thus allowing us to keep track of the number of neighbors with the label “N”

```
@Override
protected void beforeStart(){
    setLocalProperty("label",vertex.getLabel()); // Give a label for each node
    setLocalProperty("nbNeighbors",vertex.getDegree()); // Get the number of neighbors for each node
}
```

Figure 29- beforeStart method

```java
@Override
protected void onStarCenter(){
    if(getLocalProperty("label").equals("N")){ // Check if the local node has the label "N"
     int nbNeighbors = checkNeighbors();
     if(nbNeighbors == 0){ // Check if the local node has any local neighbors with the label "N"
         setLocalProperty("label","E"); // If no neighbors with label N then this node is the Leader
     }
    }
}
```

Figure 30- starOnCenter method

```java
//Method to check if a node has neighbors
private int checkNeighbors(){
    int nbNeighbors = 0;
    for(int i = 0; i< getActiveDoors().size(); i++){ // Loop through the edges using the getActiveDoors method
        int numPort = getActiveDoors().get(i); // Get the port number of each edge
        if(getNeighborProperty(numPort,"label").equals("N")){ // Check if the neighbor node has the label "N"
            nbNeighbors++;
            if((int) getNeighborProperty(numPort,"nbNeighbors") == 1){ // Check if the local node has 1 neighbor
                setNeighborProperty(numPort,"label", "F"); // Rewrite this node's label to "F" so it's not the leader
            }
        }
        setLocalProperty("nbNeighbors", nbNeighbors);// Set the number of neighbors after adding the new value
    }
    return nbNeighbors; // Return the number of neighbors
}
```

Figure 31- checkNeighbors method to get the number of neighbors with the label "N"

c) Using the algorithm in ViSiDia

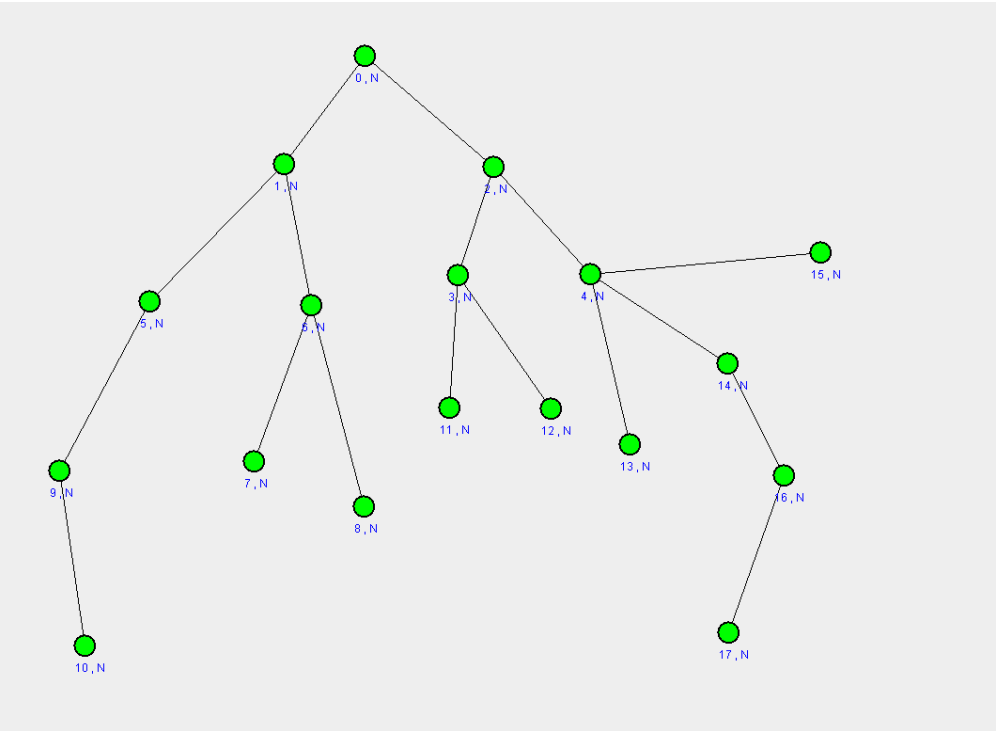

Figure 32-Tree before using the algorithm

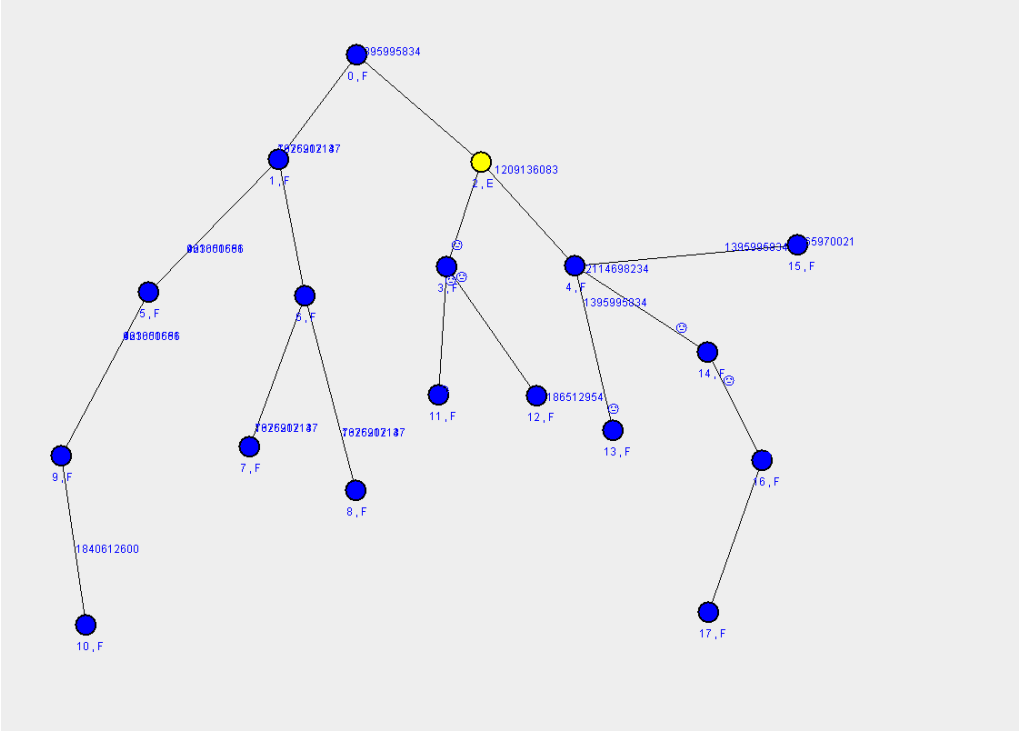

Figure 33-Tree after using the algorithm

3. Detecting the local termination

a) Spanning Tree

In the spanning tree algorithm we add a few adjustments to the onStarCenter method, whereby we add a counter of the number of neighbors with the label “N” and once this counter is 0 indicating that there are no neighbor with the label “N” for the local node we call the “localTermination” method.

```
@Override
protected void onStarCenter(){
    int countNeighborsN = 0;
   if(getLocalProperty("label").equals("A")){// Check if the local node has label = "A"
    for(int i=0; i<getActiveDoors().size(); i++){ // Get all the edges using the getActiveDoors method
        int numPort = getActiveDoors().get(i); // Get the port number of each edge
        if(getNeighborProperty(numPort,"label").equals("N")){ // Check if the neighbor node has the state "N"
            countNeighborsN++;
            setNeighborProperty(numPort,"label", "A"); //if the condition is verified then change the neighbor's label to "A"
            setDoorState(new MarkedState(true),numPort); // Mark the edge between the local node and the neighbor node
        }
    }
    if(countNeighborsN == 0){
        localTermination();
    }
    }
}
```

Figure 34-modified onStarMethod for the spanning tree algorithm

b) Electing a leader

In the case of electing a leader algorithm we add small condition when the node has already passed to the “F” or “E” state we call the “localTermination” method.

```
@Override
protected void onStarCenter(){
   if(getLocalProperty("label").equals("N")){ // Check if the local node has the label "N"
    int nbNeighbors = checkNeighbors();
    if(nbNeighbors == 0){ // Check if the local node has any local neighbors with the label "N"
        setLocalProperty("label","E"); // If no neighbors with label N then this node is the Leader
    }
   }else if (getLocalProperty("label").equals("F") || getLocalProperty("label").equals("E")){
    localTermination();
   }       You, 7 days ago • TP5 …
}
```

Figure 35-onStarCenter method in electing a leader algorithm

Table of figures: