Algorithmique de la mobilité
Wireless Rechargeable Sensor Networks

Kinda AL CHAHID - Inès FRIHI - Othmane FOUZI

January 2018

1 Introduction

Lors de l'UE "Algorithmique de la mobilité", nous avons dû réaliser un projet basé sur cette question : "Comment faire communiquer des objets mobiles?". Pour cela, l'énoncé est transposé à une difficulté existante : des noeuds transmettent des informations à intervalles irréguliers. A chaque transmission d'information, leur batterie se décharge, des robots doivent les survoler régulièrement pour les recharger et tout cela est coordonné par une *basestation* qui récupère les informations des capteurs et ordonne aux robots les destinations de chaque noeud. Plusieurs approches sont faisables. Au bout d'un certain temps fixe, les robots repartent vers la base pour actualiser leurs informations mais cela signifierait une perte de temps régulière. Une gestion par rapport à la distance ou par descendance de chaque noeud et ainsi transmettre une seule fois l'ordre aux robots. De plus, une question bonus fut proposée pour imposer les robots à visiter la base régulièrement : que se passe-t-il si un capteur tombe en panne ? En ne visitant la base qu'une seule fois, les robots ne seront jamais informés de cette panne et continueront à visiter ce noeud disparu. Nous verrons dans ce rapport les différents algorithmes travaillés, les limitations de notre code et la réflexion que nous avons eue concernant la partie bonus.

2 Algorithmes étudiés

Lors de ce projet, nous avons dû réécrire à quatre reprises le code. Chaque version possédait des avantages et des inconvénients mais ces versions sont divisibles en deux grandes parties:

- La première se base sur le principe de distance uniquement.
- L'autre sur la descendance des noeuds.

2.1 Algorithme comprenant la descendance d'un noeud

Cet algorithme ne fut pas le premier, mais il correspond aux versions 2 et 3 de notre code.
Voir en annexes les fichiers SensorChild et BaseStation Child.

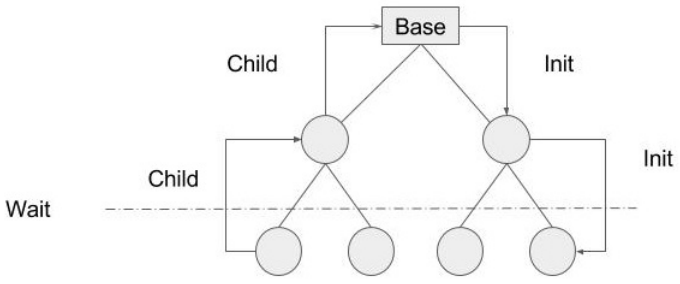

Figure 1: Algortihme des descendants

L'algorithme fonctionne en plusieurs étapes. Tout d'abord, la *basestation* envoie l'ordre à ses voisins directs (des capteurs) de s'initialiser et ces derniers transmettent l'information à leurs propres voisins directs qui, à leur

tour, la transmettent à nouveau. Ainsi, un arbre couvrant se forme et chaque noeud a connaissance de son *Parent*. Lors de l'étape d'initialisation, un compteur démarre pour chacun des noeuds : il s'agit de la variable *timer*.

Une fois le compteur enclenché pour chaque noeud, on parvient à une étape de comptage : il s'agit de déterminer si chaque noeud est un parent (il possède alors un ou plusieurs descendants) ou bien une feuille (il n'a alors aucun descendant). Chaque noeud envoie son état (feuille ou parent) à son parent et lors de la retransmission de cette information, le *timer* du parent repasse à zéro. S'il ne reçoit aucune information de type *parent* ou *children*, le *timer* du noeud s'incrémente de 1. Si le *timer* possède une valeur supérieure à 1, cela signifie que plus aucune descendance ne se manifeste ou qu'il n'y en a pas. Par conséquent, le noeud envoie dans une variable *Node* ses propres coordonnées en x et y ainsi qu'un ID correspondant à la variable *nb_children*.

Dans un troisième temps, la *basestation* comptabilise le nombre de descendant qu'elle a, ainsi que leurs coordonnées, mais elle enregistre également tous les robots détectés et les fait patienter au-dessus d'elle pendant qu'elle récupère le plus d'informations possibles.

Finalement, cette approche n'a pas pu aboutir car nous n'arrivions pas à transmettre aux robots un parcours optimal en fonction des descendants. Au bout de quelque semaines d'essais infructueux, nous avons préféré abandonner cette option pour reprendre le premier algorithme que nous avions trouvé, celui qui est basé sur les distances et qui donnait des résultats convenables.

2.2 Algorithme utilisant la distance

La première version est assez basique et se repose sur le principe du voyageur de commerce (TSP - Travelling Salesman Problem).

Lors de notre première tentative, nos scores, pourtant corrects, nous semblaient insuffisants au vu du résultat d'autres groupes qui parvenaient à avoir des temps d'exécution infinis. Cependant, puisque l'algorithme des descendants ne fonctionnait pas, nous avons préféré repartir sur notre première idée.

Notre idée de départ était que le sujet nous faisait énormément penser au TP du voyageur de commerce, donc nous nous sommes basés sur ce dernier. Durant le TP, Karl Schulz nous avait fourni une fonction pour le nearest neighbour. Nous l'avons réutilisé pour ce projet.

L'algorithme fonctionne de la manière suivante : une étape d'initialisation et de détection des noeuds, puis une étape de tri et de répartition des noeuds.
Dans un premier temps, chaque noeud s'initialise, mais à chaque message "init" reçu, il répond par ses propres coordonnées.

Dans un deuxième temps, lorsque chaque noeud a répondu (le temps que les robots se manifestent) la base effectue un tri de l'ensemble des informations qu'elle a reçues. Le tri repose sur une fonction nearest neighbour, puis une répartition est faite selon un quota d'un quart de noeud pour un robot contre trois quarts pour le deuxième.

Cet algorithme possède un temps d'exécution assez important. Toutefois, une astuce a permis d'améliorer encore ce temps.

En effet, chaque robot possède une distance de détection qui permet de détecter des capteurs ou la base avec une distance de 30, nommé range. Sur les fichiers fournis lors de ce projet, le robot considère avoir atteint sa destination si l'avatar du robot était au-dessus du noeud. Notre idée fut de réduire cette distance. Ainsi, si un capteur se trouve dans la zone de détection du robot (range - 3), le robot considère avoir atteint sa destination et la change.

Cette astuce peut sembler étrange car, si cela était utilisé sur un réseau existant, un robot verrait de loin un capteur et s'en éloignerait dès que ce dernier entrerait dans le champ de vision du robot. Or, il n'en est rien : un récent article fait part d'une nouvelle technologie émergente, celle d'un appareil (Pi[2]) qui permet la recharge à distance de plusieurs appareils en même temps et sans contact.

Grâce à cette astuce, l'algorithme exécuté sur la topologie fournie fonctionne : les noeuds ne sont jamais tombés à zéro de batterie au bout de deux heures de tests. Cependant, il est très facile de casser notre algorithme. Nous détaillerons les limitations dans le chapitre suivant.

Les différentes versions sont disponibles sur : https://github.com/SWGAKamui/Projet_AlgoMob

Nous vous recommandons la dernière mise à jour de la branche master(version rendu), et la dernière version de la branche tpfix (méthode descendants)

3 Limitations et améliorations possible

Lors de nos tests, nous avons sélectionné deux topologies identiques. Seul le positionnement de la base permet d'avoir des arbres couvrants extrêmement différents.

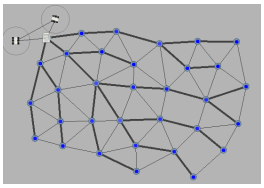

Figure 2: Topologie de base

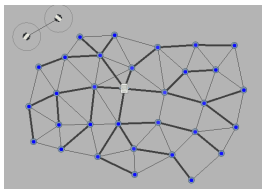

Figure 3: Topologie test

La première topologie (figure: 2) était celle fournie au départ et notre temps d'exécution dépasse les deux heures sans aucune batterie vide.

La deuxième topologie (figure: 3) aura toujours une batterie qui se videra complètement au bout de quelques secondes.

Pour régler la deuxième topologie, nous aurions aimé avoir une fonction qui détecte les noeuds les plus proches de la base et les envoie au premier robot tandis que les noeuds suivants seraient envoyés au deuxième robot.

Pour résumer, la zone bleue aurait été traitée par le premier robot (figure: 4 & 5) (avec un parcours de type nearest neighbour des noeuds de cette zone) et la partie verte par le deuxième. Puisque les noeuds qui ont beaucoup de descendants sont proches de la base, la gestion des descendants aurait également été faite en partie.

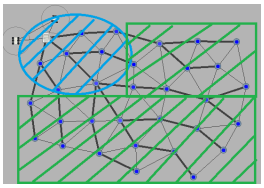

Figure 4: Répartition avec la topologie de base.
Robot 1 a sa zone bleue et Robot 2, sa zone verte

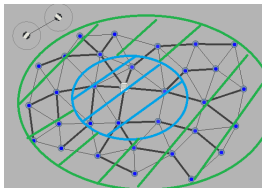

Figure 5: Répartition avec la topologie test.
Robot 1 a sa zone bleue et Robot 2, sa zone verte

Cette fonction a été implémentée, mais cet algorithme ne fonctionnait plus sur la première topologie (figure: 2). Nous avons donc fait le choix de ne pas le rajouter à notre projet mais plutôt d'en parler ici.

4 Question Bonus

Bien que non implémenté, nous avons tenté de résoudre cette question bonus. Notre idée fut amené lors d'un cours de sécurité des réseaux, lorsque notre professeur nous indiqua l'existence de "super cookie" nommé Evercookie[1]. Un Evercookie s'installe en plusieurs nombre à des endroits différents connus entre toutes ces versions. Lors de la suppresion de l'un d'entre eux, les autres jouent un rôle de sentinelle et recréer le cookie supprimer.

Dans notre cas, les noeuds ont conscience de leur voisins directs car la distance leur permet de les détecter. Ainsi, lors de l'initialisation, les noeuds enregistre le nombre de voisins qu'ils ont ainsi qu'une liste contenant leurs voisins. A Intervalle régulier, les noeuds vérifient que ce nombre n'a pas bougé. Si il y a un changement, ils envoient l'information ainsi que le noeud manquant à la base (Un super-noeud pourra être "élue" pour éviter la redondance d'envoi). Les robots retournent vers la base pour vérifier qu'il n'y a pas de changement. Si il y en a un, les robots enregistre la nouvelle version de la liste de noeud à visiter et repars.

5 Conclusion

Notre projet fut en partie réussi. Notre code fonctionne sur la topologie de base et permet d'avoir des temps d'exécution infini. Cependant, malgré nos efforts, l'algorithme fonction avec les conditions de départ mais n'est pas adaptable à des modifications : l'ajout de robots, de noeuds ou une topologie différente. Il est également possible d'améliorer l'algorithme en y rajoutant des notions de distance. Avec plus de temps, nous aurions pu essayer d'implémenter la partie bonus (gestion de la disparition d'un noeud) pour voir si notre idée fonctionnait.

References

[1] Samy Kamkar. EverCookie. `https://samy.pl/evercookie/`, 2010. [Online; accessed 05-Septembre-2017].

[2] David Nield. Pi wirelessly charges your devices at a distance, no mat required. `https://newatlas.com/pi-wireless-charger/51402/`, 2017. [Online; accessed 04-Septembre-2017].

Annexe

```
public class SensorChild extends Node {
    private Node parent = null;
    int battery = 255;
    private int nb_children = 0;
    private int time = 0;
    private Boolean send = true;
    private Boolean isParent = false;
    private Boolean isLeaf = false;
    @Override
    public void onMessage(Message message) {
        switch (message.getFlag()) {
            case "INIT":
                if (parent == null) {
                    // ..... Identique
                    send(parent, new Message(null, "PAR"));
                }
                break;
            case "SENSING":
                //.....Identique
            case "PAR":
            case "CHILD":
                this.isParent = true;
                this.isLeaf = false;
                if (message.getFlag().equals("PAR"))
                    this.nb_children++;
                this.time = 0;
                send(parent, message);
                break;
        }
    }
    @Override
    public void send(Node destination, Message message) {
        if (battery > 0) {
            super.send(destination, message);
            battery--;
            if(battery == 0)
                System.out.println("Batterry " + getTime()+"  "+this.getID() + " nb_children
                    "+nb_children);
            updateColor();
        }
    }
    @Override
    public void onClock() {
        if (parent != null) { // if already in the tree
            if (Math.random() < 0.02) { // from time to time...
                double sensedValue = Math.random(); // sense a value
                send(parent, new Message(sensedValue, "SENSING")); // send it to parent
            }
            if(this.send && (this.isLeaf || (this.isParent && this.time > 1))){
                this.send = false;
                Node node = new Node();
                node.setLocation(this.getX(), this.getY());
                node.setID(this.nb_children);
                send(parent, new Message(node, "CHILD"));
            }
            else if (send && !this.isParent && !this.isLeaf)
                if(time > 0)
                    isLeaf = true;
            time++;
        }
    }
    //.....Fonction updateColor() identique
}
```

```java
import jbotsim.Message;
import jbotsim.Node;
import java.util.ArrayList;

public class BaseStationChild extends Node {
    private ArrayList<Node> listNode_ = new ArrayList<>();
    private ArrayList<Robot> listRobot = new ArrayList<>();

    private int send = 0;
    private int timer;
    private Tri tri = new Tri(); // Fonction de tri en fonction de la distance entre les noeuds et la
        taille d'un noeud (nombre de descendants)


    @Override
    public void onClock() {
        if (listNode_.size() != 0 && timer > 1 && listRobot.size() != 0 && listRobot.size() > send) {
            if(send == 0) {
                tri.setOption(listRobot.size(), listNode_, this);
            }
            System.out.println(listRobot.size()+ " size "+listNode_.size());
            send(listRobot.get(send), new Message(tri.getListNode(send), "LIST"));
            send++;
        }
        timer++;
    }
    @Override
    public void onStart() {
        setIcon("src/server.png"); // to be adapted
        setSize(12);
        // Initiates tree construction with an empty message
        sendAll(new Message(null, "INIT"));
    }
    @Override
    public void onMessage(Message message) {
        if (message.getFlag().equals("CHILD"))
            if (!listNode_.contains(message.getContent())){
                listNode_.add((Node) message.getContent());
                timer = 0;
            }
        if (message.getFlag().equals("SEND_LIST")) {
            if (!listNode_.contains(message.getContent())) {
                listRobot.add((Robot) message.getSender());
                timer = 0;
            }
        }
    }

}
```