Rapport de projet
ASPD

Sarah Combe Deschaumes
Mathieu Dreyer
Marine Riva

31 mai 2021

Table des matières

1 Introduction

Le but de ce projet est d'appréhender des algorithmes distribués, de déterminer des facultés à modéliser un problème et enfin des facultés d'analyse.

Le projet se découpe en deux grandes parties. La première vise à modéliser et analyser deux algorithmes, work1 et work2 à l'aide des langages TLA/TLA+ et PlusCal.
Work1 représente le partage d'une ressource R, gérée par le serveur Q, par différents processus $P_1 ... P_n$. Le but est de proposer une technique de partage équitable entre les processus, sans utiliser de règle de communication trop stricte, comme FIFO.
Work2 est basé sur une analyse et sur la proposition d'algorithme de ramasse-miettes (garbage collection) de Ben-Ari. Il s'agit de modéliser cet algorithme.

La deuxième partie consiste en l'analyse d'un article présentant un algorithme et d'expliquer son fonctionnement. Le dossier que nous avons à analyser s'intitule ***A Distributed Algorithm for Minimum-Weight Spanning Trees***, de G.Gallager, A. Humblet et M. Spira.

Dans un premier temps, nous allons présenter la mise en place du projet, puis nous aborderons chacune des trois parties du travail, avant de conclure.

Il est important de prendre en compte, pour la partie sur l'explication de l'article, que nous avons choisi d'ajouter les définitions des mots importants qui sont évoqués et indispensables à la compréhension de l'article, ainsi que des schémas explicatifs. Ces ajouts n'ont pas été comptabilisés dans les explications du fonctionnement, ne devant pas dépasser 4 pages.

2 Mise en place du projet

Le projet étant découpé en trois parties, nous avons décidé dans un premier temps d'affecter une partie à chacun des membres du groupe, non pas par affinité mais par tirage au sort.
Par la suite, nous avons croisés nos travaux, afin d'effectuer des vérifications et de permettre une certaine entraide.

3 Partage de ressource (work 1)

L'algorithme vise en la création du partage d'une ressource R par un serveur Q à des processus $P_1 ... P_n$. Les communications se font par envoie et réception de messages et on suppose que les communcations sont fiables (pas de perte de message). On souhaite mettre en place une technique permettant le partage équitable de la ressource R par tous les processus P_n.

Il existe trois façons d'exprimer des exigences, qui sont liées à l'état initial de plusieurs composants :

- Par la Confirmation des conditions initiales de tous les composants. Bien que ce soit symétrique, cela semble inutilement superflu.
- L'assignation arbitraire des exigences à l'un des composants. Intuitivement, nous attribuons la responsabilité au composant pour nous assurer que les exigences sont respectées.
- La confirmation que les exigences sont distinctes des spécifications de l'un ou l'autre des composants. Cela montre intuitivement qu'il s'agit d'une hypothèse sur la façon dont les composants sont assemblés, et non sur les exigences des composants individuels.

Dans TLA+, une action est une formule qui décrit comment l'état change. Plus précisément, c'est une formule qui est vraie ou fausse pour une paire d'états. Nous disons s→t et A step if action et A est "vrai" pour les états pairs (s,t). Pour chaque opération , la traduction TLA+ définit une action L qui représente l'opération, en d'autres termes, s→t et une étape si l'exécution de l'opération L dans les états peut produire l'état t.

```
---------------------------------- MODULE work1 ----------------------------------
EXTENDS Integers,TLC,Sequences
CONSTANTS nb_process
process_x == 1..nb_process
(*
--algorithm work1
    variables R = [Rto \in process_x |-> FALSE] ; attente_message = <<>>; traitement_termine = [Rto \in
process_x |-> FALSE]
    fair process Q = nb_process+1
    variables tmp;
    begin Qp:
        while \neg \A P \in process_x : traitement_termine[P]  do
            traitement_message:
                await Len(attente_message)>0;
                tmp := Head(attente_message);
                attente_message := Tail(attente_message);
                R[tmp] := TRUE;
            attente_fin_traitement:
                await \neg R[tmp];
```

```
        end while;
    end process
    fair process P \in process_x
    begin
        SendMsg:
            attente_message := Append(attente_message, self);
        execution:
            await R[self];
            R[self]:=FALSE;
            traitement_termine[self]:=TRUE;
    end process
end algorithm;
*)
CONSTANT defaultInitValue
VARIABLES R, attente_message, traitement_termine, pc, tmp
vars == << R, attente_message, traitement_termine, pc, tmp >>
ProcSet == {nb_process+1} \cup (process_x)

Init == (* Global variables *)
        /\ R = [Rto \in process_x |-> FALSE]
        /\ attente_message = <<>>
        /\ traitement_termine = [Rto \in process_x |-> FALSE]
        (* Process Q *)
        /\ tmp = defaultInitValue
        /\ pc = [self \in ProcSet |-> CASE self = nb_process+1 -> "Qp"
                                       [] self \in process_x -> "SendMsg"]

Qp == /\ pc[nb_process+1] = "Qp"
      /\ IF \neg \A P \in process_x : traitement_termine[P]
            THEN /\ pc' = [pc EXCEPT ![nb_process+1] = "traitement_message"]
            ELSE /\ pc' = [pc EXCEPT ![nb_process+1] = "traitement_termine"]
      /\ UNCHANGED << R, attente_message, traitement_termine, tmp >>
traitement_message == /\ pc[nb_process+1] = "traitement_message"
                      /\ Len(attente_message)>0
                      /\ tmp' = Head(attente_message)
                      /\ attente_message' = Tail(attente_message)
                      /\ R' = [R EXCEPT ![tmp'] = TRUE]
                      /\ pc' = [pc EXCEPT ![nb_process+1] = "attente_fin_traitement"]
                      /\ traitement_termine' = traitement_termine
attente_fin_traitement == /\ pc[nb_process+1] = "attente_fin_traitement"
                          /\ \neg R[tmp]
                          /\ pc' = [pc EXCEPT ![nb_process+1] = "Qp"]
                          /\ UNCHANGED << R, attente_message, traitement_termine, tmp >>
Q == Qp \/ traitement_message \/ attente_fin_traitement
SendMsg(self) == /\ pc[self] = "SendMsg"
                 /\ attente_message' = Append(attente_message, self)
                 /\ pc' = [pc EXCEPT ![self] = "execution"]
                 /\ UNCHANGED << R, traitement_termine, tmp >>
execution(self) == /\ pc[self] = "execution"
                   /\ R[self]
                   /\ R' = [R EXCEPT ![self] = FALSE]
                   /\ traitement_termine' = [traitement_termine EXCEPT ![self] = TRUE]
                   /\ pc' = [pc EXCEPT ![self] = "traitement_termine"]
                   /\ UNCHANGED << attente_message, tmp >>
P(self) == SendMsg(self) \/ execution(self)
Terminating == /\ \A self \in ProcSet: pc[self] = "traitement_termine"
               /\ UNCHANGED vars
Next == Q
        \/ (\E self \in process_x: P(self))
        \/ Terminating
Spec == /\ Init /\ [][Next]_vars
        /\ WF_vars(Q)
        /\ \A self \in process_x : WF_vars(P(self))
Termination == <>(\A self \in ProcSet: pc[self] = "traitement_termine")
AllProcesstraitement_termine ==  <>(\A Pi \in process_x: traitement_termine[Pi])
plus_dun_utilisateur == \neg (\E x,y \in process_x: x /= y /\ R[x] /\ R[y] )
```

4 Algorithms for On-the-fly Garbage Collection (work2)

L'algorithme de work2 est un algorithme de garbage collection (ou ramasse-miettes en français) permettant le recyclage des zones mémoires inutilisées. Dans cet exemple, les zones mémoire sont représentées par des noeuds, et les zones mémoire allouées puis libérées sont considérées comme inaccessibles. L'algorithme fonctionne en deux temps :

- Un collecteur marque, à l'aide d'une couleur, tous les noeuds qui sont accessibles (en propageant la couleur à ses noeuds voisins) et permet donc d'identifier les noeuds "garbage" qui sont à ajouter à la liste Free.
- Le mutateur va ajouter les noeuds identifiés par le collecteur (c'est à dire ceux qui n'ont pas la bonne couleur) à la liste Free, afin qu'ils soient réutilisés.

Nous pouvons donc en déduire 4 actions différentes : colorer les racines (un noeud est choisi arbitrairement comme racine), puis propager la couleur (un noeud propage la couleur à ses fils). Ensuite vient la phase d'identification puis la phase de "collecte" des noeuds garbage pour les ajouter à la liste des noeuds libres.

Bien que les actions semblent être assez faciles de primes abords, nous avons rencontré de nombreuses diffultés à mettre en place l'algorithme, et notammant la traduction du langage ADA en langage TLA+ et PlusCal.
Nous avons éprouvé des difficultés quant à la manière de mettre en place les spécificités de l'algorithme, notamment sur la manière de déclarer les différents tableaux afin d'avoir accès aux bonnes valeurs.
Nous avons finalement écrit un code ressembalnt à l'algorithme de base, mais malheureusement non fonctionnel. Lors du lancement du Model Checker, sans spécifier les différents invariants du papier, nous obtenons une erreur, que nous n'avons pas réussi à régler. Nous vous présentons cependant le code partiel que nous avons pu produire dans le listing suivant.

```
---------------------- MODULE garbage1 ----------------------
EXTENDS Integers, Naturals, TLC
CONSTANT n


Index == 1..n
Sons == 1..n
Roots == 1..n

son == [ i \in 1..n |-> Index]
Hue == {"black","white"}
Node == {son}
M == [i \in 1..n |-> Node]
Free == [i \in 1..n |-> Node]
Color == [node \in M |-> Hue]



(*
--algorithm garbage {
variables   R;S;T;i=0;j= 0;k= 0;l= 0;I= 0;black_count= 0;old_black_count= 0;
    M[R]:= T;
    Color[M[R]]:= "black";

    while i \in Roots do
        Color[M[i]] := "black";
        i := i+1;
    end while;
    while j \in Index do
        if Color[M[j]] = "black" then
            while k \in Sons do
                Color[M[M[I[k]]]] := "black";
            end while;
        end if;
    end while;
    while l \in Index do
        if Color[M[l]] = "black" then
            black_count := black_count + 1;
        end if;
    end while;
    if black_count > old_black_count then
        old_black_count := black_count;
    end if;
    while I \in Index do
        if Color[M[I]] = "white" then
            Free[I] := M[I];
        else
            Color[M[I]] := "white";
        end if;
    end while;

}
end algorithm;


*)
\* BEGIN TRANSLATION
CONSTANT defaultInitValue
VARIABLES R, S, T, i, j, k, l, I, black_count, old_black_count, pc

vars == << R, S, T,  i, j, k, l, I, black_count, old_black_count, pc >>

Init == (* Global variables *)
        /\ R = defaultInitValue
        /\ S = defaultInitValue
        /\ T = defaultInitValue
        /\ pc = "Lbl_1"

Lbl_1 == /\ pc = "Lbl_1"
         /\ i' = 0
         /\ j' = 0
         /\ k' = 0
         /\ l' = 0
         /\ I' = 0
         /\ black_count' = 0
         /\ old_black_count' = 0
         /\ M' = [M EXCEPT ![R] = T]
```

```
          /\ Color' = [Color EXCEPT ![M[R]] = "black"]
          /\ pc' = "Lbl_2"
          /\ UNCHANGED << R, S, T >>

Lbl_2 == /\ pc = "Lbl_2"
          /\ IF i \in Roots
                THEN /\ Color' = [Color EXCEPT ![M[i]] = "black"]
                     /\ i' = i+1
                     /\ pc' = "Lbl_2"
                ELSE /\ pc' = "Lbl_3"
          /\ UNCHANGED << R, S, T >>

Lbl_3 == /\ pc = "Lbl_3"
          /\ IF j \in Index
                THEN /\ IF Color[M[j]] = "black"
                           THEN /\ pc' = "Lbl_4"
                           ELSE /\ pc' = "Lbl_3"
                ELSE /\ pc' = "Lbl_5"
          /\ UNCHANGED << R, S, T >>

Lbl_4 == /\ pc = "Lbl_4"
          /\ IF k \in Sons
                THEN /\ Color' = [Color EXCEPT ![M[M[I[k]]]] = "black"]
                     /\ pc' = "Lbl_4"
                ELSE /\ pc' = "Lbl_3"
          /\ UNCHANGED << R, S, T >>

Lbl_5 == /\ pc = "Lbl_5"
          /\ IF l \in Index
                THEN /\ IF Color[M[l]] = "black"
                           THEN /\ black_count' = black_count + 1
                           ELSE /\ TRUE
                     /\ pc' = "Lbl_5"
                ELSE /\ IF black_count > old_black_count
                           THEN /\ old_black_count' = black_count
                           ELSE /\ TRUE
                     /\ pc' = "Lbl_6"
          /\ UNCHANGED << R, S, T >>

Lbl_6 == /\ pc = "Lbl_6"
          /\ IF I \in Index
                THEN /\ IF Color[M[I]] = "white"
                           THEN /\ Free' = [Free EXCEPT ![I] = M[I]]
                           ELSE /\ Color' = [Color EXCEPT ![M[I]] = "white"]
                     /\ pc' = "Lbl_6"
                ELSE /\ pc' = "Done"
          /\ UNCHANGED << R, S, T >>

Next == Lbl_1 \/ Lbl_2 \/ Lbl_3 \/ Lbl_4 \/ Lbl_5 \/ Lbl_6
           \/ (* Disjunct to prevent deadlock on termination *)
              (pc = "Done" /\ UNCHANGED vars)

Spec == Init /\ [][Next]_vars

Termination == <>(pc = "Done")

\* END TRANSLATION
```

5 Etude de l'article

5.1 Principe de l'algorithme

5.1.1 Définitions

Un **graphe non orienté** G est la donnée d'un couple G = (S, A) tel que :

— S est un ensemble fini de sommets,

— A est un ensemble de couples non ordonnés de sommets $\{s_i, s_j\} \in S^2$

Une paire s_i, s_j est appelée une **arête**, et est représentée graphiquement par s_i-s_j. On dit que les sommets s_i et s_j sont adjacents. L'ensemble des sommets adjacents au sommet $s_i \in$ S est noté Adj(s_i) = $\{s_j \in$ S, $\{s_i, s_j\} \in$ A$\}$.

Un **arbre** est un graphe non orienté, acyclique (ne contient aucun cycle) et connexe (quels que soient les sommets u et v de S, il existe une chaîne reliant u à v).

Le **poids** d'un arbre est défini comme la somme des poids des arêtes qui le constitue.

L'arbre couvrant de poids minimum (MST) du graphe est un sous-ensemble de sommets connectés ensembles dont la somme des poids des arêtes est minimale.

Un **fragment** est un sous arbre du MST. Au début on considère chaque nœud comme un fragment et à la fin de l'algorithme le MST entier sera un fragment.

Une arête est **sortante** si un nœud adjacent se trouve dans le fragment et pas l'autre.

Le **noyau** est un sous-ensemble de sommets du graphe à la fois stable et dominant.

5.1.2 Algorithme

L'algorithme se base sur un graphe non orienté connecté avec N nœuds et E arêtes, avec un poids fini distinct pour chaque arête. L'algorithme sert à déterminer l'arbre couvrant de poids minimum (MST) du graphe. On suppose que chaque nœud connait initialement le poids des arêtes adjacentes. Tous les nœuds exécutent le même algorithme soit : envoyer des messages sur les arêtes adjacentes, attendre la réception de messages et traiter les messages. Les messages peuvent être transmis dans les deux directions d'une arête puisqu'il s'agit d'un graphe non orienté. Les messages arrivent après un délai imprévisible, mais fini, sans erreur et dans l'ordre. Quand les nœuds ont fini leur algorithme ils savent quelles arêtes adjacentes se trouvent dans l'arbre et quelle arête mène au noyau.

Au début les nœuds sont endormis et soit spontanément soit par une réception de messages, certains nœuds se réveillent et commencent leur algorithme.

Pour trouver le MST, chaque nœud va chercher son arête sortante de poids minimal et va essayer de se combiner avec le nœud adjacent pour former un fragment. Quand tous les nœuds sont dans le même fragment, cela signifiera que le MST est trouvé et que c'est le fragment.

Quand un nœud se réveille il va chercher son arête sortante de poids minimal. Nous considérons que chaque nœud est un fragment de niveau 0. Une fois que le fragment a trouvé son arête sortante de poids minimal il va tenter de se combiner avec le fragment à l'autre extrémité de l'arête grâce à l'envoi d'un message *Connect*.

La combinaison des deux fragments est dépendante des niveaux des deux fragments. Un fragment est au niveau 0 quand il n'est composé que d'un nœud. Supposons que nous ayons un fragment F de niveau $L \geq 0$ et que de l'autre côté de l'arête sortante nous ayons un fragment F' de niveau L'. Trois cas sont possibles :

— L = L', les deux fragments ont la même arête sortante de poids minimal, ils peuvent donc se combiner et l'arête sortante sera le nouveau noyau du fragment. Le nouveau fragment aura un niveau L+1. Exemple : On a F et F' de niveau 1, l'arête sortante de poids minimale de F est l'arête reliant les nœuds 2 et 3. Comme les 2 fragments ont le même niveau,

ils vont pouvoir fusionner pour former un fragment de niveau 2, dont les nœuds de l'arête 2-3 seront le nouveau noyau, voir figure 1.

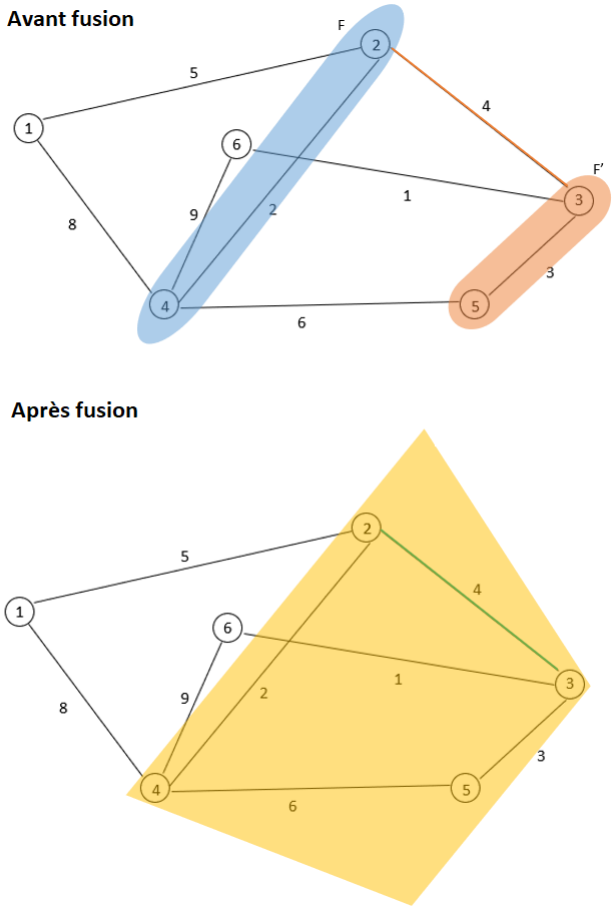

FIGURE 1 – Combinaison quand L=L'

— $L < L'$, alors F est absorbé par F' et le niveau du nouveau fragment est L'. Exemple : F est de niveau 1 et F' est de niveau 2. Tous les deux ont la même arête sortante 6-3. F et F' vont donc fusionner pour former un nouveau fragment de niveau 2, voir figure 2.

— $L > L'$, F va attendre que F' ait un niveau supérieur ou égal au sien pour pouvoir se combiner.

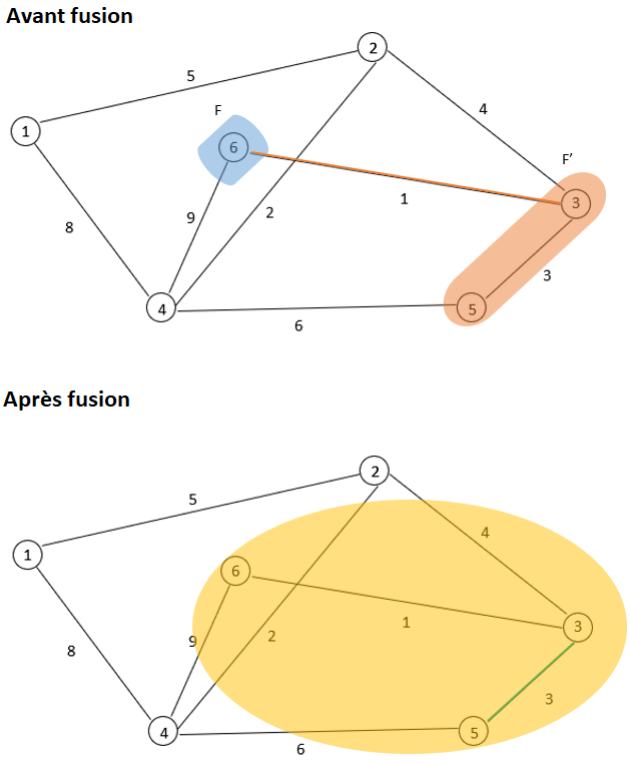

FIGURE 2 – Combinaison quand $L < L'$

Chaque nœud peut avoir 3 états : *Sleeping* (le nœud est en « sommeil »), *Found* (le nœud a trouvé une arête sortante), *Find* (le nœud participe à la recherche de l'arête sortante de poids minimal).
Chaque arête peut avoir 3 états : *Branch* (l'arête est une branche dans le fragment actuel), *Basic* (l'arête n'est ni *Branch* ni *Rejected*), *Rejected* (l'arête n'est pas une branche du fragment, mais a été utilisée pour joindre deux nœuds).

Il y a différents cas pour trouver l'arête sortante de poids minimal d'un fragment :

— Pour un fragment de niveau 0 : Le nœud est à l'état *Sleeping* et lors de son réveil il va choisir son arête adjacente de poids minimal et la marquer comme *Branch* du MST. Il va envoyer un message *Connect*

sur cette arête et passer à l'état *Found* pour attendre une réponse du nœud relié à l'arête marquée, voir la figure 3 pour une meilleure compréhension.

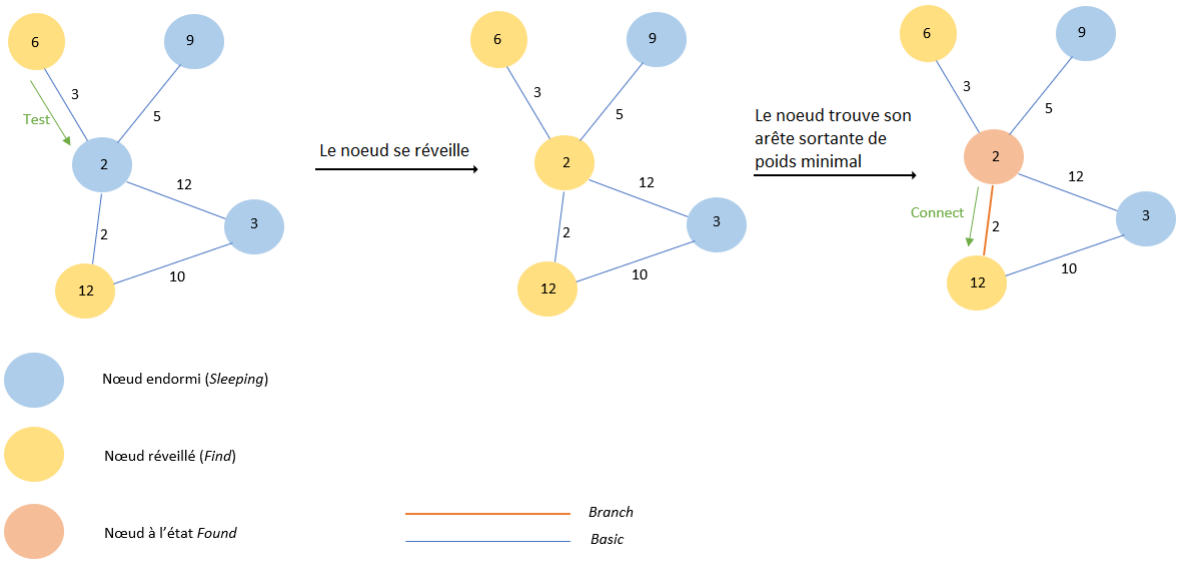

FIGURE 3 – Réveil d'un noeud

— Pour un fragment de niveau L formé par la combinaison de deux fragments de niveau $L-1$. L'arête sortante a été définie comme le nouveau noyau, son poids va être utilisé comme identité du fragment. Les 2 nœuds adjacents au noyau vont émettre un message *Initiate* aux autres nœuds. Ce message contient la nouvelle identité du fragment, son nouveau niveau L et l'argument Find qui va mettre les nœuds dans l'état *Find*. Le message sera relayé par tous les nœuds du fragment et si d'autres fragments de niveau $L-1$ attendaient de se connecter aux nœuds du nouveau fragment L, le message leur sera transmis. Quand un nœud reçoit le message *Initiate*, il va se mettre à chercher son arête sortante. Pour la trouver, il va classer chacune de ses arêtes selon trois états : *Branch* si l'arête est une branche dans le fragment actuel, *Rejected* si l'arête n'est pas une branche du fragment mais a été utilisée pour joindre deux nœuds, *Basic* si l'arête n'est ni *Branch* ni *Rejected*. Le nœud va commencer par choisir l'arête *Basic* de poids minimal et envoie un message *Test* dessus qui contient l'identité et le niveau du

fragment. Lorsqu'un nœud de niveau L' reçoit un message *Test*, il va vérifier si son identité correspond à celle du message. Si les identités concordent, le nœud enverra un message *Reject* au nœud émetteur car ils font parti du même fragment. Les deux nœuds classifieront l'arête comme *Rejected* et le nœud émetteur testera l'arête *Basic* suivante de poids minimal. Si les 2 nœuds n'ont pas la même identité et que $L \leq L'$ alors le nœud récepteur envoi le message *Accepted* pour signifier que l'arête est une arête sortante du fragment. Si $L > L'$ alors le nœud récepteur du message *Test* va attendre que le niveau de son fragment augmente avant de répondre, voir figure 4.

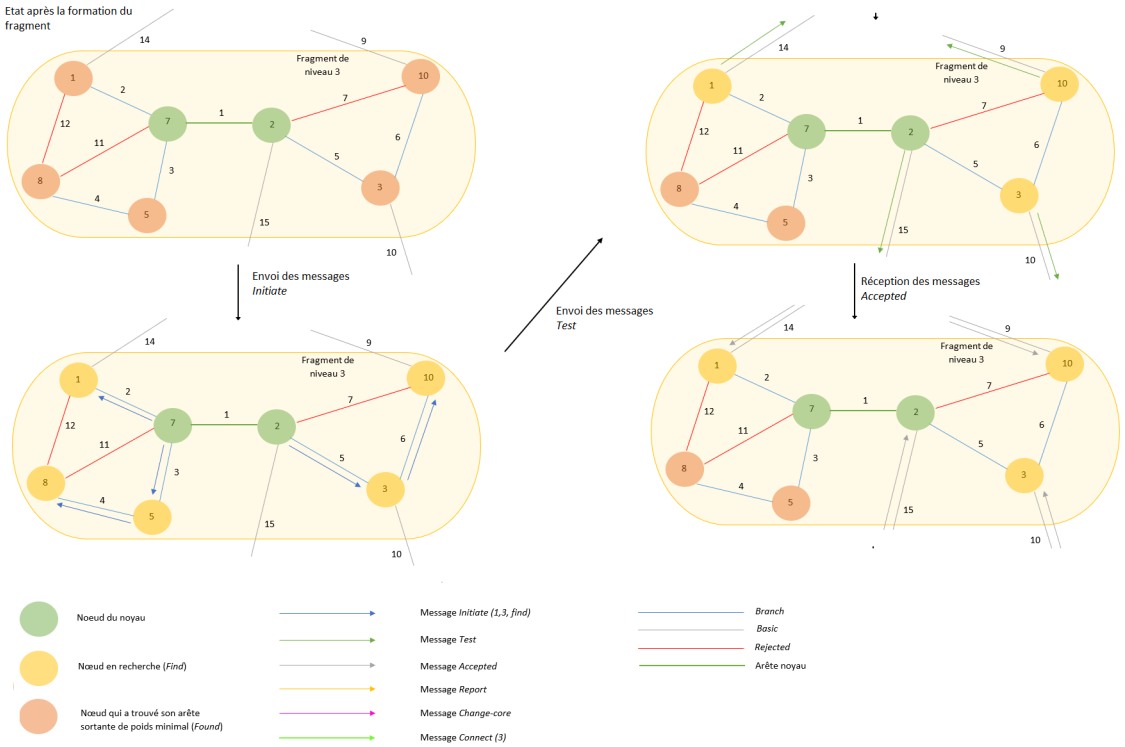

FIGURE 4 – Trouver les arêtes sortantes

Il est important de noter que le niveau d'un fragment change seulement quand son niveau augmente.
A ce stade chaque nœud a trouvé son arête sortante de poids minimal. Il

faut maintenant faire monter l'information pour déterminer quelle est l'arête sortante de poids minimal du fragment. Plusieurs cas sont possibles :

— Aucun nœud n'a d'arête sortante, l'algorithme est fini et le fragment est le MST.

— Au moins un nœud a une arête sortante. Chaque nœud feuille, c'est-à-dire adjacent à une seule branche du fragment, va envoyer le message *Report(W)* sur sa branche entrante. W est le poids de l'arête sortante de poids minimal trouvé ou infini s'il n'existe pas d'arête sortante. Chaque nœud non-feuille va attendre de recevoir les messages *Report* des branches sortantes. Il va envoyer le message *Report(W)* avec W comme étant l'arête de poids minimal trouvé. Quand un nœud envoie le message *Report* il passe à l'état *Found*, ce qui indique qu'il a fini son rôle dans la recherche de l'arête sortante de poids minimal. Finalement, les deux arêtes liées au noyau vont envoyer au noyau leur rapport et il sera facile de déterminer le chemin pour accéder à l'arête sortante de poids minimal. Le message *Change-core* sera envoyé sur chaque branche du chemin et l'arête entrante pour chacun de ses nœuds est modifiée pour correspondre à la meilleure arête. Lorsque le message *Change-core* atteint le nœud avec l'arête sortante de poids minimal, les arêtes entrantes forment un arbre dont la racine est le nœud. Enfin ce nœud va envoyer un message *Connect(L)* sur l'arête sortante de poids minimum où L est le niveau du fragment, voir figure 5.

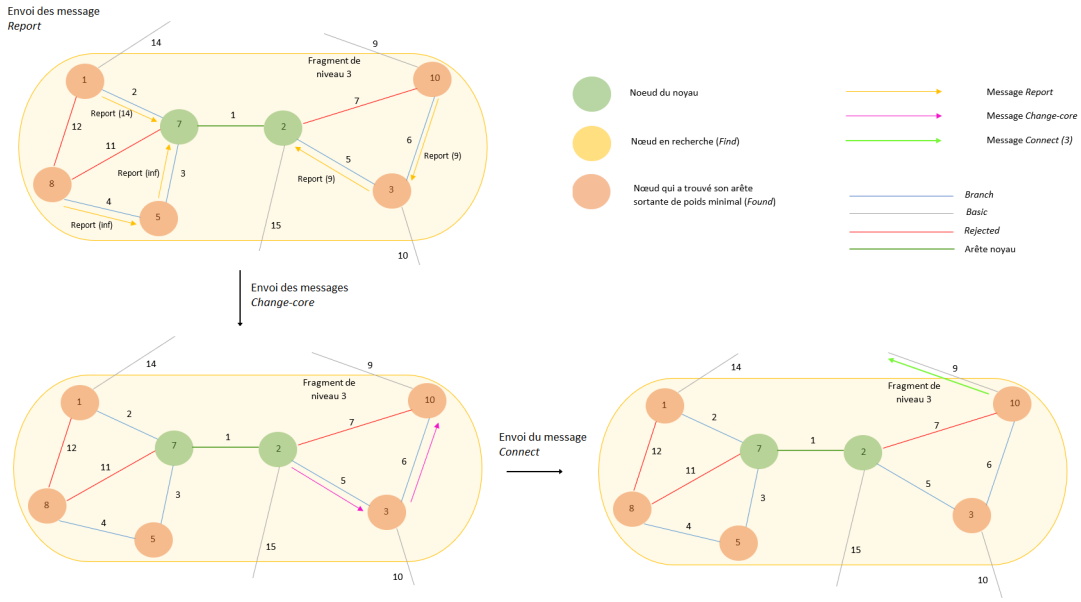

FIGURE 5 – Remonter l'information

5.2 Preuves

5.2.1 Borne supérieure du nombre de message

Rappel : N est le nombre de nœuds.

Chaque message comprend au plus un poids d'arête, un entier entre 0 et $\log 2N$ et trois bits supplémentaires. On sait qu'une arête ne peut être rejetée au plus qu'une seule fois et qu'un rejet utilise 2 messages (un pour chaque nœud). On a donc maximum $2E$ messages de rejet.

Un nœud à un niveau donné (or le niveau 0 et le dernier niveau) peut recevoir maximum un message *Initiate* et un message *Accepted*. Il peut transmettre au plus un message *Report*, un message *Change-core* ou un message *Connect*.

$\log 2N$ est la limite du niveau le plus élevé, donc un nœud peut passer par au plus $\log N - 1$ niveaux si on ne compte pas le niveau 0 et le dernier. Donc chaque nœud peut envoyer ou recevoir $5N(\log N - 1)$ messages.

Au niveau 0, chaque nœud peut recevoir au maximum un message *Initiate* et envoyer un message *Connect*. Au dernier niveau, chaque nœud peut envoyer au plus un message *Report*. Ce qui fait au maximum $3N$ que l'on va borner à $5N$.
On a donc une borne supérieure de $5N * (\log N - 1) + 5N + 2E = 5N * \log N + 2E$.

A noter : Si le nombre de nœuds est initialement inconnu, alors on ne peut pas trouver le MST avec moins de E messages.

5.2.2 Absence de blocage

Considérons l'ensemble des fragments existants à un moment de l'algorithme et qu'on exclut les nœuds de niveau 0 qui sont à l'état *Sleeping* et en supposant que chaque fragment a une arête sortante de poids minimum. Si on prend le fragment de niveau le plus bas avec l'arête sortante de poids minimal, alors si ce fragment envoie un message *Test* cela réveillera un fragment de niveau 0 endormi ou il recevra une réponse immédiatement. Pareil pour un message *Connect* envoyé par ce fragment, il réveillera un fragment, ou sera absorbé par un fragment supérieur ou fusionnera avec un nœud du même niveau pour former un fragment de niveau supérieur.

Comme l'état du système était supposé arbitraire, cela prouve que les blocages n'existent pas.

5.3 Extensions possibles

Plusieurs extensions sont possibles, par exemple si les arêtes n'ont pas de poids distincts, mais que tous les nœuds sont distincts et peuvent être ordonnés, l'algorithme pourra s'appliquer en ajoutant au poids des arêtes les identités des nœuds en commençant par le nœud de rang inférieur. Cependant, si les arêtes ne sont pas distinctes et que les nœuds ne sont pas distincts, on ne peut pas appliquer ce genre d'algorithme.

6 Difficultés rencontrées

6.1 Lecture de l'article

Lire et décrire un article peut paraître simple de prime abord, mais l'article a analyser est en anglais et utilise beaucoup de notions et un vocabulaire spécifiques qu'il faut réussir à s'approprier pour réussir à appréhender l'article. De plus, nous avons trouvé que même en surmontant la langue et le vocabulaire spécifique, le texte reste difficile à comprendre. Nous avons fait de notre mieux pour expliquer le principe de l'algorithme et ce que nous en avons compris mais cela a été long et fastidieux. La réalisation de schéma, même si elle aide à la compréhension, a été très longue et difficile aussi.

6.2 Modélisation TLA+ et PlusCal

Nous avons éprouvé de grandes difficultés, comme il a été évoqué plus tôt dans ce rapport. Même si nous avons eu l'occasion d'utiliser de nombreux cas d'exemples en cours, mais aussi des guides d'utilisation trouvé sur Internet, nous n'avons pas réussi à mettre en place des travaux et des algorithmes fonctionnels et capables d'être analysés par l'outil.

7 Conclusion

En conclusion, nous ne pouvons bien évidemment pas dire que ce projet fut un succès. Cependant, nous ne le considérons pas non plus comme un échec à proprement parler. Premièrement, il nous a quand même permi d'approfondir des notions de systèmes distribués vu en cours, et de développer une certaine réflexion autours d'un sujet complexe, même si nous n'avons pas réussi à mener à bien les différentes vérifications.
Dans un second temps, nous sommes fiers du travail de vulgarisation et de compréhension fait au cours de l'étude de l'article, qui, comme nous l'avons vu, n'était pas un travail facile.