Introduction à l'algorithmique répartie et à l'auto-stabilisation

Cours de Joffroy Beauquier - Notes de Philippe Gambette

21 avril 2006 - version 2.0

1 A propos de ces notes

Ce qui suit provient de notes prises au cours d'algorithmique distribuée et de résistance aux défaillances du Master Parisien de Recherche en Informatique (MPRI), à l'automne 2005[1]. Merci à Quang Cuong Phạm, qui a permis d'améliorer le style LaTeX de ce document, et Charly Laporte, qui a fourni ses notes de cours en complément des miennes, et a proposé plusieurs corrections et améliorations au texte ci-dessous. Ce cours pourra être complété par les cours d'algorithmique répartie en anglais, disponibles en ligne, de Nancy Lynch [LPS92] ou Hagit Attiya [Att94]. En ce qui concerne plus précisément l'auto-stabilisation, on pourra se reporter à des slides de Joffroy Beauquier[2], à l'article de synthèse en anglais par Marco Schneider [Sch93], et à la bibliographie de la page `http://www.cs.uiowa.edu/ftp/selfstab/bibliography/stabib.html`.

Contactez-moi à philippe.gambette@laposte.net pour signaler des erreurs éventuelles ou obtenir toute information sur ces notes.

2 Principes et buts de l'algorithmique distribuée

Si l'on dispose d'un grand nombre d'ordinateurs, répartis géographiquement sur un vaste territoire et reliés entre eux par un réseau, peut-on utiliser simultanément la puissance de toutes ces machines pour effectuer un même calcul global ?

Réunir un groupe d'ordinateurs reliés en réseau afin de former un système de calcul équivalent à une machine vectorielle de très grande puissance est l'objectif de l'algorithmique répartie (ou algorithmique distribuée). Pour l'utilisateur d'un tel système, celui-ci doit être vu comme une seule machine, *mono-utilisateur* et *mono-processeur*, de grande puissance.

Une première ambition de l'algorithmique répartie consiste à pouvoir considérer qu'aucun ordinateur réparti n'a un rôle différent des autres, c'est à dire qu'il n'y a pas de serveurs. Ainsi, tout ordinateur défaillant peut être remplacé par n'importe quel autre ordinateur opérationnel. Pour effectuer de tels remplacements, il convient de disposer d'algorithmes qui résistent aux défaillances et qui sont capables de maintenir la spécification générale du système même si plusieurs ordinateurs ont un comportement incohérent.

Les algorithmes auto-stabilisants sont des algorithmes tolérants aux défaillances : malgré une remise à zéro (ou à des valeurs aléatoires) de toutes les variables du système (sans effectuer de changement de code), les algorithmes convergent, au bout d'un certain temps, vers leur spécification.

3 Modèles de communication

Définir un problème d'algorithmique distribuée demande de modéliser les échanges d'informations entre processus. Il faut décider si chaque processus peut être actif et envoyer ou recevoir des informations quand il le souhaite (cadre *asynchrone*) ou si un ordonnanceur décide des périodes d'activité des processus (cadre *synchrone*). Il faut aussi choisir un modèle de communication :

– lecture d'états : en une action (indivisible), un processus lit son état et celui de ses voisins.
– registres : partagés deux à deux, selon la figure 1.

[1] ou plus précisément aux 4 séances de trois heures de Joffroy Beauquier : 25 septembre, 3, 10, et 17 octobre 2005
[2] `http://www.liafa.jussieu.fr/ecole-de-printemps/Self-stab-english.ppt`

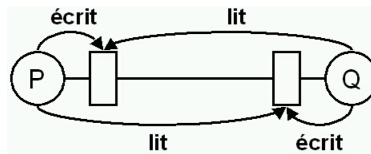

FIG. 1 – Modèle de communication par registres.

4 Un premier algorithme distribué : l'élection de leader

4.1 Sur un anneau unidirectionnel

On suppose les ordinateurs reliés en un *anneau unidirectionnel* c'est à dire un réseau circulaire orienté dans lequel chaque machine ne peut communiquer qu'avec son voisin suivant, dans le sens de l'orientation.

Le nombre de processus dans l'anneau est inconnu, et la communication se fait par messages (transmission asynchrone des messages dans un seul sens, sans message perdu).

Chaque processus a un numéro propre. On cherche à minimiser le nombre total de messages émis pour élire un leader, ce qui correspondra à la *terminaison* : quand tous les processus arrêtent de travailler, sur l'état *élu* ou *battu*.

Algorithme 1 (Le Lann, 1977 [Lan77] - Chang, Roberts, 1979 [CR79]).
Chaque processus transmet son numéro à son voisin de droite. S'il reçoit un numéro de voisin de gauche, il le transmet à celui de droite seulement si ce numéro est supérieur au sien. Sinon, il transmet le sien. Le but de l'algorithme est donc que le processus de numéro maximal devienne leader.

Il y aura $O(n^2)$ messages transmis au total.

4.2 Sur un anneau bidirectionnel

Dans un anneau bidirectionnel, un ordinateur peut communiquer avec ses deux voisins adjacents : suivant et précédent. Le principe est le même, en divisant par deux le nombre de processus vivants, tous les deux tours, pour arriver à un nombre de messages émis en $O(n \log(n))$.

Algorithme 2.
Le premier envoi se fait dans le sens positif : si un processus reçoit un numéro supérieur au sien, il meurt. Le deuxième envoi se fait dans le sens négatif avec la même action. Les deux types d'envois alternent jusqu'à ce qu'un des processus (le plus grand) reçoive sa propre valeur.

La démonstration se fait en trouvant pour chaque processus survivant une de ses victimes au cours des deux tours précédents.

5 Défaillances transitoires et auto-stabilisation

Les *défaillances transitoires* placent le système dans une configuration arbitraire : on doit considérer que toute la mémoire, les registres et tous les canaux ont été corrompus. En revanche, le code est incorruptible.

L'*approche pessimiste* consiste à s'assurer que le système ne dévie jamais de la spécification : elle nécessite des contrôles permanents et donc un coût élevé. L'*approche optimiste* consiste à supposer que les défaillances sont rares, et accepter que le système dévie un peu. C'est ce que l'on va définir plus clairement avec le concept d'*auto-stabilisation*.

Un *système* est un couple $(C, \to)$ où C est l'ensemble des configurations et $\to$ une relation binaire sur C. Une *exécution* de S est une suite maximale $E = (\gamma_1, \gamma_2, \ldots)$ telle que $\forall i, \gamma_i \to \gamma_{i+1}$.

Un système S *se stabilise* pour une *spécification* (ou *post-condition*) P (parfois notée ψ) ssi il existe un sous-ensemble L de C, l'*ensemble des configurations légitimes*, illustré en figure 2, qui vérifie les deux propriétés suivantes :

Convergence : toute exécution atteint une configuration de L.

Correction : toute exécution à partir d'une une configuration de L satisfait P.

Un système est *pseudo-stabilisant* si toute exécution a un suffixe qui satisfait la spécification P.

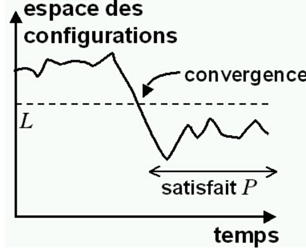

FIG. 2 – Stabilisation d'un système vers L, l'ensemble des configurations légitimes.

Corollaire 1. *Si S auto-stabilisant pour P, alors il est pseudo-stabilisant.*

Proposition 1. *Supposons que :*
- *toutes les configurations terminales sont dans L.*
- *il existe une fonction $f : C \to w$, où w est un ensemble bien fondé*[3]*, telle que pour toute transition $\gamma \to \delta, f(\gamma) > f(\delta)$ ou $\delta \in L$.*

Alors L satisfait la convergence.

6 Le token-ring auto-stabilisant de Dijkstra

On se place dans le modèle à lecture d'états, sur un anneau de taille N. La spécification du problème est qu'un unique jeton circule indéfiniment et de manière équitable. Ceci répond à une problématique d'exclusion mutuelle : chaque machine a de temps en temps besoin d'accéder, seule, à une ressource partagée (ceci correspond à une *section critique* de son code). Le jeton de notre modélisation correspond donc à la machine qui se trouve en section critique.

> **Algorithme 3** (Dijkstra [Dij65]).
> Soit $k \in \mathbb{N}^*$. On initialise pour chaque machine i un σ_i pris au hasard dans $[\![0, k-1]\!]$
> On définit un *jeton* (*token*, en anglais) ainsi :
> – P_0 tient le jeton ssi $\sigma_0 = \sigma_{n-1}$.
> – $P_{i \neq 0}$ tient le jeton ssi $\sigma_i \neq \sigma_{i-1}$.
> Un des processus, P_0, est distingué, mais les autres exécutent le même code.
> – Pour P_0 : $\sigma_0 = \sigma_{n-1} \Rightarrow \sigma_0 = \sigma_0 + 1 \mod k$.
> – Pour $P_{i \neq 0}$: $\sigma_i \neq \sigma_{i-1} \Rightarrow \sigma_i = \sigma_{i-1}$.

Idée de l'algorithme de Dijkstra[4] : on considère que les σ_i sont des hauteurs d'eau, toutes inférieures à k. Un processus (différent de P_0) a le jeton si l'eau a un niveau différent avant lui : dans ce cas, l'algorithme équilibre le niveau d'eau pour ce processus, et la différence de niveau est décalée. Ainsi, l'algorithme permet de déplacer un front d'onde (une vague) le long de l'anneau, jusqu'à P_0 où une nouvelle vague est créée.

Proposition 2. *Il n'existe pas de configuration sans jeton.*

Démonstration. Si aucun des P_i n'a de jeton, alors $\sigma_0 = \sigma_1 = \ldots = \sigma_{n-1}$, ce qui est invalide. □

Proposition 3. *Il n'y a pas de configuration terminale.*

Démonstration. Soit L, l'ensemble des configurations dans lequelles il n'y a qu'un jeton. Toute activation d'une règle fait passer le jeton d'une machine à la suivante. □

Proposition 4. *Le système converge vers L.*

Démonstration. Toute exécution est infinie, elle contient une infinité de configurations où il y a un seul jeton, tenu par P_0.

Soit $F = \sum_{i \in S}(N - i)$ (où $S = \{i/i \neq 0$ et P_i a un jeton$\}$), la somme des distances à P_0 des processus à 1 jeton (dans le sens de circulation de la vague).

A chaque pas n'impliquant pas P_0, cette valeur F diminue de 1, voire plus (dans le cas où un jeton disparaît). Donc le nombre de pas consécutifs ne donnant pas le jeton à P_0 est borné par F, donc par $\frac{N(N-1)}{2}$. Ainsi, toute exécution contient bien une infinité de configurations où $F = 0$, c'est à dire le jeton est tenu par P_0. □

[3] ensemble *bien fondé* : il n'y existe pas de suite infinie strictement décroissante.
[4] http://sirac.imag.fr/~krakowia/Enseignement/M2R-SL/SR/Flips/algo-repartis.pdf

7 Algorithme de passage de messages

L'objectif est de calculer la taille d'un anneau bidirectionnel. Chacun a son numéro identificateur.

Algorithme 4 (naïf)**.**
Chaque processus envoie à son voisin une liste avec en tête son identificateur, et en queue, la liste qu'il a reçue au tour précédent de son autre voisin. Lorsqu'il reçoit son identificateur en tête de liste, la taille de la liste lui donne la taille de l'anneau.

Algorithme 5 (auto-stabilisant)**.**
On reprend l'algorithme précédent, en éliminant les messages corrompus (résultant d'une mauvaise initialisation) de la façon suivante : si un processus détecte son identificateur qui ne se trouve pas en tête de liste, alors il ne passe pas le message.

Algorithme 6 (résistant à une panne crash)**.**
Pour tout message envoyé dans un sens, on envoie le même dans l'autre sens. Ainsi, si une panne crash affecte l'une des machines, les messages contourneront cette machine en tournant autour de l'anneau dans l'autre sens.

Proposition 5. *Il n'existe pas d'algorithme à la fois auto-stabilisant et résistant à une panne crash pour ce problème.*

8 Orientation d'un anneau uniforme

On se place sur un anneau *uniforme* : les processus exécutent le même code, et l'anneau est *anonyme*, c'est à dire que les processus n'ont pas de numéro d'identification. Chaque processus étiquette ses deux liens : *pred* et *succ*, avec $pred \neq succ$.

Post-condition ψ : pour tout processus p et q, $pred(p) = q \Leftrightarrow succ(q) = p$.

On se place en mode de communication par registres : p, au lieu pq, associe un registre lpq de type $\{pred, succ\}$. Ainsi, le sens de l'anneau est localement défini de *pred* vers *succ*, et un processus peut savoir ce que son voisin pense de son orientation.

Algorithme 7 (Israeli et Jalfon, 1992 [IJ93])**.**
Chaque processus, en plus des deux registres, a une variable de type :
- R : receive
- S : send
- I : idle

Un processus *tient un jeton* :
- s'il est dans l'état S.
- s'il est dans l'état R et que son prédécesseur :
  - n'est pas dans l'état S.
  - est dans l'état S mais ne pointe pas vers lui (ne le considère pas comme son successeur).

$S_p \in \{S, R, I\}$
- $\{S_p = I$ et $lpq = succ$ et $S_q = R$ et $lqp = pred\} \rightarrow S_p := R$; si $lpq = succ$ alors *flip* (on échange les étiquettes *succ* et *pred*).
- $\{S_p = S$ et $lpq = succ$ et $S_q = R$ et $lqp = pred)\} \rightarrow S_p := I$ (effet : transmission de jeton).
- $\{S_p = R$ et $lpq = pred$ et non$(S_q = S$ et $lqp = succ)\} \rightarrow S_p := S$.
- $\{S_p = S_q = S$ et $lpq = lqp = succ\} \rightarrow S_p := R$; flip.
- $\{S_p = S_q = S$ et $lpq = lqp = succ\} \rightarrow S_p := S$.

Idée de l'algorithme d'Israeli et Jalfon : les processus envoient des jetons, dans le bon sens, espèrent-ils, qui disparaissent quand ils se rencontrent.

Récapitulatif des actions

Actions	q	p	p
(1)	$\overrightarrow{S}$	I	$\overrightarrow{R}$
(2)	$\overleftarrow{R}$	$\overleftarrow{S}$	$\overleftarrow{I}$
(3)	$\neg\overrightarrow{S}$	$\overrightarrow{R}$	$\overrightarrow{S}$
(4)	$\overrightarrow{S}$	$\overleftarrow{S}$	$\overrightarrow{R}$
(5)	$\overrightarrow{I}$	$\overleftarrow{I}$	$\overleftarrow{S}$

Enchaînement des actions
– Après (1), le prochain pas du processus est (3).
– Après (3) : (2) ou (4).
– Après (2) : (1) ou (5).
– Après (4) : (3).
– Après (5) : (2) ou (4).
Mouvements des jetons

Actions	q	p	$\to$	q	p
(1)	$\bullet$	$\times$	$\to$	$\bullet$	$\times$
(2)	$\times$	$\bullet$	$\to$	$\bullet$	$\times$
(3)	?	$\bullet$	$\to$	$=$	$\times$
(4)	$\bullet$	$\bullet$	$\to$	$\bullet$	$\times$
(5)	$\times$	$\times$	$\to$	$\times$	$\bullet$

Une configuration est légitime ssi tous les processeurs pointent dans la même direction.

Lemme 1. *L est fermé et $\gamma \in L \Rightarrow \psi(\gamma)$*

Démonstration. Les seules règles applicables sont (1), (2), et (3), et aucune des trois ne modifie l'orientation de p. □

Lemme 2. *Une configuration terminale est légitime.*

Démonstration. Soit γ une configuration terminale. Aucun processus n'est dans l'état R (sinon on peut appliquer (2) ou (3)). Si γ n'est pas orienté alors deux processus pointent l'un vers l'autre :
– si l'un des deux est dans l'état I, alors l'autre est soit dans l'état S et on peut appliquer (1), soit dans l'état I et alors on applique (5).
– sinon, les deux processus pointant l'un vers l'autre sont dans l'état S, et on applique (4).

□

Proposition 6. *L'algorithme converge vers les états légitimes.*

Démonstration. La création du jeton n'intervient qu'avec la règle (5), et aucune des règles (1), (2), (3), (4), ne peut créer le cadre pour l'exécution de la règle (5), donc les cas où (5) est exécutée ne proviennent que d'une mauvaise initialisation. Le nombre de jetons créés est donc borné par N, qui dépend de l'état initial. Si un jeton parcourt une distance N, l'anneau est orienté. Donc le nombre de pas où un jeton se déplace sans qu'une configuration légitime soit atteinte est au plus $N(N-1)$.

Le nombre total de pas avant stabilisation est donc en $O(N^2)$. □

9 Couplage maximal d'un graphe

Un *couplage*(*matching* en anglais) d'un graphe est un ensemble d'arêtes non adjacentes, c'est à dire telles que chaque sommet du graphe appartient au plus à une arête. Il est *maximal* s'il ne peut être augmenté, comme celui représenté en figure 3.

On se place dans le modèle à lecture d'états, où chaque processus a une variable $pref(p)$ dont la valeur est dans $voisins \cup \emptyset$.

On définit tout d'abord un peu de vocabulaire qui nous sera utile dans les démonstrations. Si p a choisi q, alors p :
– *attend* si q n'a pas choisi (état *wait*).
– est *apparié* si q a choisi p (état *match*).
– est *chaînant* si q a choisi un autre que p (état *chain*).
Si p n'a pas choisi, alors p :
– est *mort* si tous ses voisins sont appariés (état *dead*).

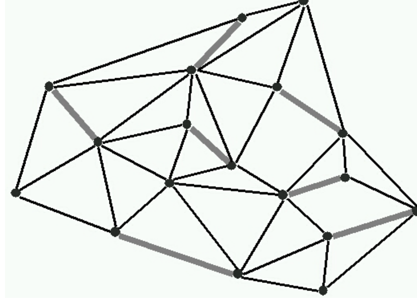

FIG. 3 – L'ensemble des arêtes grises constitue un couplage maximal du graphe.

– est *libre* dans le cas contraire (état $free$).

Les situations suivantes sont mutuellement exclusives :

– $wait(p) = \{pref(p) = q \in voisins$ et $pref(q) = \emptyset\}$

– $match(p) = \{pref(p) = q$ et $pref(q) = p\}$

– $chain(p) = \{pref(p) = q$ et $pref(q) = r \neq p\}$

– $dead(p) = \{pref(p) = \emptyset$ et $\forall q \in voisins, match(q)\}$

– $free(p) = \{pref(p) = \emptyset$ et $\exists q / \neg match(q)\}$

La post-condition est $\psi = \forall p, \big(match(p) \vee dead(p)\big)$.

Proposition 7. *Si ψ est vraie alors l'ensemble $M = \big\{\big(p, pref(p)\big)/pref(p) \neq \emptyset\big\}$ est un couplage maximal.*

Démonstration. D'abord que ψ assure que M est un couplage, puisque p n'appartient à aucune arête s'il est dans l'état *dead* et à une seule s'il est dans l'état *match*. Montrons alors que M est maximal. Supposons par l'absurde qu'on peut trouver une arête (p, q) tels que $M \cup \{(p, q)\}$ est un couplage. M ne contient aucune arête incidente à p. Ceci entraîne $dead(p)$ ou $free(p)$ et d'après ψ, $dead(p)$. Mais $dead(p)$ entraîne $match(q)$. q est donc présent dans 2 arêtes : absurde ! □

Algorithme 8 (Hsu et Huang [HH92])**.**
$pref(p) := voisins \cup \emptyset$.
$Match, M_p : \{pref(p) = \emptyset$ et $pref(q) = p\} \rightarrow pref(p) := q$.
$Select, S_p : \{pref(p) = \emptyset$ et $\forall r \in voisins(p), pref(r) \neq p$ et $pref(q) = \emptyset\} \rightarrow pref(p) := q$.
$Unchain, U_p : \{pref(p) = q$ et $pref(q) \neq p$ et $pref(q) \neq \emptyset\} \rightarrow pref(p) := \emptyset$.

Lemme 3. *La configuration γ est terminale ssi $\psi(\gamma)$ est vraie.*

Démonstration. $\Leftarrow$: L'action M_p n'est déclenchable que si q est en attente. L'action S_p n'est déclenchable que si q est libre. L'action U_p n'est déclenchable que si q est chaînant. Donc $\psi(\gamma)$ entraîne qu'aucune action n'est déclenchable dans γ.

$\Rightarrow$: Supposons que $\psi(\gamma)$ est faux. Alors il existe p qui est soit chaînant, soit libre, soit en attente.

- Si p est chaînant, U_p est déclenchable.
- Si p est en attente de q, M_q est déclenchable.
- Si p est libre, il existe un de ses voisins q non apparié, q n'est pas mort donc :
  - Si q est en attente, M_k déclenchable pour un des voisins k de q.
  - Si q est chaînant, U_q est déclenchable.
  - Si q est libre, S_p ou S_q sont déclenchables.

Ainsi, si $\psi(\gamma)$ est faux, alors γ n'est pas terminale. □

Lemme 4 (convergence [Tel94])**.** *L'algorithme atteint une configuration terminale en un nombre de pas au pire en $O(N^2)$, N étant le nombre de processus.*

Démonstration. Soit F telle que : $F(\gamma) = (c + f + w, 2c + f)$, où c est le nombre de processus chaînants, f le nombre de processus libres, et w le nombre de ceux en attente. On va alors démontrer que F est strictement décroissante pour l'ordre lexicographique.

Pour M_p : p est libre et q est en attente. Résultat : p et q appariés, et la première composante de $F(\gamma)$ a diminué d'au moins 2.

Pour S_p : p, libre, passe en attente. La première composante est donc inchangée et la seconde diminue de 1.

Pour U_p : p, chaînant, devient libre ou mort. F est là encore décroissante.

Or $c+l+w \leq N$ et $2c+l$ est aussi en $O(N)$, donc le nombre de valeurs possibles pour $F(\gamma)$ est en $O(N^2)$. □

On peut même vérifier que l'algorithme de Hsu et Huang pour le couplage maximal se stabilise après au plus $2m+N$ changements, m étant le nombre d'arêtes [HJS01].

10 Composition et auto-stabilisation

L'objectif est de construire de manière auto-stabilisatrice une *coloration* de graphe planaire en 6 couleurs, en commençant par construire une orientation acyclique du graphe, puis en exploitant les informations de l'orientation acyclique pour colorier le graphe. On utilisera alors le fait que la composition de deux algorithmes auto-stabilisants est auto-stabilisante.

Soient deux systèmes S_1 et S_2 tels que toute variable écrite par S_2 n'apparaisse pas dans S_1. La *composition* de S_1 et S_2, notée $S_1 \lhd S_2$ est l'algorithme qui comprend toutes les variables et toutes les règles de S_1 et S_2.

Théorème 1. *On suppose que les conditions suivantes sont satisfaites :*
- *l'algorithme S_1 se stabilise vers une première spécification θ.*
- *l'algorithme S_2 se stabilise vers ψ un autre prédicat si θ est vrai.*
- *l'algorithme S_1 ne modifie pas les variables de S_2 une fois que θ est vrai.*
- *les exécutions sont équitables pour S_1 et S_2 (on veut qu'à la fois S_1 et S_2 fassent des actions indéfiniment, sinon, comment S_2 se stabiliserait-il en n'étant jamais exécuté ?).*

Alors $S_1 \lhd S_2$ se stabilise vers ψ.

11 Coloration de sommets d'un graphe planaire

On veut associer à chaque processus p une variable de couleur : $c_p \in \{1,2,3,4,5,6\}$ de telle façon que deux sommets adjacents n'aient pas la même couleur.

Ceci correspond à la post-condition ψ suivante : $\psi = \left(\forall p, q/(p,q) \in E, c_p \neq c_q\right)$, E étant l'ensemble des arêtes.

On se place dans le cadre d'un réseau de machines à numéro identificateur unique, et avec une communication par lecture d'états.

Lemme 5. *Tout graphe planaire admet une orientation acyclique telle que chaque noeud a un degré sortant d'au plus 5.*

Démonstration. Commençons par démontrer que tout graphe planaire admet au moins un sommet de degré 5 ou moins. Le corollaire 4.2.10 de [Die05] indique qu'un graphe planaire avec $n \geq 3$ sommets a au plus $3n-6$ arêtes. Donc en supposant par l'absurde que tout sommet est de degré 6 ou plus, il y a au moins $3n$ arêtes dans le graphe, ce qui n'est pas possible.

On procède alors par récurrence sur n, le nombre de noeuds, pour montrer le lemme. L'initialisation est immédiate. Et pour l'hérédité, soit un graphe G à n noeuds. Il existe un noeud v de degré 5 ou moins. On oriente donc les arêtes de ce noeud vers ses voisins, comme indiqué en figure 4. Son degré sortant est donc égal à son degré, d'au plus 5. On applique alors l'hypothèse de récurrence sur $G \backslash v$, en ajoutant ces nouvelles arêtes orientées, ne créant pas de cycles, pour obtenir une orientation sur G avec les propriétés voulues. □

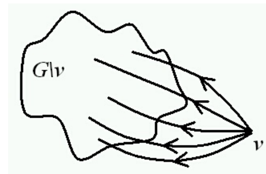

FIG. 4 – Etape de récurrence pour la démonstration du lemme 5.

Ainsi, si l'on connaît une orientation acyclique du graphe planaire, il suffit d'affecter, pour chaque sommet, une couleur différente, et différente de la sienne, à tous ses successeurs, et on obtiendra une coloration du graphe en 6 couleurs. Ceci nous incite donc à composer les deux algorithmes suivants : S_1, la construction d'une orientation acyclique, et S_2, la coloration des sommets à partir d'une orientation acyclique.

Pour construire l'orientation acyclique, on associe à tout processus p un entier x_p tel que toute arête orientée dans le sens $\overrightarrow{pq}$ ssi $(x_p < x_q)$ ou $(x_p = x_q$ et $p < q)$. On note $out(p)$ le degré sortant de p.

Algorithme 9 (S1). $D_p : \{out(p) > 5\} \rightarrow x_p := max\{x_q : q \in voisins(p)\} + 1$

Proposition 8. *Soit la post-condition* $\theta = (\forall p, out(p) \leq 5)$*.* S_1 *se stabilise vers* θ*, et si* θ *est vraie, alors* x_p *reste constante.*

Démonstration. Une configuration est terminale ssi θ est vraie : on va prouver que l'algorithme termine, par décroissance.

Soit une orientation acyclique de référence, avec degré sortant d'au plus 5. Une arête (v_i, v_j) est mauvaise si elle est orientée $\overrightarrow{v_i v_j}$ suivant l'orientation de l'algorithme et $\overrightarrow{v_j v_i}$ suivant l'orientation acyclique de référence $(j > i)$.

Soit $f(\gamma) = (n_0, n_1, \ldots)$ où n_i est le nombre d'arêtes adjacentes à v_i. L'application de D_p fait décroître $f(\gamma)$ pour l'ordre lexicographique. □

Algorithme 10 (S2). $C_p : \big((\exists q / \overrightarrow{pq} \text{ et } c_p = c_q) \text{ et } (\forall r / \overrightarrow{pr}, c_r \neq b)\big) \rightarrow c_p := b$

Proposition 9. *Le programme* S_2 *se termine et si* θ *est vraie, la configuration finale satisfait* ψ*.*

Démonstration. Les arêtes sont orientées de manière acyclique, donc il existe une énumération $v_1, \ldots, v_n$ tel qu'un noeud n'a comme successeur que des noeuds de numéro plus petit.

Soit $g(\gamma) = (m_1, m_2, \ldots)$ où m_i prend la valeur 0 si c_i est différente des couleurs des successeurs, et 1 sinon. L'application de C_p fait décroître $g(\gamma)$ pour l'ordre lexicographique. □

Index

$\lhd$, composition, 7
ψ, post-condition, 2

anneau anonyme, 4
anneau unidirectionnel, 2
anneau uniforme, 4
approche optimiste, 2
approche pessimiste, 2
asynchrone, 1

bien fondé, 3

C, ensemble des configurations, 2
Chang, Ernest, 2
communication par lecture d'états, 1, 3, 5
communication par registres, 1, 4
composition, 7
configuration légitime, 2
convergence, 2
correction, 2
couplage, 5

défaillance transitoire, 2
Dijkstra, Edsger, 3

exclusion mutuelle, 3
exécution, 2

Hsu, Su-Chu, 6
Huang, Shing-Tsaan, 6

Israeli, Amos, 4

Jalfon, Marc, 4

L, ensemble des configurations légitimes, 2
Le Lann, Gérard, 2
lecture d'états, 1, 3, 5
légitime, 2

matching, voir couplage

passage de messages, 4
post-condition, 2
pseudo-stabilisant, 2

registre, 1, 4
Roberts, Rosemary, 2

section critique, 3
spécification, 2
stabilisation, 2
synchrone, 1
système, 2

terminaison, 2
token, 3
token-ring, 3

Références

[Att94] Hagit ATTIYA. « Distributed Algorithms ». http://people.ksp.sk/~misof/skola/Uvod%20do%20distribuovanych%20algoritmov%20(3ita%204ita)/Attiya%20-%20Distributed%20Algorithms.ps.gz, lien vérifié le 21/04/2006, 1994. 1

[CR79] Ernest CHANG and Rosemary ROBERTS. « An improved algorithm for decentralized extrema-finding in circular configurations of processes ». *Communications of the ACM*, 22(5) :281–283, 1979. 2

[Die05] Reinhard DIESTEL. *Graph Theory, third (electronic) edition*, volume 173 of *New York Graduate Texts in Mathematics*. Springer, 2005. http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/GraphTheoryIII.pdf, lien vérifié le 21/04/2006. 7

[Dij65] Edsger Wybe DIJKSTRA. « Solution of a Problem in Concurrent Programming Control ». *Communications of the ACM*, 8(9) :569, 1965. 3

[HH92] Su-Chu HSU and Shing-Tsaan HUANG. « A self-stabilizing algorithm for maximal matching ». *Information Processing Letters*, 43(2) :77–81, 1992. 6

[HJS01] Stephen T. HEDETNIEMI, David P. JACOBS and Pradip K. SRIMANI. « Maximal matching stabilizes in time $O(m)$ ». *Information Processing Letters*, 80(5) :221–223, 2001. http://www.cs.clemson.edu/~stabiliz/Papers/matching-ipl-2001.pdf, lien vérifié le 21/04/2006. 7

[IJ93] Amos ISRAELI and Marc JALFON. « Uniform self-stabilizing ring orientation ». *Information and Computation*, 104(2) :175–196, 1993. 4

[Lan77] Gérard Le LANN. « Distributed systems - Towards a formal approach ». In B. GILCHRIST, editor, *1977 IFIP Congress Proceedings, Information Processing*, volume 77, pages 155–160. North-Holland, Amsterdam, 1977. 2

[LPS92] Nancy A. LYNCH and Boaz PATT-SHAMIR. « Distributed algorithms, lecture notes for 6.852 ». http://people.ksp.sk/~misof/skola/Uvod%20do%20distribuovanych%20algoritmov%20(3ita%204ita)/Lynch%20-%20Distributed%20Algorithms.ps.gz, lien vérifié le 21/04/2006, 1992. 1

[Sch93] Marco SCHNEIDER. « Self-stabilization ». *ACM Computing Surveys*, 25(1) :45–67, 1993. 1

[Tel94] Gerard TELL. « Maximal Matching Stabilizes in Quadratic Time ». *Information Processing Letters*, 49(6) :271–272, 1994. 6