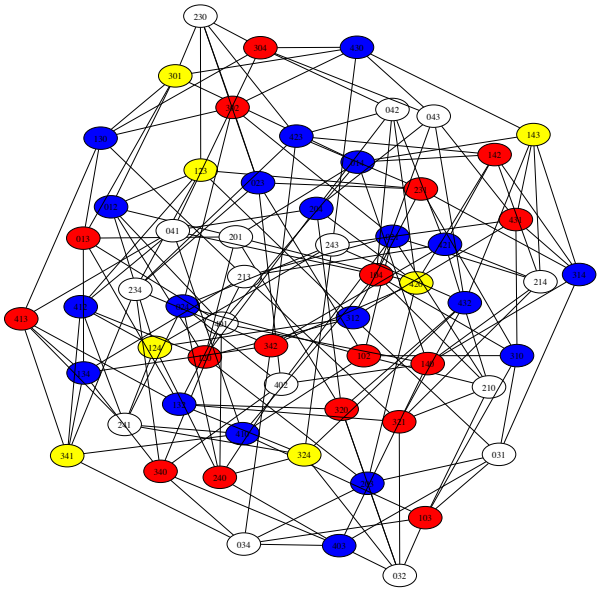
Algorithmes distribués



Cyril Gavoille

LaBRI
Laboratoire Bordelais de Recherche
en Informatique, Université Bordeaux

gavoille@labri.fr

19 novembre 2012

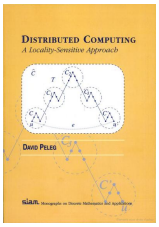
Algorithmes distribués

Master 1, Master 2

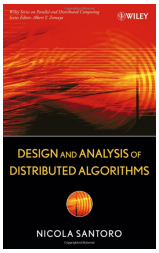
Objectifs : Introduire l'algorithmique distribuée ; concevoir et analyser des algorithmes distribuées (M2) ; présenter différentes applications et problèmatiques actuelles ; programmer des algorithmes sur un simulateur/visualisateur de calcul distribué (M1).

Pré-requis : Algorithmique ; notions de théorie des graphes

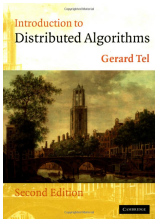
Références :



Distributed Computing : A Locality-Sensitive Approach, David Peleg
SIAM Monographs on Discrete Mathematics and Applications, 2000



Design and analysis of distributed algorithms, Nicola Santoro
Wiley Series on Parallel and distributed computing, 2006



Introduction to Distributed Algorithms (2nd Edition), Gerard Tel
Cambridge University Press, 2000

Table des matières

CHAPITRE 1

Introduction

1.1 Qu'est ce que le calcul distribué ?

Une machine (ou architecture) *séquentielle* permet le traitement d'une seule opération à la fois. Dans un programme séquentiel (comme l'extrait ci-dessous) les instructions s'exécutent les unes à la suite des autres, dans un ordre bien précis.

```
1.   x := x + 1
2.   y := 2*x - 4
3.   print y
4.   ...
```

Les machines *parallèles* permettent le traitement de plusieurs opérations en parallèle. En générale ces machines permettent d'exécuter en parallèle de nombreuses instructions élémentaires toutes similaires (SIMD : Single Instruction Multiple Data). Les ordinateurs classiques, les architectures 64 bits par exemple, sont déjà des sorte de machines parallèles dans la mesure où elles traitent 64 bits simultanément (à l'intérieur d'un même cycle machine). Le degré de parallélisme (qui mesure le nombre d'opérations réalisables en parallèles) est assez limité dans ce cas. Lorsque le nombre de bits manipulés en parallèle devient très important, on parle de machines *vectorielles* (ex : CRAY dans les années 1990). Sur de telles machines, ont peut par exemple calculer $A \cdot X + B$ où A, B sont des vecteurs de grandes tailles et X une matrice. Les machines parallèles possèdent un grand degré de synchronisation.

Un système distribué est une machine parallèle à la différence qu'il y a plusieurs entités de calculs autonomes distantes. La différence principale avec une machine parallèle ...

Multi-cœur ...

En résumé, ce qui caractérise le distribué (par rapport au parallélisme) :

1. potentiellement beaucoup plus d'entités de calcul

2. distance physique entre entités plus importante

Synchrone vs. asynchrone

1.2 Un cadre de base pour les systèmes distribués

1.3 Un cadre de base pour les systèmes distribués

1.4 Spécificités du calcul distribué

1.4.1 Les communications

Elles ne sont pas gratuites.

...

Lorsque le nombre de processeurs dépasse l'ordre du millier, il est difficile de tous les mettre dans une même boîte de 30cm de coté, surtout que dans la pratique, les vitesses de transmission sont assez loin d'être réalisée à la vitesse de la lumière. L'information est transportée à 273 000 km/s dans le cuivre (soit $0.91c$) et les convertions optique-électroniques ne sont pas instantanées.

Si les processeurs sont maintenant équidistant de 3km, alors la fréquence des processeurs doit être alors diminuer d'un facteur 10 000. Dit autrement, à fréquence égale, des processeurs distant de 3km peuvent effectuer 10 000 fois plus d'instructions avant de pouvoir interagir que s'ils sont distant de 30cm.

1.4.2 Connaissance partielle

1.4.3 Erreurs et défaillances

1.4.4 Synchronisme

1.4.5 Non-déterminisme

1.5 Sur les algorithmes distribués

On verra C-distribué.

Send, Receive, NewPulse

...

Tous les algorithmes sont cencés pouvoir s'excuter sur un système asynchrone. Pour écrire un algorithme exclusivement et explicitement synchrone on utilisera l'instruction NewPulse permettant de démarrer un nouveau cycle Send/Receive/calcul. Cette instruction détruit tous les messages reçus à l'étape précédente.

...

Parler de SEND(M, u) : envoie un message contenu dans M vers le voisin u du sommet qui exécute cette commande.

RECEIVE(M, u) est plus délicat : attend de recevoir un message, puis à la réception d'un message met son contenu dans M et dans v le nom du voisin émetteur. Est-ce bien bloquant en synchrone ?

On peut imaginer plusieurs variantes pour RECEIVE.

(M, v) := RECEIVE (= attend de recevoir un message, puis à la réception d'un message met son contenu dans M et met dans v le nom du voisin émetteur).

M := RECEIVE (= comme avant, sauf qu'on oublie d'où est venu le message)

M := RECEIVE(v) (=attend de recevoir un message du voisin v, puis met le contenu dans M).

v := RECEIVE(« toto ») (=attend de recevoir un message de type « toto » et met émetteur dans v). C'est équivalent à : répéter (M, v) := RECEIVE jusqu'à ce que M=« toto »

RECEIVE(« toto », v) (=de recevoir un message « toto » depuis le voisin v)

CHAPITRE 2

Complexités et modèles

À partir de maintenant on suppose un mode de communication point-à-point modélisé par un graphe connexe et simple, *a priori* non orienté. On peut communiquer directement entre deux entités si elle sont connectées par une arête du graphe.

2.1 Mesures de complexité

En séquentiel, on utilise surtout la complexité en temps, plus rarement celle en espace bien qu'en pratique l'espace se révèle bien plus limitant que le temps. Pour un programme qui prend un peu trop de temps, il suffit d'être patient. Un programme qui prend un peu trop d'espace doit être réécrit ou alors il faut changer le *hardware*. C'est donc plus contraignant en pratique.

En distribué, les notions de complexités sont plus subtiles. Les notions de nombre de messages échangés ou de volume de communication font leur apparition. On utilisera essentiellement les complexités en temps et en nombre de messages. Ces complexités dépendant, le plus souvent, du graphe sur lequel s'exécute l'algorithme. Et comme on va le voir, les définitions diffèrent aussi suivant que l'on considère un système synchrone ou asynchrone.

En distribué, les entrées (l'instance) et les sorties d'un algorithme (ou d'un programme) sont stockées sur les sommets du graphe. Aussi, par l'expression « durée d'exécution un algorithme A sur un graphe G » on entend la durée entre le démarrage du premier processeur de G exécutant A et l'arrêt du dernier processeur.

Définition 1 (temps synchrone) *La complexité en* temps *d'un algorithme distribué A sur un graphe G en mode synchrone, noté* $\textsc{Temps}(A, G)$*, est le nombre de tops horloge générés durant l'exécution de A sur G dans le pire des cas (c'est-à-dire sur toutes les entrées valides de A).*

En synchrone, on parle aussi du nombre de *rondes* (*rounds* en Anglais) puisque qu'un algorithme synchrone est une succession de cycles (rondes) Send/Receive/calcul.

Une remarque évidente mais fondamentale est qu'un algorithme de complexité en temps synchrone t implique que pendant son exécution, tout sommet ne peut interagir qu'avec des sommets situés à une distance inférieure ou égale à t.

Définition 2 (temps asynchrone) *La complexité en* temps *d'un algorithme distribué A sur un graphe G en mode asynchrone, noté* $\textsc{Temps}(A, G)$*, est le nombre d'unité de temps durant l'exécution de A sur G dans le pire des cas (c'est-à-dire sur toutes les entrées valides de A et tous les scenarii possibles), chaque message prenant un temps au plus unitaire pour traverser une arête.*

La remarque précédente ne s'applique plus en mode asynchrone, puisqu'il est possible que les messages transitent entre voisin bien plus vite que le temps unitaire, et donc en temps $< t$ entre sommets à distance t.

Un scenario possible pour le mode asynchrone est que tous les messages sont émis en même temps et traversent chaque arête en temps 1 exactement, des messages pouvant se croiser sur une même arête. Il s'agit du scenario synchrone. Autrement dit, la complexité en temps en mode asynchrone est toujours au moins égale à celle synchrone, puisqu'on maximise le temps sur tous les scenarii possibles. En particulier, si durant l'exécution de A deux sommets à distance t interagissent, c'est que la complexité en temps (synchrone ou pas) de A doit être d'au moins t.

Lors d'un scenario synchrone s'exécutant sur un système asynchrone, tous les messages traversent les arêtes à la vitesse du lien le plus lent, ce qui n'est pas le scenario le plus avantageux. L'enjeu d'un algorithme asynchrone est de pouvoir fonctionner dans n'importe quel scenario et donc surtout dans des scenarii ou des messages traversent beaucoup plus vite les arêtes, par exemple dans certaines régions du graphe ou à certains moments de l'exécution de l'algorithme.

Définition 3 (nombre de messages) *La complexité en* nombre de messages *d'un algorithme distribué A sur un graphe G, noté* $\textsc{Message}(A, G)$*, est le nombre de messages échangés durant l'exécution de A sur G dans le pire des cas (sur toutes les entrées valides de A et tous les scenarii possibles en mode asynchrone).*

Résoudre une tâche dans un système distribué prend un certain temps et nécessite un certain volume de communication. Et la plupart du temps, il existe un compromis en ces deux valeurs. Bien sûr c'est très schématique. Il existe d'autres mesures de complexité, comme la complexité en nombre de bits (*bit complexity* en Anglais), donnant le nombre total de bits échangés lors d'une exécution. C'est une mesure plus fine que le nombre de messages puisqu'à nombre de message identique, on pourrait préférer l'algorithme utilisant des messages plus courts.

Comme en séquentiel, on peut aussi définir toute une série de complexités, selon qu'on analyse l'algorithme sur toutes les instances ou pas : analyse dans pire des cas, dans un

cas moyen, selon une distribution particulière, *etc.* En asynchrone on peut également limiter l'ensemble des scenarii possibles, par exemple en limitant la puissance de nuissance de l'adversaire. Le comportement de l'adversaire est par exemple indépendant des choix aléatoires fait par l'algorithme.

2.2 Modèles représentatifs

Parmis les nombreux modèles qui existent dans la litérature, on en distingue trois qui correspondent à des ensembles d'hypothèses différents. Chaque ensemble d'hypothèses permet d'étudier un aspect particulier du calcul distribué. Les trois aspects sont : la *localité*, la *congestion*, l'*asynchronisme*.

Bien sûr, d'autres aspets du calcul distribué pourrait être étudiés comme : la tolérance aux pannes, le changement de topologie, la fiabilité (liens/processeurs qui, sans être en panne, sont défaillant), la sécurité.

Les hypothèses suivantes sont communes aux trois modèles :

1. **Pas d'erreur** : il n'y a pas de crash lien ou processeur, pas de changement de topologie. Les liens et les processeurs sont fiables. Un message envoyé fini toujours par arriver. Les messages peuvent se croiser sur une arête (pas de collision) mais il ne peuvent pas se doubler (les messages arrivent dans le même ordre d'émission).
2. **Calculs locaux gratuits** : en particulier faire un ou plusieurs SEND dans le même cycle de calcul à un coût nul. Finalement, seules les communications sont prises en compte.
3. **Identité unique** : les processeurs on une identité codée par un entier de $O(\log n)$ bits n étant le nombre de processeurs du graphe. Le réseau n'est pas anonyme. Ces identités font partie de l'instance du problème, l'instance étant codée par les sommets du graphe.

La troisième hypothèse (identité unique) ne se révèle pas fondamentale (et donc peut être supprimer) pour certains problèmes que nous allons aborder dans la suite. Elle est en fait au cœur des problèmes consistant à casser la symétrie (*symetry breaking* en Anglais), comme l'élection d'un *leader* (que nous n'aborderons pas). Sans cette hypothèse, certains de ces problèmes ne sont pas toujours solubles. Pour la coloration (abordée au chapitre 7), cette hypothèse est nécessaire. Notons cependant qu'en mode synchrone, un algorithme de complexité en temps t n'a besoin d'identités uniques seulement dans sa boule de rayon t puisqu'au cours de l'exécution aucune interaction entre sommets à distance $> t$ n'est possible.

LOCAL pour étudier la nature locale d'un problème.
- mode synchrone, tous les processeurs démarrent le calcul au même top
- message de taille illimitée

CONGEST pour étudier l'effet du volume des communications.

– mode synchrone, tous les processeurs démarrent le calcul au même top
– message de taille limitée à $O(\log n)$ bits

ASYNC pour étudier l'effet de l'asynchronisme.
– mode asynchrone

Bien sûr, on peut rafiner et multiplier les modèles (à l'infini ou presque) selon de nouvelles hypothèses, avec par exemple le modèle ASYNC et messages limité, ASYNC et messages illimités, *etc.*

Ex : Dans un noeud interne d'un arbre, envoyer l'ID de ses fils vers son père.

2.3 Rappels sur les graphes

Soit $G = (V, E)$ un graphe. On note $V(G) = V$ l'ensemble des sommets et $E(G) = E$ l'ensemble des arêtes de G.

Graphe de bases : chemin, arbres, cycle, graphe complet, grille

Chemin entre deux sommets, connexité

La *longueur* d'un chemin est le nombre d'arêtes qui le compose, et la *distance* entre deux sommets u, v de G, noté $\text{dist}(u, v)$, est la longueur d'un plus court chemin entre u et v.

Diamètre de G est la valeur $\text{diam}(G) = \max_{u,v} \text{dist}(u, v)$.

Eccentricité d'un sommet, centre d'un graphe (=sommet d'eccentricité minimum, exemple, il peut en avoir plusieurs), rayon d'un graphe $\text{rayon}(G)$ (=eccentricité d'un centre).

Arbre : profondeur d'un sommet $\text{depth}(v)$, profondeur d'arbre $\text{depth}(T)$.

Voisinage dans G d'un sommet u à distance r : $\text{boule}(u, r) = \{v \in V(G) : \text{dist}(u, v) \leqslant r\}$.

La somme des degrés des sommets d'un graphe vaut deux fois son nombre d'arêtes : $\sum_{u \in V(G)} \deg(u) = 2m$.

Si G est connexe alors $m \geqslant n - 1$, et donc $m = \Omega(n)$.

2.4 Exercices

2.4.1 Exercice

Montrer que pour tout sommet x d'un graphe G de diamètre D, $D/2 \leqslant \text{ecc}(x) \leqslant D$.

2.4.2 Exercice

Soit G un graphe (connexe) de diamètre D avec $V(G) = \{v_1, \ldots, v_n\}$. On considère le problème IM (*Individual Message*) qui consiste à envoyer un message individuel (distinct) depuis le sommet v_1 vers tous les autres suivant un plus courts chemin spécifié de G.

Dans le modèle CONGEST prouvez ou réfutez :

(a) [Borne Supérieure] : $\forall G$, $\textsc{Temps}(IM, G) = O(D)$

Solution : c'est faux. On prend un graphe formé d'un graphe arbitraire de diamètre $D-1$ et à $n-1$ sommets dans lequel on ajoute une arête et un somme de degré 1, qu'on appel v_1, que l'on connecte à un sommet d'eccentricité maximum. Le nombre de messages devant transiter sur l'arête de v_1 est $n-1$. Donc, en CONGEST, $\textsc{Temps}(IM, G) \geqslant n-1$.

(b) [Borne Inférieure] : $\exists G$, $\textsc{Temps}(IM, G) = \Omega(D)$

(c) [Borne Inférieure Globale] : $\forall G$, $\textsc{Temps}(IM, G) = \Omega(D)$

(d) [Borne Supérieure] : $\forall G$, $\textsc{Temps}(IM, G) = O(n)$

Solution (d) : après D étapes, au moins un nouveau sommet va être informé à chaque nouvelle étape [A FINIR]

Même question dans le modèle LOCAL et ASYNC.

CHAPITRE 3

Arbre et diffusion

Dans ce chapitre nous donnons des algorithmes élémentaires pour la diffusion, la concentration, et la construction d'arbre de diffusion.

3.1 Diffusion

Il s'agit de transmettre un message M à tous les sommets du graphe depuis un sommet distingué que l'on notera r_0.

On peut vérifier que :

Proposition 1 *Tout algorithme distribué de diffusion A sur un graphe G à n sommets vérifie (en mode synchrone ou asynchrone) :*
- $\textsc{Message}(A, G) \geqslant n - 1$.
- $\textsc{Temps}(A, G) \geqslant \text{ecc}(r_0)$.

On rappelle que $\text{ecc}(r_0)$, l'*eccentricité* de r_0, est la profondeur minimum d'un arbre couvrant G et de racine r_0. C'est donc la profondeur d'un arbre couvrant en largeur d'abord qui a pour racine r_0. Notons en passant que le *diamètre* de G, noté $\text{diam}(G)$, est la valeur maximum de l'eccentricité.

Preuve de la proposition 1. Il faut informer $n-1$ sommets distincts du graphe. L'envoi d'un message sur une arête ne peut informer qu'un seul sommet (celui situé à l'extrémité de l'arête). Donc il faut que le message circule sur au moins $n-1$ arêtes différentes de G pour résoudre le problème. Donc $\textsc{Message}(A, G) \geqslant n - 1$.

Dans G il existe un sommet u à distance au $t = \text{ecc}(r_0)$ de r_0. Dans le scenario synchrone, chaque message traversant une arête prend un temps 1. Il faut donc un temps au moins t pour que u soit informé. Donc $\textsc{Temps}(A, G) \geqslant t$. □

3.2 Arbre de diffusion

Une stratégie courante pour réaliser une diffusion est d'utiliser un arbre couvrant enraciné en r_0, disons T. L'intérêt est qu'on ne diffuse M que sur les arêtes de T. Donc, ici on va supposer que chaque sommet u de G connaît T par la donnée de son père $\text{PÈRE}(u)$ et de l'ensemble de ses fils $\text{FILS}(u)$. L'algorithme ci-dessous est spécifique pour chaque arbre T.

Algorithme $\text{CAST}_T(r_0)$
(code du sommet u)

1. Si $u \neq r_0$, attendre de recevoir un message (M, v)
2. Pour tout $v \in \text{FILS}(u)$, $\text{SEND}(M, v)$

Si l'algorithme précédent s'exécute dans un environnement synchrone, il faut comprendre l'attende de reception comme bloquante pour chaque top horloge.

Proposition 2 *En mode synchrone ou asynchrone :*
- $\text{MESSAGE}(\text{CAST}_T(r_0), G) = |E(T)| = n - 1$.
- $\text{TEMPS}(\text{CAST}_T(r_0), G) = \text{hauteur}(T)$.

Notons que la hauteur de T vaut $\text{ecc}(r_0)$ si T est un arbre en largeur d'abord. Dans ce cas, l'algorithme CAST_T est optimal en temps et en nombre de message.

3.3 Innondation

On utilise cette technique (*flooding* en Anglais) en l'abscence de structure pré-calculée comme un arbre de diffusion. Un sommet u ne connaît pas $\text{PÈRE}(u)$ et $\text{FILS}(u)$.

Algorithme $\text{FLOOD}(r_0)$
(code du sommet u)

1. Si $u = r_0$, alors $\text{SEND}(M, v)$ pour tout voisin v de u.
2. Sinon,
   (a) Attendre de recevoir (M, v)
   (b) $\text{SEND}(M, w)$ pour tout voisin $w \neq v$ de u.

Proposition 3 *En mode synchrone ou asynchrone :*
- $\text{MESSAGE}(\text{FLOOD}(r_0), G) = 2|E(G)| - (n - 1)$.
- $\text{TEMPS}(\text{FLOOD}(r_0), G) = \text{ecc}(r_0)$.

Preuve. ... □

3.4 Arbre couvrant

Considérons maintenant le problème de construire de manière distribué un arbre couvrant de racine r_0. Des variantes de ce problème seront examinées plus amplement au chapitre 5. Plus précisément on souhaite qu'à la fin de l'exécution de l'algorithme chaque sommet u de G fixe une variable PÈRE(u) correspondant à son père, et un ensemble FILS(u) correspondant à l'ensemble de ses fils.

Proposition 4 *À partir de tout algorithme de diffusion à partir d'un sommet r_0 on peut construire un algorithme de construction d'un arbre couvrant en racine r_0 de complexité, en temps et en nombre de messages, équivalente.*

Preuve. Si l'on connaît un algorithme A de diffusion à partir de r_0, il suffit, pendant l'exécution de A de définir le père de u comme le voisin v d'où u a reçu M en premier. La flèche du temps interdit la création d'un cycle, et la diffusion à tous les sommets garantit que la forêt ainsi calculée est bien couvrante et connexe.

Si l'on souhaite récupérer les fils de u, il faut émettre vers son père un message particulier et collecter ces messages. Cela ajoute $n-1$ messages suplémentaires, et une unité à la complexité en temps. Cela ne modifie pas les complexités en temps et en nombre de messages car $n-1$ est une borne inférieure pour la diffusion d'après la proposition 1.

Notons qu'en asynchrone, le sommet u de sait pas déterminer à partir de quel moment sa liste de fils est complète ou non. La liste sera complète après un temps $t+1$, où t est la complexité en temps de la diffusion. Mais n'ayant pas d'horloge, u est incapable de déterminer si le temps $t+1$ est révolu ou non. C'est le problème de la détection de la terminaison. □

On laisse en exercice l'écriture d'un algorithme permettant de construire un arbre couvrant de racine r_0.

En combinant les propositions 3 et 4, on obtient directement :

Corollaire 1 *Il existe un algorithme distribué noté* SPANTREE *(basé sur* FLOOD*) permettant de construire pour tout graphe G et sommet r_0 un arbre couvrant G de racine r_0 avec (en synchrone ou asynchrone) :*
- $\text{MESSAGE}(\text{SPANTREE}, G) = O(|E(G)|)$.
- $\text{TEMPS}(\text{SPANTREE}, G) = O(\text{ecc}(r_0))$.

Il est a noté que les arbres ainsi construits diffèrent beaucoup s'ils s'exécutent en mode synchrone ou asynchrone. Par, exemple, dans le cas où G est une clique (graphe complet), l'arbre généré par SPANTREE en synchrone est toujours une étoile, alors qu'il peut s'agir de n'importe quel arbre couvrant en mode asynchrone.

3.5 Diffusion avec détection de la terminaison

Un problème potentiel des algorithmes précédant est qu'un sommet ne sait pas si la diffusion est terminée. C'est en fait un problème général en algorithmique distribuée. Cela peut être très gênant en asynchrone lorsque plusieurs algorithmes doivent être enchaînés. Le risque est que des messages des deux algorithmes se mélangent et aboutissent à des erreurs.

On considère donc la variante de la diffusion où en plus de réaliser une diffusion à proprement parlée, on souhaite que chaque sommet u possède une variable booléenne $\textsc{Fin}(u)$, qui lorsqu'elle passe à Vrai indique à u que tous les sommets de G ont bien reçus le message M.

La solution consiste à modifier l'algorithme Flood en envoyant un message (« done ») à son père lorsque qu'on est sûr que tous les sommets de son propre sous-arbre ont été informés. Puis, le sommet r_0 diffuse un message de fin vers tous les sommets de G. Pour cela on se sert de l'arbre que l'on construit en même temps que Flood (message « père »).

Algorithme $\textsc{Flood\&Echo}(r_0)$
(code du sommet u)

1. $\textsc{Fils}(u) := \varnothing$ et $\textsc{Fin}(u) :=$ Faux.
2. Si $u = r_0$, alors $\textsc{Send}(\text{« data »} + M, v)$ pour tout voisin v de u, poser $c := \deg(u)$.
3. Sinon,
   (a) Attendre de recevoir un message (M, v).
   (b) $\textsc{Send}(\text{« père »}, v)$, $\textsc{Send}(\text{« data »} + M, w)$ pour tout voisin $w \neq v$ de u.
   (c) Poser $\textsc{Père}(u) := v$ et $c := \deg(u) - 1$.
4. Tant que $c > 0$:
   (a) Attendre de recevoir un message (M', v).
   (b) Si M' de type « data », $\textsc{Send}(\text{« ack »}, v)$.
   (c) Si M' de type « père », $\textsc{Fils}(u) := \textsc{Fils}(u) \cup \{v\}$.
   (d) Si M' de type « ack » ou « done », poser $c := c - 1$.
5. Si $u \neq r_0$, $\textsc{Send}(\text{« done »}, \textsc{Père}(u))$ et attendre un message « fin ».
6. $\textsc{Fin}(u) :=$ Vrai, $\textsc{Send}(\text{« fin »}, v)$ pour tout $v \in \textsc{Fils}(u)$.

Dans la suite on notera $D = \text{diam}(G)$ le diamètre de G, et $m = |E(G)|$ le nombre d'arêtes de G.

Proposition 5 *En mode synchrone :*
- $\textsc{Message}(\textsc{Flood\&Echo}(r_0), G) = O(m)$.
- $\textsc{Temps}(\textsc{Flood\&Echo}(r_0), G) = O(\text{ecc}(r_0)) = O(D)$.

En mode asynchrone :
- $\textsc{Message}(\textsc{Flood\&Echo}(r_0), G) = O(m)$.
- $\textsc{Temps}(\textsc{Flood\&Echo}(r_0), G) = O(n)$.

Preuve. [IL FAUT MONTRER QUE L'ALGO EST VALIDE]

En ce qui concerne le nombre de messages, les étapes 2,3 envoient $\sum_u \deg(u) = 2m$ messages, qui sont reçus à l'étape 4. Les étapes 5,6 concentrent et diffusent le long d'un arbre, donc consoment $2(n-1)$ messages. En tout, cela fait $O(m+n) = O(m)$ messages puisque G est connexe (et donc $m \geqslant n-1$).

En ce qui concerne le temps, les étapes 2,3,4 prennent un temps $O(D)$. La concentration puis la diffusion aux étapes 5,6 prennent un temps $O(D)$ en synchrone. Malheureusement, en mode asynchrone, on ne maîtrise pas la hauteur de l'arbre, qui peut être de $n-1$. La complexité en temps asynchrone est donc $O(D+n) = O(n)$ (car $D < n$). □

3.6 Concentration

La concentration est un problème similaire à celui de la diffusion. Il s'agit pour chaque sommet u du graphe d'envoyer un message personnel M_u vers un sommet distingués r_0. C'est en quelques sortes le problème inverse, sauf que les messages ne sont pas tous identiques comme dans le cas d'une diffusion.

Voici une solution, lorsqu'un arbre T de racine r_0 est disponible, c'est-à-dire connu de chaque sommet u par les variables $\textsc{Père}(u)$ et $\textsc{Fils}(u)$.

Algorithme $\textsc{Converge}_T(r_0)$
(code du sommet u)

1. Poser $c := |\textsc{Fils}(u)|$.
2. Si $c = 0$ et $u \neq r_0$, $\textsc{Send}(M_u, \textsc{Père}(u))$.
3. Tant que $c > 0$:
   (a) Attendre de recevoir (M, v)
   (b) $c := c - 1$.
   (c) Si $u \neq r_0$, $\textsc{Send}(M, \textsc{Père}(u))$.

Si aucun arbre n'est disponible, le plus simple est alors d'en calculer un, par exemple grâce à SpanTree, puis d'appliquer $\textsc{Converge}_T$. La mauvaise solution étant que chacun des sommets réalise indépendamment une diffusion : on est donc sûr que r_0 recoive une copie de chaque message. Cela n'est pas efficace pour le nombre de messages.

Proposition 6 *Il existe un algorithme distribué* CONVERGE *qui réalise pour tout graphe G et sommet r_0 la concentration vers r_0 avec :*

En mode synchrone :
- $\text{MESSAGE}(\text{CONVERGE}, G) = O(m)$,
- $\text{TEMPS}(\text{CONVERGE}, G) = O(D)$.

En mode asynchrone :
- $\text{MESSAGE}(\text{CONVERGE}, G) = O(m)$,
- $\text{TEMPS}(\text{CONVERGE}, G) = O(n)$.

CHAPITRE 4

Le langage C-distribué

Dans ce chapitre nous introduisons un langage de programmation pour les algorithmes distribués. Ce langage est très proche du Java utilisé dans le logiciel ViSiDiA. La traduction d'un programme écrit en C-distribué en ViSiDiA est quasi automatique. La difficulté reste bien sûr dans la traduction de l'algorithme donné en langage naturel vers un langage de plus bas niveau comme le C-distribué qui gère individuellement et précisément chaque messages.

4.1 Détection de la terminaison

Le langage C-distribué se présente comme du langage C (ou C++) étendu de primitives de communication (`SEND`, `RECEIVE`, *etc.*). Par habitude, on notera les extensions au langage C en majuscule. Voici quelques principes qu'il faut bien avoir en tête.

- Le même programme est chargé sur tous les processeurs. Au départ, tous les processeurs sont dans un état de veille (endormis). On peut cependant en réveiller un (une source par exemple) ou plusieurs sommets. À la réception d'un message, un processeur se réveille, mais il peut se réveillé avant.
- Toutes les variables sont locales au processeur qui exécute le programme. Une déclaration du type `int i ;` à pour effet de réserver un espace mémoire sur le processeur qui exécute cette instruction. Quand on déclare une variable, la déclaration est faite sur tous les processeurs (tout le monde possède la variable `i`) mais elle est physiquement locale à chaque processeur.
- Sur un lien donné les messages sont reçus dans l'ordre de leur émission (*First-In First-Out*), ils ne peuvent se croiser. Les messages arrivant sur un processeurs sont automatiquement stockés dans une file d'attente. En fait, il y a autant de files d'attentes que d'arêtes incidentes. Les fonctions de type `RECEIVE` ont pour effet de vider ces files. Les réceptions sont bloquantes, mais pas les envois.
  Cependant en mode synchrone (`SYNC`), après que le cycle t de l'horloge se soit écoulé, tous les messages reçus pendant le cycle t sont détruits de la file et les instructions de réceptions relâchées. Donc on ne peut recevoir des messages que pendant le cycle correspondant à leur émission. En quelques sortes les messages sont estampillés par le top horloge de leur émission. Une instruction de réception qui s'excécute au top t

ne peut recevoir qu'un message estampillé du top t.

Il y a deux types spéciaux de variables [1] :

`arête` / `message` une arête indique le cannal (ou lien) sur lequel un message est reçu ou envoyé. Le type `arête` est compatible avec le type `int`. Sa valeur est toujours $\geqslant 0$. On peut donc s'en servir comme index d'un tableau, la valeur -1, par exemple, peut être utilisée comme sentinelle. Une variable de type `message` est une structure avec deux champs (`.from` et `.data`) détailés plus loin.

Le type `arête` correspond au type `Door` de ViSiDiA, le type `message` plus ou moins au type `Message` de ViSiDiA.

Exemple de déclaration :

```
arête a;
message M;
```

`SYNC` / `ASYNC` permet de choisir le mode de programmation synchrone ou asynchrone. Cette directive doit figurer en début de programme. Notez bien que dans le mode `SYNC` tous les processeurs se réveillent en même temps. Par défaut, les programmes sont toujours écrits en mode `ASYNC`. Donc en pratique on met `SYNC` ou rien.

`ID` constante de type `int` donnant l'identité du sommet courant, toujours de valeur $\geqslant 0$.

`SEND(a,M)` envoie le message `M` sur l'arête `a` incidente au sommet courant.

`SENDALL(M)` envoie le message `M` à tous les voisins du sommet courant.

`M.from` de type `arête` donne l'arête incidente au sommet courant sur laquelle le message `M` a été reçu. Il n'est pas conseillé d'écrire dans ce champs, et de laisser cette tâche aux fonctions de réception de messages.

`M.data` donne le contenu du mesage `M` (dont le type est à définir, dépend du contexte, par exemple une chaîne de caractères).

`M=RECEIVE()` attend de recevoir un message, ou bien lit et enlève le premier message de la file d'attente de réception (si le message est déjà arrivé), et le stocke dans `M`. Les champs `.from` et `.data` sont mis à jour. Dans le mode `SYNC` l'attente ne dure qu'un cycle horloge.

`M=RECEIVEFROM(a)` attend de recevoir un message sur l'arête `a` incidente au sommet courant, ou bien lit et enlève le premier message de la file d'attente associée à l'arête `a` (si un tel message est déjà arrivé), et le stocke dans `M`. Les autres messages reçus et émis sur d'autres arêtes incidentes au sommet courant ne sont pas supprimés des files de réception. Dans le mode `SYNC` l'attente ne dure qu'un cycle horloge.

`FORALL(a) { ... }` répète une instruction ou un bloc d'instruction pour toutes les arêtes incidentes `a` du sommet courant. C'est une façon de traduire la phrase :
« pour tous ses voisins faire ... »

1. Types qui, contrairement aux habitudes énoncés précédemment, ne sont pas écrits en majuscules.

`PULSE` dans le mode `SYNC`, permet de passer au cycle suivant. Les files de réceptions de messages émis au cycle courant sont supprimés.

Quelques exemples :

- Envoi d'un message à tous ses voisins :
  `FORALL(a) SEND(a,M) ;` ce qui est équivalent de `SENDALL(M) ;`
- Calcule dans `d` le degré du sommet courant :
  `d=0 ; FORALL(a) d++ ;`
- Attend de recevoir d voisins messages (si d est le degré du sommet courant) :
  `FORALL(a) RECEIVE() ;`
- Attend de recevoir un message de chaque voisin :
  `FORALL(a) RECEIVEFROM(a) ;`
- Attend un message en synchrone pendant une durée indéterminée :
```
M.from=-1;
while(M.from<0)
{ M=RECEIVE();
  PULSE; }
```
- Diffusion d'un message `M` à partir d'une source `r0` :
```
main(r0,M){
if (ID==r0) SENDALL(M);
else { M=RECEIVE();
       FORALL(a) if (a<>M.from) SEND(a,M); }
}
```
- La même chose en synchrone :
```
SYNC;
main(r0,M){
if (ID==r0) { SENDALL(M); PULSE; }
else { M.from=-1;while(M.from<0) { M=RECEIVE(); PULSE; }
       FORALL(a) if (a<>M.from) SEND(a,M);
       PULSE; }
}
```

4.2 Transformation de C-distribué en ViSiDiA

On ne peut pas implémenter directement une instruction C-distribué `M.from=-1` en ViSiDiA. Il faut utiliser une variable supplémentaire, `Mfrom=-1`, et remplacer toute occurence en lecture à `M.from` par `Mfrom`. Ensuite, il faut ajouter à toute réception `M=receive(door)` ou `M=receiveFrom(i)`, les lignes `Mfrom=door.getNum()` ou `Mfrom=i` de mise à jour de la variable `Mfrom`.

```
M=receive();
j=M.from;
```

se traduit en

```
Door p=new Door();
M=receive(p);
j=p.getNum();
```

Pour endormir le processeur local pendant x miliseconde : `Thread.sleep(x)`

Pour générer un entier aléatoire :

```
Random generator=new Random();
int x=generator.nextInt();
```

4.3 Ce que ne fait pas ViSiDiA et qu'il devrait faire

Door. Lorsqu'on déclare une porte avec `Door d=new Door()`, à quelle valeur est initialisé le champs `d.getNum()` ? On peut initialiser le champs `d.data`, mais pas le `.getNum()`. Or parfois il est utile de construire un message de toute pièce pour simuler la réception ou l'émission d'un message à soit même.

stateArc. Dans le programme on devrait pouvoir afficher une orientation sur les arêtes. Typiquement pour la sélection du père comme premier message arrivant, on aimerait avoir une instruction `stateArc(i,type)` qui dessine un petit arc, on met une valeur prêt du port `i`. Cela devrait aussi permettre de faire figurer le numéro des port.

Numérotation. La numérotation des sommets (identité) devrait pouvoir être fixé dans l'éditeur de graphe sans que la rénumérotation s'oppère lors de l'appel du simulateur. En effet, pour tester les algorithmes de coloration en $\log^* n$ dans un cycle, il est important que les numéros de soient pas consécutifs. Il faudrait si possible, dans l'éditeur ET dans le simulateur, prévoir un bouton qui effectue une permuation aléatoire. Il devrait en être de même avec les numéros de ports, certains algorithmes de routage étant assez sensible à la numérotation.

Attribu, éditeur de graphes décorés. L'idée serait que l'éditeur permette d'entrer une connaissance a priori : par exemple, le numéro de port 0 est le père (s'il s'agit d'un arbre), où alors la racine est le numéro d'identité 0. Mais plus généralement, il faudrait pouvoir affecter des attribus aux sommets et ports (voir arêtes) pour gérer les "oracles", les "étiquettes" ou encore les poids pour le calcul de MST. Actuellement, la couleur d'un sommet ne peut être fixé que dans le simulateur, pas dans l'éditeur. Or un arbre enraciné, par exemple, est un objet qui est un arbre munit d'une racine. L'éditeur devrait permettre d'éditer des graphes décorés.

Le sytème de fichier. Il faudrait qu'on puisse compiler son .java dans le répertoire de son choix, et que le similateur puisse charger correctement le .class correspondant

Version enseignement. Il faudrait faire deux versions. L'une pour les développeur comportant tous les algos déjà programmés, une autre avec seulement quelques exemples significatifs (illustration des colorations d'arête, type de message, mode synchrone, etc.)

Dessin de graphe. L'éditeur devrait permettre le dessin de graphes spécifique, arbre, planaire, spring.

send asynchrone. Il faudrait pouvoir modifier le délais de transmission des messages. Actuellement, la vitesse est constante le long des arêtes et donc le délais de transmission est proportionnelle à la longueur des arêtes. Ceci implique donc un scénario très particulier de l'exécution, qui est forcément selon un graphe Euclidien. Beaucoup de scénarii échappant à cette constrainte, il faudrait avoir un mode où les `send` se font avec un délais d'attente par exemple, ou encore un autre mode où la vitesse varie dans le temps, est différente pour chaque arc (et pas seulement les arêtes, il peut avoir l'asymétrie).

Graphe dynamique. L'idée serait ici de pouvoir tester des algorithmes distribués pour les graphes dynamiques. Par exemple, on ajouter des feuilles à un arbre initial et on maintient une approximation du nombre de descendant. Il faudrait donc un moyen de décrire un scénario, par un langage ?

4.4 Exercices

4.4.1 Exercice

Montrer que le problème de construire un arbre couvrant enraciné en r_0 est équivalent au problème de la diffusion depuis r_0.

4.4.2 Exercice

Prouver que pour tout algorithm de diffusion sur un graphe G à n sommets et comme source r_0, $\textsc{Message}(B, G) \geqslant n-1$ et $\textsc{Temps}(B, G) \geqslant \text{ecc}(r_0)$ en synchrone ou asynchrone.

4.4.3 Exercice

Donner un algorithme distribué B^* qui dans un graphe G diffuse un message M depuis une source r_0 avec détection de la terminaison. Prouver sa validité, puis analyser $\textsc{Message}(B^*, G)$ et $\textsc{Temps}(B^*, G)$ en fonction de $\text{ecc}(r_0)$, du nombre de sommets n et d'arêtes m, en synchrone et asynchrone. Les performances sont elles les mêmes dans le modèle CONGEST ? En déduire un algorithme distribué, si un sommet est distingué, permet de compter le nombre de sommets de G avec détection de la terminaison (donc en particulier chaque sommet doit avoir connaissance du n).

4.4.4 Exercice : diffusion avec écho

Ecrire en C-distribué un programme implémenant la diffusion d'un message `M` depuis `r0` avec détection de la terminaison (codée dans une variable `terminée` qui passe à `TRUE` seulement lorsque la tâche globale est terminée).

4.4.5 Exercice

On peut adapter facilement le programme pour compter le nombre total de sommets et que chaque sommet le sache.

CHAPITRE 5

Arbres couvrants

Dans ce chapitre on s'intéresse au calcul d'arbres couvrants (BFS, DFS, de poids minimum) en mode asynchrone. On suppose donc dans la suite qu'on est en mode asynchrone (model ASYNC).

5.1 Arbres en largeur d'abord (BFS)
[*Breath First Search*]

Un arbre BFS de racine r_0 est un arbre T tel que $d_T(r_0, u) = d_G(r_0, u)$ pour tout sommet u de T. L'arbre T est couvrant s'il passe par tous les sommets de G, c'est-à-dire si $V(T) = V(G)$.

Nous avons vu au chapitre précédant comment construire un arbre couvrant à partir de FLOOD, voir le corollaire 1. En mode synchrone cet arbre est précisément un BFS. L'enjeux ici est de le faire en mode asynchrone.

5.1.1 Dijkstra

Ici ...

Il ne s'agit pas vraiment de l'algorithme de Dijkstra car on suppose que les arêtes de G on un poids unitaire.

Algorithme DIJDIST(r_0)

1. ...

	Message	Temps
Borne inférieure	$\Omega(m)$	$\Omega(D)$
DijDist	$O(m + nD)$	$O(D^2)$
Bellman-Ford	$O(mn)$	$O(D)$
[BDLP08]	$O(mD)$	$O(D)$
Meilleur connu [AP90]	$O(m \log^3 n)$	$O(D \log^3 n)$

Table 5.1 – Compromis Temps/Message pour les algorithmes distribués en mode asynchrone pour le calcul d'un arbre couvrant BFS.

5.1.2 Bellman-Ford

5.1.3 Résumé

5.2 Arbres en profondeur d'abord (DFS) [*Depth First Search*]

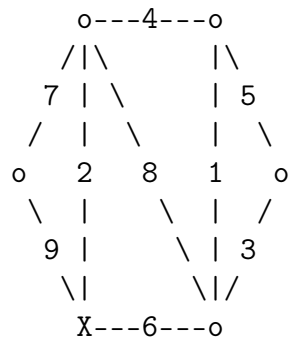
Il s'agit maintenant de visiter tous les sommets d'un graphe en suivant les arêtes. En séquentiel on utilise un parcours en profondeur d'abord (DFS). Le principe est de démarrer depuis un sommet r_0 et pour chaque sommet v faire : si v a des voisins non encore visités, alors on en visite un. Sinon, on revient vers le sommet qui a visité v en premier. S'il aucun de ces sommets n'existe, la recherche est terminée. En séquentiel, cela prend un temps $\Theta(|E|)$, tous les voisins de tous les sommets pouvant être testés.

...

5.3 Arbres de poids minimum (MST) [*Minimum Spanning Tree*]

En séquentiel, il y a principalement deux algorithmes : PRIM et KRUSKAL. Pour PRIM, on maintient un arbre T de racine r_0 qui grossit en ajoutant à chaque fois l'arête sortante de poids minimum, c'est-à-dire l'arête incidente dont une seule extrémité est dans T et qui est de poids le plus petit possible. L'arbre est initialisé au sommet seul r_0. Pour KRUSKAL, on trie les arêtes, puis on ajoute à la forêt l'arête la plus petite ne créant pas de cycle. Au départ T est vide.

Prendre un exemple et faire tourner avec PRIM puis KRUSKAL.



Remarque de culture générale : Le problème de MST peut être généralisé de la manière suivante, appelé « arbre de Steiner » : on fixe un ensemble de sommets, disons $U \subseteq V(G)$, et on demande de construire un arbre couvrant U et de poids minimum. Donc MST = Steiner avec $U = V(G)$. Motivation : un groupe d'utilisateurs doit recevoir le même film vidéo et il faut optimiser le coût de diffusion.

Calculer un arbre de Steiner est un problème est NP-complet [Karp 1972] (c'est-à-dire on ne connaît pas encore d'algorithme polynomial en n et on pense qu'il n'y en a pas), et il le reste même si tous les poids sont égaux.

5.3.1 Algorithme PRIM

5.3.2 Algorithme GHS synchrone

C'est une variante de l'algorithme de Kruskal

5.4 Exercices

5.4.1 Exercice

Écrire en C-distribué un programme de paramètre r_0 implémentant le DFS décrit dans le cours (version optimisant le temps), c'est-à-dire le jeton doit passer exactement deux fois par chaque arête d'un arbre couvrant, et passer par tous les sommets.

Bibliographie

[AP90] Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In 31^{st} *Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 514–522. IEEE Computer Society Press, October 1990.

[BDLP08] Christian Boulinier, Ajoy K. Datta, Lawrence L. Larmore, and Franck Petit. Space efficient and time optimal distributed BFS tree construction. *Information Processing Letters*, 108(5) :273–278, November 2008.

CHAPITRE 6

Synchroniseurs

La conception d'un algorithme distribué pour un système asynchrone est bien plus difficile que pour un système synchrone. L'analyse de sa validité doit prendre en compte tous les scenarii possibles, alors qu'un synchrone il n'y a, le plus souvent, qu'un seul scenario possibles. Pour s'en convaincre, il suffit de considérer le problème de la construire d'un arbre BFS (vu au chapitre 5 précédent).

L'idée, introduite en 1982 par Gallager [Gal82] puis généralisée par Awerbuch en 1985 [Awe85], est de transformer automatiquement un algorithme synchrone pour qu'il puisse fonctionner sur un système asynchrone. Le synchroniseur simule donc des envoies de messages synchrones, au top d'horloge t, sur un système asynchrone, sans horloge. Ceci se fait en payant des surcoûts en nombre de messages dits de synchronisation et sur le temps. Il faut également prendre en compte de temps de construction éventuelle du synchroniseur. Plusieurs synchroniseurs sont possibles, chaque fois avec des compromis différents sur ces surcouts.

6.1 Méthodologie

Soit S un algorithme synchrone écrit pour un système synchrone et σ un synchroniseur. On souhaite construire un nouvel algorithme $A = \sigma(S)$, obtenu en combinant σ et S, qui puisse être exécuté sur un système asynchrone. Pour que la simulation soit correcte, il faut que l'exécution de A sur un système asynchrone soit « similaire » à l'exécution de S sur un système synchrone (on va donnera plus tard un sense plus précis à « exécutions similaires »).

Dans chacun des sommets, l'algorithme A a deux composantes :

1. La composante originale représentant l'algorithme S que l'on veut simuler, c'est-à-dire toutes les variables locales, les routines et les types de message utilisés par S.
2. La composante de synchronisation, c'est-à-dire le synchroniseur σ, qui peut comprendre ces propres variables locales, routines et types de message utilisés.

Ces deux composantes doivent être bien distinctes puisque pendant l'exécution de A, par exemple, des messages des deux composantes vont être aménér à coexister. Les mes-

sages « ack » originaux de S ne doivent pas être confondus avec ceux ajoutés par le synchroniseurs.

L'idée de base est que chaque processeur u possède un générateur de *pulse* (=top horloge), $\textsc{Pulse}(u) = 0, 1, 2, \ldots$ variable qui intuitivement simule les tops horloges d'une horloge globale. Dans l'algorithme A on va exécuter la phase p de l'algorithme S (grâce à la composante originale) uniquement lorsque $\textsc{Pulse}(u) = p$.

Malheureusement, après avoir exécuté la phase p de S, u ne peut pas se contenter d'incrémenter $\textsc{Pulse}(u)$ et de recommencer avec la phase $p+1$. Il faut attendre ses voisins puisque sinon un voisin v pourrait recevoir des messages de u de la phase $p+1$ alors que v est toujours dans la phase p. La situation peut être encore moins favorable, avec un décalage pire encore. Il pourrait par exemple se produire qu'aux 5 phases suivantes, $p+1, \ldots, p+6$ de S, il ne soit pas prévu que u envoye et recoive des messages. Si bien que u pourrait effectuer tous ses calculs locaux de la phase $p+1$ à la phase $p+6$ et passer directement $\textsc{Pulse}(u) = p+7$. Si à la phase phase $p+7$ il envoie un message à v, v qui est dans la phase p aurait légitimement l'impression de recevoir des messages du futur.

Pour résoudre le problème, on pourrait penser à première vue qu'il suffit de marquer chaque message par le numéro de phase de l'émetteur, et ainsi retarder les messages venant du futur. La question qui se pose est alors de savoir combien de temps il faut les retarder. Malheureusement, un processeur v ne peut pas savoir *a priori* combien de messages de la phase p il est censé recevoir. Sans cette information v pourrait attendre de recevoir un message de la phase p, alors dans S à la phase p il est peut être prévue qu'aucun voisin ne communique avec v.

En asynchrone, lorsqu'un sommet attend de recevoir un message il attend potentiellement pour toujours. La réception étant bloquante, il n'y a pas de *time out* car pas d'horloge. En synchrone, les réceptions se débloquent à la fin de chaque phase.

Comme on va le voir dans la proposition 7, pour que la simulation fonctionne, il suffit de garantir ce qu'on appelle la « compatibilité de pulse ».

Définition 4 (Compatibilité de pulse) *Si un processeur u envoie à un voisin v un message M de la composante originale de la phase p lorsque $\textsc{Pulse}(u) = p$, alors M est reçu en v lorsque $\textsc{Pulse}(v) = p$.*

Bien sûr, les exécutions de A et de S ne peuvent pas être identiques, car d'une part sur un système asynchrone, les exécutions ne sont pas déterministes (deux exécutions sur les mêmes entrées ne donne pas forcément le même résultat !), et que d'autre part les messages de σ dans l'exécution de A n'existent évidemment pas dans l'exécution de S.

On donne maintenant un sens plus précis à la notion de similarité entre les exécutions de S et de A.

Définition 5 (Exécutions similaires) *Les exécutions de S sur un système synchrone et de $A = \sigma(S)$ sur un système asynchrone sont* similaires *si pour tout graphe G (l'instance) sur lesquels s'exécutent S et A, toute variable X, toute phase p et toute arête uv de G :*

- *Les messages de la composante originale envoyés (ou reçus) par u sur l'arête uv lorsque* $\textsc{Pulse}(u) = p$ *sont les mêmes que ceux de S à la phase p envoyés (ou reçus) par u sur l'arête uv.*
- *La valeur de X en u au début de l'exécution de A lorsque* $\textsc{Pulse}(u) = p$ *est la même que celle de X en u dans l'exécution du début de la phase p de S. La valeur de X en u lorsque A est terminé est la même que celle de X en u lorsque S est terminé.*

Pour comprendre la définition précédente supposons que $S = \textsc{SpanTree}$, qui calcule un arbre couvrant BFS (en largeur d'abord) sur un système synchrone. Si $A = \sigma(S)$ et S ont des exécutions similaires, alors en particulier les variables $\textsc{Père}(u)$ et $\textsc{Fils}(u)$ seront identiques à la fin des exécutions de A et de S. Elles définiront alors les mêmes arbres. Il en résulte que l'algorithme A calcule un arbre couvrant BFS sur un système asynchrone.

Proposition 7 *Si le synchronisateur σ garanti la compatibilité de pulse pour tout algorithme synchrone S, toute phase, tout graphe G et tout sommet, alors les exécutions de S et de $\sigma(S)$ sur G sont similaires.*

La question principale est donc : « Comment implémenter la compatibilité de pulse ? c'est-à-dire quand est-ce que u doit incrémenter $\textsc{Pulse}(u)$? »

Pour cela nous avons encore besoin de deux notions. Le sommet u est $\textsc{Safe}$ pour la phase p, et on note $\textsc{Safe}(u, p)$ le prédicat correspondant, si tous les messages originaux de la phase p envoyés de u à ses voisins ont été reçus. Le sommet u est $\textsc{Ready}$ pour la phase $p+1$, et on note $\textsc{Ready}(u, p+1)$ le prédicat correspondant, si tous ces voisins sont $\textsc{Safe}$ pour la phase p.

La proposition suivante nous indique lorsque $\textsc{Pulse}(u)$ peut être incrémenté.

Proposition 8 *Si* $p = \textsc{Pulse}(u)$ *et si* $\textsc{Safe}(u, p)$ *et* $\textsc{Ready}(u, p+1)$ *sont vraies, alors la compabilité de pulse est garantie.*

Notons qu'une seule des deux conditions ne suffit pas à garantir la compatibilité de pulse.

Preuve. ... [A FAIRE] □

6.2 Mesures de complexité

Grâce à la proposition 8 il suffit d'implémenter les propriétés $\textsc{Safe}(u, p)$ et $\textsc{Ready}(u, p+1)$. On a donc deux étapes à réaliser.

Étape Safe. À chaque message de la composante originale reçu on renvoie un « ack ». Ainsi l'émetteur u peut savoir lorsque tous ses messages de la phase $p = \textsc{Pulse}(u)$ ont été reçus.

Étape READY. Lorsqu'un sommet v devient SAFE pour la phase $p = \text{PULSE}(v)$ il diffuse cette information à ses voisins via un message « ready », mais pas forcément sur l'arête uv. Il indique ainsi à son voisin u la possibilité d'incrémenter $\text{PULSE}(u)$.

L'étape SAFE n'est pas très couteuse. Le temps et le nombre de messages sont au plus doublés. L'étape READY est généralement plus coûteuse puisque u peut, à une phase p donnée, recevoir potentiellement beaucoup plus de messages « ready » qu'il n'a émis de messages originaux. Par exemple, u pourrait ne pas avoir émis de messages originaux à la phase p, et pourtant u devra recevoir un message « ready » de ces voisins pour pouvoir incrémenter $\text{PULSE}(u)$ (cf. proposition 8).

Notons cependant qu'il n'y a aucune raison pour que v envoie son message « ready » à u directement sur l'arête uv. Certes c'est la solution la plus rapide de communiquer l'information. Mais si le temps n'est pas le critère principal on peut faire différemment et économiser des messages. Ce qui importe à u c'est de savoir quand tous ses voisins sont SAFE. Un autre sommet pourrait jouer le rôle de collecteur de plusieurs messages « ready ». Et, dans un deuxième temps, ce sommet pourrait informer u par un message différent, disons « pulse », que tous ses voisins sont SAFE pour la phase p. Globalement, à un moment donnée, il pourrait avoir m messages « ready » émis si tous les sommets sont SAFE en même temps, alors qu'il y a que seulement n sommets qui sont intéressés de recevoir un message « pulse ». Il serait donc envisageable de n'utiliser que n messages « pulse ».

Les synchroniseurs diffèrent sur l'étape READY, l'étape SAFE étant commune à tous les synchroniseurs que l'on va présenter.

On note respectivement $T_{init}(\sigma)$ et $M_{init}(\sigma)$ le temps et le nombre de messages pour initialiser la composante de synchronisation. Notons toute de suite que l'initialisation de σ est toujours au moins aussi couteuse qu'une diffusion car il faut au minimum que chaque sommet u exécute $\text{PULSE}(u) := 0$.

On note $M_{pulse}(\sigma)$ le nombre maximum de messages de la composante de synchronisation générés dans tout le graphe entre deux changements de pulse consécutifs, hormis les messages « ack » de l'étape SAFE. Si on ne compte pas ces messages dans $M_{pulse}(\sigma)$ c'est parce qu'ils correspondent toujours au nombre de messages envoyés pendant deux changements de pulse et que ce nombre peut être très variable d'une phase à une autre. Il dépend de S, pas vraiment de σ.

Enfin, on note $T_{pulse}(\sigma)$ est la durée (en unité de temps) entre le moment où le dernier sommet passe à la phase p et le moment où le dernier sommet passe à la phase $p+1$ (maximisé sur toutes les phases p). Attention ! Cela ne veut pas dire que pour un sommet u, la durée entre le moment où $\text{PULSE}(u) = p$ et où $\text{PULSE}(u) = p+1$ est borné par $T_{pulse}(\sigma)$. Cela pourrait être plus, car il est possible que le pulse de u soit passer de $p-1$ à p un peu avant le dernier.

Proposition 9 *Soit σ un synchroniseur implémentant les étapes* SAFE *et* READY*, et S un algorithme synchrone S sur G. Alors,*

- $\textsc{Temps}(\sigma(S), G) \leqslant T_{init}(\sigma) + T_{pulse}(\sigma) \cdot \textsc{Temps}(S, G)$.
- $\textsc{Message}(\sigma(S), G) \leqslant M_{init}(\sigma) + 2 \cdot \textsc{Message}(S, G) + M_{pulse}(\sigma) \cdot \textsc{Temps}(S, G)$.

Preuve. [A FAIRE] Le facteur 2 est pour les messages « ack » du synchroniseur. □

6.3 Deux synchroniseurs élémentaires

6.3.1 Synchroniseur α

...

Le synchroniseur α est efficace pour les graphes avec $m = O(n)$, comme les graphes de degré bornés, les graphes planaires, *etc.*

6.3.2 Synchroniseur β

6.4 Synchroniseur γ

Synchroniseur γ : Les complexité en Temps et Message sont chacune multipliée par un facteur $O(\log^3 n)$. D'où le résultat du calcul d'un BFS, car en synchrone il est possible de faire $\textsc{Temps} = O(D)$ et $\textsc{Message} = O(m)$.

Comme il n'existe pas de borne sur le temps mis par un message pour traverser un lien (et surtout les processeurs n'ont pas d'horloge capable de mesurer ce temps de manière absolu), il n'est pas possible pour un processeur d'attendre suffisament longtemps.

6.5 Compromis optimal

On peut se demander s'il n'existe pas de meilleur synchroniseur. En fait, on a :

Proposition 10 ([Awe85]) *Pour chaque entier $k > 0$, il existe des graphes G_k à n sommets où tout synchroneur σ tel que $\textsc{Temps}_{pulse}(\sigma) < k-1$ doit vérifier $\textsc{Message}_{pulse}(\sigma) > \frac{1}{4}n^{1+1/k}$.*

Preuve. La preuve est basé sur le fait que pour tout entier k il existe un graphe G_k avec n sommets et $\frac{1}{4}n^{1+1/k}$ arêtes où tout cycle est de longueur au moins k.

... □

La proposition précédente laisse l'espoir de trouver un synchroniseur σ avec $\text{TEMPS}_{pulse}(\sigma) = O(\log n / \log\log n)$ et $\text{MESSAGE}_{pulse}(\sigma) = O(\log n / \log\log n)$ en choisissant $k = \log n / \log\log n$. Même s'il existe, encore faudrait-il savoir le construire efficacement.

Au chapitre 9 on verra comment construire rapidement ces graphes à la base de la construction de ces synchroniseurs.

6.6 Exercices

Bibliographie

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4) :804–823, 1985.

[Gal82] Robert G. Gallager. Distributed minimum hop algorithms. Technical Report LIDS-P-1175, M.I.T., Cambridge, MA, January 1982.

CHAPITRE 7

Coloration

Dans ce chapitre nous allons voir comment partitionner les sommets d'un graphe en k ensembles indépendant, ce qui revient à calculer une k-coloration du graphe. L'enjeu ici est de réaliser cette tâche le plus rapidement possible, et d'étudier la nature locale de ce problème. Seule la complexité en temps importe. On se place dans les hypothèses du modèle LOCAL (voir la section 2.2).

7.1 Introduction

Casser la symétrie joue un rôle majeur en calcul distribué. Beaucoup de tâches nécessitent la collaboration d'un grand nombre de processeurs mais l'interdissent à certaines paires de processors. Dans certains cas, par exemple, il peut être interdit que des processeurs voisins agissent en concurrence.

Si l'on voit les pixels d'un écran de télévision comme une grille de processeurs élémentaires, le mode 1080i de la norme *HD Ready* impose que les lignes paires et les lignes impaires fonctionnement en parallèles mais différemment (1920×1080 interlacé) afin d'optimiser le flux vidéo et la vitesse d'affichage.

Un autre exemple typique est la construction et l'optimisation des tables de routage. On est souvent amené à partitionner le réseau en régions (ou *cluster*) et à choisir un processeurs particulier par région jouant le rôle de centres ou de serveur.

Dans certaines situations on impose même qu'il n'y ait qu'un seul processeur actif en même temps. C'est le problème de l'élection d'un *leader*, un problème classique en calcul distribué dans les réseaux anonymes. Dans le cadre de notre cours, cependant, nous supposons que nous avons des identités, donc on pourrait faire quelque chose comme « Si ID(u), alors LEADER(u) := VRAI ... » et ainsi distinguer un processeur explicitement de tous les autres. Il peut aussi s'agir du problème de l'exclusion mutuelle où une ressource ne doit être accédée que par une entité à la fois.

Dans ce chapitre nous allons nous intéresser à deux problèmes qui reviennent à casser la symétrie et qui sont de nature locale, à savoir la coloration et l'ensemble indépendant maximal. Ce dernier problème sera étudié plus précisément au chapitre suivant.

7.2 Définition du problème

Une *coloration propre* d'un graphe G est une fonction $c : V(G) \to \mathbb{N}$ telle que $c(u) \neq c(v)$ pour toute arête $\{u, v\} \in E(G)$. On dit que c est une *k-coloration* pour G si $c(u) \in \{0, \ldots, k-1\}$ pour tout sommet u de G.

Colorier un graphe revient à partitionner les sommets en ensembles indépendant. Les ensembles indépendant permettent de créer du parallélisme puisque tous les sommets d'un même ensemble peuvent interagir librement avec leurs voisins, qui par définition ne sont pas dans le même ensemble. Moins il y a de couleurs, plus le parallélisme est important.

Le *nombre chromatique* de G, noté $\chi(G)$, est le plus petit k tel que G possède une k-coloration. Déterminer si $\chi(G) \leqslant 2$ est linéaire (graphes bipartis ou composés de sommets isolés), alors que savoir si $\chi(G) = 3$ est un problème NP-complet même si G est un graphe planaire. Si G est planaire $\chi(G) \leqslant 4$.

Dans ce chapitre, on ne va pas s'interesser au calcul du nombre chromatique d'un graphe, mais plutôt essayer de déterminer le plus rapidement possible et de manière distribuée des colorations avec relativement peu de couleurs. Par « rapidement » on veut dire en temps significativement plus petit que le diamètre du graphe.

Les meilleurs algorithmes de coloration calculent une $(\Delta + 1)$-coloration où Δ est le degré maximum du graphe. Le meilleur d'entres eux [PS92] a une complexité en temps de $2^{O(\sqrt{\log n})}$, ce qui est sous-polynomial (plus petit que n^c pour toute constante $c > 0$). C'est donc potentiellement bien plus petit que le diamètre du graphe. Si $\Delta < 2^{O(\sqrt{\log n})}$, il est possible de faire mieux. Un autre algorithme [BE09] a une complexité en temps de $O(\Delta) + \frac{1}{2}\log^* n$ (on définira plus tard la fonction $\log^*$). Ce deuxième algorithme nécessite de connaître Δ, en plus de n.

7.3 Réduction de palette

Parmi les hypothèses du modèle LOCAL, chaque sommet u possède une identité unique, $\mathrm{ID}(u)$. Donc $c(u) = \mathrm{ID}(u)$ est trivialement une coloration propre, certes qui utilise potentiellement beaucoup de couleurs. Le but va être de réduire ce nombre tout en maintenant une coloration propre.

En séquentiel, il y a une technique très simple pour réduire le nombre de couleur. Initialement, $c(u) := \mathrm{ID}(u)$. Et pour chaque sommet u, pris dans un ordre quelconque, on modifie la couleur de u par la plus petite couleur non utilisée par un voisin de u. Voir la figure 7.1 pour une illustration.

Plus formellement, on définit FirstFree(u) comme étant le plus plus petit entier $k \geqslant 0$ tel que $k \notin \{c(v) : v \in N(u)\}$, où $N(u)$ représente l'ensemble des voisins u dans G. Il est clair que FirstFree(u) $\in \{0, \ldots, \deg(u)\}$. En séquentiel, on fait donc :

Algorithme FirstFit(G)

1. Pour tout sommet u, poser $c(u) := \mathrm{ID}(u)$.
2. Pour tout sommet u, poser $c(u) :=$ FirstFree(u).

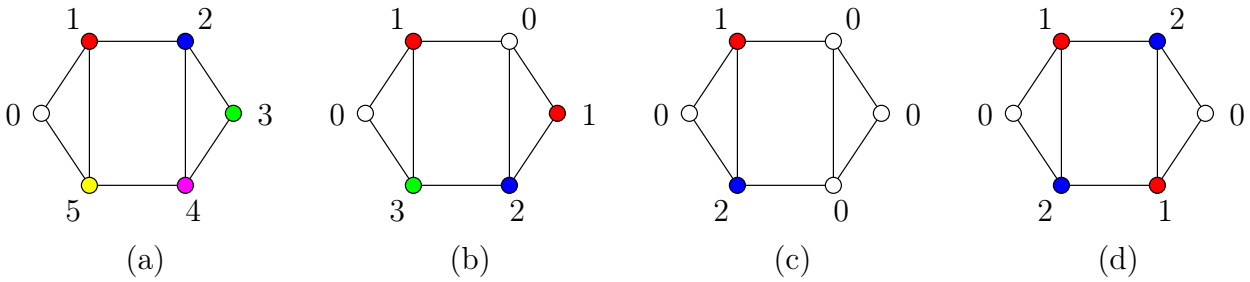


FIGURE 7.1 – Exemple de coloration suivant la règle FirstFree. En (a), une 6-coloration initiale. En (b) résultat en appliquant l'algorithme séquentiel avec l'ordre $0, 1, \ldots, 5$. En (c) résultat en appliquant FirstFree(u) en parallèle. Comme le montre (d) la coloration (b) n'est pas optimale.

Dans la suite on notera $\Delta(G) = \max\{\deg(u) : u \in V(G)\}$ le degré maximum de G.

Proposition 11 *Pour tout graphe G, $\chi(G) \leqslant \Delta(G) + 1$.*

Preuve. Il suffit d'analyser l'algorithme séquentiel FirstFit(G). Après l'étape 1, c est une coloration propre. Si, avant l'application de $c(u) :=$ FirstFree(u), c est une coloration propre, alors après c est encore une coloration propre. Donc, à la fin de l'algorithme, c est une coloration propre de G. On remarque aussi qu'après $c(u) :=$ FirstFree(u), $c(u) \in \{0, \ldots, \deg(u)\}$ (soit $\deg(u) + 1$ valeurs possibles), car il n'est pas possible que les voisins utilisent plus de $\deg(u)$ couleurs différentes. □

L'objectif est de trouver un algorithme distribué « rapide » pour calculer une $(\Delta(G)+1)$-coloration de G, ce qui est toujours possible grâce à la proposition 11. En terme de degré maximum, c'est la meilleure borne possible puisque pour la clique K_n à n sommets, on $\chi(K_n) = n$ alors que $\Delta(K_n) = n - 1$. Autrement dit, $\chi(K_n) = \Delta(K_n) + 1$.

L'algorithme FirstFit est intrinsèquement séquentiel. En distribué il y a danger à appliquer la règle FirstFree à tous les sommets, tout pouvant se dérouler en parallèle (cf. les sommets 2,3,4 de la figure 7.1(c)). Le problème est qu'en appliquant FirstFree à des sommets voisins il peut se produire que la couleur résultante soit la même. Il faut donc trouver une règle locale pour éviter ceci. On voit ici que l'idéale serait d'avoir justement une coloration et d'appliquer FirstFree aux sommets d'une même couleur pour éviter les collisions ...

En général on procède en essayant de réduire la palette de couleur en appliquant la règle FirstFree à certains sommets seulement. Initialement on utilise l'intervalle $[0, n-1]$, puis

ronde après ronde, on réduit jusqu'à l'intervalle $[0, \Delta(G)]$. À chaque ronde on garantit que la coloration reste propre.

L'algorithme distribué synchrone suivant permet plus généralement de réduire le nombre de couleurs de k_0 à k_1. On peut l'appliquer avec $k_0 = n$ et $k_1 = \Delta(G) + 1$. Mais plus tard on verra des exemples où c'est différent. On suppose donc qu'au départ c est une k_0-coloration et que les valeurs k_0, k_1 sont connues de tous les processeurs.

Algorithme REDUCEPALETTE(k_0, k_1, c)
(code du sommet u)

Pour k entier allant de $k_0 - 1$ à k_1 faire :

1. NEWPULSE
2. SEND$(c(u), v)$ à tous les voisins v de u
3. Si $c(u) = k$, alors
   (a) RECEIVE(c_i, v_i) pour tout voisin v_i de u
   (b) Poser $c(u) := \min\left\{t \geqslant 0 : t \notin \left\{c_1, \ldots, c_{\deg(u)}\right\}\right\}$

L'instruction NEWPULSE permet en mode synchrone de démarrer une nouvelle ronde, c'est-à-dire un nouveau cycle SEND/RECEIVE/calcule (voir les explications du chapitre 1). Dans un système asynchrone, cette instruction peut être vue comme une synchronisation, un appel à une routine du synchroniseur par exemple (voir le chapitre 6).

On rappelle que dans le mode synchrone, la complexité en temps d'un algorithme A peut se mesurer en nombre de rondes, chaque message mse transmettant un temps 1 entre voisins. Autrement dit, si A s'exécute en t rondes sur le graphe G, alors MESSAGE$(A, G) = t$.

Proposition 12 *Si $k_0 \geqslant k_1 \geqslant \Delta(G) + 1$, l'algorithme* REDUCEPALETTE(k_0, k_1, c) *construit en $k_0 - k_1$ rondes une k_1-coloration pour G ayant préalablement une k_0-coloration.*

Preuve. Il faut exactement $k_0 - k_1$ rondes pour exécuter REDUCEPALETTE(k_0, k_1, c) car k prend les $k_0 - k_1$ valeurs de l'intervalles $[k_1, k_0 - 1]$.

On remarque que la règle FIRSTFREE (instruction 3) est appliquée en parallèle que sur des sommets non adjacents, ceux de couleurs k. Donc après chacune de ces étapes parallèles, la coloration de G reste propre.

Les sommets u sur lesquels FIRSTFREE est appliquée ont une couleur initiale $c(u) = k \geqslant k_1 \geqslant \Delta(G) + 1$. Après cette application sur u, $c(u) \leqslant \deg(u) \leqslant \Delta(G) \leqslant k_1 - 1$. On a donc construit une k_1-coloration pour G, en temps $k_0 - k_1$. □

On peut donc construire une $(\Delta(G)+1)$-coloration en temps synchrone $O(n)$, en réduisant la palette de $k_0 = n$ à $k_1 = \Delta(G) + 1$ couleurs. Celà revient à traiter les sommets un par un en sautant les sommets de couleurs $\leqslant \Delta(G)$. On va voir comment faire beaucoup mieux, dans certains cas tout au moins.

7.4 Coloration des arbres

On va s'intéresser ici à la coloration des arbres. On va présenter un algorithme très rapide pour calculer une 3-coloration. Une 2-coloration serait envisageable pour les arbres (les sommets de niveau pair recoivent la couleur 0 les autres la couleur 1), mais en distribué cela nécessite un temps d'au moins la profondeur de l'arbre pour la calculer. On va voir qu'on peut aller beaucoup plus vite si l'on admet de perdre une couleur.

En fait, le même algorithme s'applique aussi aux cycles, et plus généralement aux graphes 1-orientable. On verra même qu'on peut facilement le généraliser aux graphes k-orientables dont voici la définition.

Définition 6 (k-orientation) *Une* k-orientation *d'un graphe G est une orientation de ses arêtes telle que tout sommet possède au plus k arcs sortant. Un graphe* k-orientable *est un graphe qui possède une k-orientation.*

Les arbres (et les forêts) sont clairement 1-orientables : il suffit de fixer un sommet comme racine et la relation $\text{PÈRE}(u)$ définit alors une 1-orientation, chaque sommet possédant au plus un père. Les cycles sont aussi 1-orientables (orientation définie par la relation successeur). C'est donc une famille qui capture à la fois les cycles et les arbres ou chemins.

Les graphes 1-orientables connexes sont les graphes qui possèdent au plus un cycle. Ils ressemblent donc à un cycle (éventuellement de longueur nulle) où à ses sommets pendent des arbres (cf. figure 7.2).

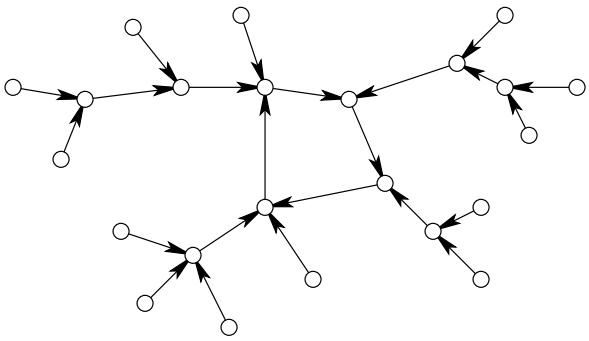


FIGURE 7.2 – Un graphe 1-orientable.

Les graphes k-orientables peuvent être vu comme l'union de graphes 1-orientables. Les graphes planaires sont 3-orientables, car leurs arêtes peuvent même être partitionner en trois forêts. Plus généralement, les graphes dont les arêtes peuvent être partitionnées en k forêts sont dits d'*arboricité* bornée par k. Les graphes d'arboricité k sont bien évidemment k-orientables. Il existe des algorithmes distribués efficaces pour calculer des $O(k)$-orientations pour un graphe d'arboricité k (voir la section 7.5).

7.4.1 Algorithme pour les 1-orientations

On va supposer que chaque sommet u connaît une 1-orientation de G via la variable Père(u). Si u n'a pas d'arc sortant (pas de père, ce qui arrive si G est un arbre), alors Père(u) $= \perp$.

L'idée de l'algorithme est de répéter un certain nombre de fois un calcul (via la fonction PosDiff décrite ci-après) entre la couleur de u (variable x) et celle de son père (variable y). À chaque étape le résultat de ce calcul devient la nouvelle couleur de u. Cette couleur définie à chaque étape une coloration propre de G. De plus, la plus grande couleur diminue fortement à chaque étape. Pour un sommet u sans père, on lui associe un père virtuel de couleur 0 ou 1 suivant que la couleur initiale de u (ID(u)) est $\neq 0$ ou $= 0$.

On suppose que les identités des sommets sont des entiers de $[0, n[$. La couleur finale du sommet u est stockée dans la variable Color(u). Comme on va le voir, la variable ℓ représente le nombre de bits dans l'écriture binaire de la plus grande couleur d'un sommet de G. On rappelle que pour écrire en binaire un entier quelconque de $[0, n[$ il faut $\lceil \log n \rceil$ bits. En effet, pour représenter un entier de $[0, 2^k[$ is faut k bits, et $k = \lceil \log n \rceil$ est le plus petit entier tel que $2^k \geqslant n$.

Cet algorithme est la version distribuée de l'algorithme de Cole et Vishkin [CV86] présenté originalement dans le modèle PRAM, un modèle du calcul parallèle. On remarque que dans l'algorithme suivant, la valeur de $\lceil \log n \rceil$ doit être connue de tous les sommets.

Algorithme Color6
(code du sommet u)

1. Poser $x := \mathrm{ID}(u)$ et $\ell := \lceil \log n \rceil$
2. Si Père(u) $= \perp$, alors $y := 1 - (x \bmod 2)$.
3. Répéter :
   (a) NewPulse
   (b) Send(x, v) à tout $v \neq$ Père(u) voisin de u
   (c) Si Père(u) $\neq \perp$, alors Receive(y, Père(u))
   (d) $x :=$ PosDiff(x, y)
   (e) $\ell' := \ell$ et $\ell := 1 + \lceil \log \ell \rceil$
   Jusqu'à ce que $\ell = \ell'$
4. Color(u) $:= x$

On note $\log x$ le logarithme en base deux de x, et $\lceil x \rceil$ la partie entière suppérieure de x.

7.4.2 La fonction PosDiff

Dans la suite on notera par bin(x) l'écriture binaire de l'entier x, et $|\mathrm{bin}(x)|$ sa longueur. Le bit de position p de bin(x), pour $p \in \{0, \ldots, |\mathrm{bin}(x)| - 1\}$, est le $p+1$-ième bit à partir

de la droite dans l'écriture standard de $\text{bin}(x)$. Par exemple, $\text{bin}(6) = 110$, et $\text{bin}(6)[0] = 0$, $\text{bin}(6)[1] = \text{bin}(6)[2] = 1$.

La fonction $\textsc{PosDiff}(x, y)$, définie pour des entiers $x \neq y \in \mathbb{N}$, est calculée comme suit :

1. On calcule les écritures binaires de x et y, notées B_x et B_y, la plus courte des deux chaînes étant complétée par des 0 à gauche de sorte qu'elles aient même longueur, et on calcule une position p telle que $B_x[p] \neq B_y[p]$.
2. $\textsc{PosDiff}(x, y)$ est alors la valeur dont l'écriture binaire est $\text{bin}(p) \circ B_x[p]$ (soit p écrit en binaire suivit d'un bit 0 ou 1).

Ce qui importe pour le point 1. c'est que tous les sommets calculent la position suivant la même convention (la plus petite, la plus grande, peu importe).

Exemple (en choisissant pour p la plus petite valeur possible) :
- $\textsc{PosDiff}(10, 4) = \textsc{PosDiff}(1010_2, 100_2) = 11_2 = 3$ car $p = 1$ et $b = 1$.
- $\textsc{PosDiff}(6, 22) = \textsc{PosDiff}(110_2, 10110_2) = 1000_2 = 8$ car $p = 4 = 100_2$ et $b = 0$.
- $\textsc{PosDiff}(4, 0) = \textsc{PosDiff}(100_2, 0_2) = 101_2 = 5$ car $p = 2 = 10_2$ et $b = 1$.

Notons qu'il est possible que $\textsc{PosDiff}(x, y) = y$ ou que $\textsc{PosDiff}(x, y) = x$. C'est le cas de (et c'est sans doute les seuls cas) :
- $\textsc{PosDiff}(4, 2) = \textsc{PosDiff}(100_2, 10_2) = 10_2 = 2$.
- $\textsc{PosDiff}(8, 4) = \textsc{PosDiff}(1000_2, 100_2) = 100_2 = 4$.
- $\textsc{PosDiff}(1, 0) = \textsc{PosDiff}(1_2, 0_2) = 1_2 = 1$.
- $\textsc{PosDiff}(0, 1) = \textsc{PosDiff}(0_2, 1_2) = 0_2 = 0$.
- $\textsc{PosDiff}(3, 5) = \textsc{PosDiff}(11_2, 101_2) = 11_2 = 3$.
- $\textsc{PosDiff}(5, 9) = \textsc{PosDiff}(101_2, 1001_2) = 101_2 = 5$.

En utilisant le fait que $p \in [0, k[$ où $k = |\text{bin}(\max\{x, y\})|$, et donc que $|\text{bin}(p)| \leqslant \lceil \log k \rceil$, on a alors par construction que :

$$|\text{bin}(\textsc{PosDiff}(x, y))| \;=\; 1 + |\text{bin}(p)| \;\leqslant\; 1 + \lceil \log(|\text{bin}(\max\{x, y\})|) \rceil \;. \qquad (7.1)$$

La propriété essentielle de la fonction PosDiff est la suivante :

Propriété 1 *Soient $x, y, z \in \mathbb{N}$. Si $x \neq y$ et $y \neq z$, alors* $\textsc{PosDiff}(x, y) \neq \textsc{PosDiff}(y, z)$.

Preuve. Supposons que $x \neq y$ et $y \neq z$. Soit (p, b) la position et le bit obtenus dans le calcul de $\textsc{PosDiff}(x, y)$ et (p', b') dans le calcul de $\textsc{PosDiff}(y, z)$. Si $\textsc{PosDiff}(x, y) = \textsc{PosDiff}(y, z)$ alors $p = p'$ et $b = b'$ car $|b| = |b'| = 1$ (si les longueurs de b et b' avaient été différentes cela pourrait être faux). Si $p = p'$, alors $b = B_x[p]$ et $b' = B_y[p'] = B_y[p]$. Mais x et y diffèrent à la position p, donc $B_x[p] \neq B_y[p]$ et $b \neq b'$: contradiction. Donc $\textsc{PosDiff}(x, y) \neq \textsc{PosDiff}(y, z)$. □

Cette propriété garantit que la coloration reste propre tant que chaque sommet u de couleur x utilise $\textsc{PosDiff}(x, y)$ avec un voisin de couleur $y \neq x$. En particulier si y est la

couleur du père de u. Si u n'a pas de père c'est encore valable car on utilise, tout au long du calcul, une valeur $y \neq x$. Plus précisément :

- si x est pair, alors $y = 1$ et $\textsc{PosDiff}(x, 1) = 0$ (ce qui reste pair) ;
- si x est impair, alors $y = 0$ et $\textsc{PosDiff}(x, 0) = 1$ (ce qui reste impair).

Donc après le premier calcul de PosDiff, un sommet sans père ne change plus de couleur qui devient 0 ou 1.

7.4.3 Analyse de l'algorithme

Dans la suite on supposera que $n \geqslant 2$, c'est-à-dire possède au moins une arête. Pour l'analyse du temps, on définit $x_i(u)$ et ℓ_i comme les valeurs de x du sommet u, et de ℓ à la fin de la i-ème boucle « répéter ». (La valeur de ℓ ne dépend pas du sommet u, elle est commune à tous les sommets.) On a :

- $x_0(u) = \mathrm{ID}(u)$.
- $\ell_0 = \lceil \log n \rceil$.
- $\ell_1 = 1 + \lceil \log \lceil \log n \rceil \rceil$.
- $\ell_2 = 1 + \lceil \log (1 + \lceil \log \lceil \log n \rceil \rceil) \rceil$.
- *etc.*

La proposition suivante va nous montrer que l'algorithme termine toujours.

Proposition 13 *Si $\ell_{i-1} > 3$, alors $\ell_i < \ell_{i-1}$. Sinon $\ell_i = \ell_{i-1}$.*

Preuve. On a $\ell_i = 1 + \lceil \log \ell_{i-1} \rceil < 2 + \log \ell_{i-1}$ (car $\lceil x \rceil < 1 + x$). La proposition est donc vraie dès que $2 + \log \ell_{i-1} \leqslant \ell_{i-1}$, c'est-à-dire lorsque $4\ell_{i-1} \leqslant 2^{\ell_{i-1}}$. C'est vrai si $\ell_{i-1} \geqslant 4$. Donc si $\ell_{i-1} > 3$, alors $\ell_i < \ell_{i-1}$.

Si $\ell_{i-1} = 3$, alors $\ell_i = 1 + \lceil \log 3 \rceil = 3$. Si $\ell_{i-1} = 2$, alors $\ell_i = 1 + \lceil \log 2 \rceil = 2$. Et si $\ell_{i-1} = 1$, alors $\ell_i = 1 + \lceil \log 1 \rceil = 1$. Il n'est pas possible d'avoir $\ell_{i-1} = 0$, car $n > 1$. □

On déduit de la proposition 13 que l'algorithme Color6 termine toujours puisque soit $\ell_0 > 3$ et alors la valeur de ℓ diminue strictement. Soit $\ell_{i-1} \leqslant 3$, et alors la valeur suivante ℓ_i reste aussi à ℓ_{i-1}. Donc on sort toujours le la boucle « Répéter », le test « $\ell = \ell'$ » étant alors vrai.

Montrons maintenant qu'il calcule une 6-coloration. Pour ceci, montrons d'abord que ℓ_i borne supérieurement la longueur de l'écriture binaire de toutes les couleurs du graphes. Dit autrement :

Proposition 14 *À chaque étape i et chaque sommet u, $|\mathrm{bin}(x_i(u))| \leqslant \ell_i$.*

Preuve. Par induction sur i, pour un sommet arbitraire u.

Pour $i = 0$, on a $x_0(u) < n$, et donc $|\mathrm{bin}(x_0(u))| \leqslant \lceil \log n \rceil$. Comme $\ell_0 = \lceil \log n \rceil$, on a bien $|\mathrm{bin}(x_0(u))| \leqslant \ell_0$.

Supposons que $i > 0$. On a vu que si u n'a pas de père, alors $x_i(u) \in \{0, 1\}$ une fois passée la première étape. Donc $|\text{bin}(x_i(u))| = 1 \leqslant \ell_i$ (on a bien $\ell_i \geqslant 1$ pour tout $i > 0$). Supposons donc que u possède un père $v = \text{PÈRE}(u)$.

On a $x_{i+1}(u) = \text{POSDIFF}(x_i(u), x_i(v))$. Par l'équation 7.1, $|\text{bin}(x_{i+1}(u))| \leqslant 1 + \lceil \log(|\text{bin}(\max\{x_i(u), x_i(v)\})|) \rceil$. Par induction, $|\text{bin}(x_i(u))|$ et $|\text{bin}(x_i(v))|$ sont $\leqslant \ell_i$, et donc $|\text{bin}(\max\{x_i(u), x_i(v)\})| \leqslant \ell_i$. Donc $|\text{bin}(x_{i+1}(u))| \leqslant 1 + \lceil \log \ell_i \rceil = \ell_{i+1}$ par définition de ℓ_{i+1}. □

Proposition 15 *À la fin de la dernière étape i de l'algorithme, $x_i(u) \in \{0, \ldots, 5\}$ pour tout sommet u.*

Preuve. D'après la proposition 13, l'algorithme se termine avec $\ell_i \leqslant 3$. Si $\ell_i \leqslant 2$, alors d'après la proposition 14, on a $|\text{bin}(x_i(u))| \leqslant 2$. Donc $x_i(u) \in \{0, \ldots, 3\}$. Si $\ell_i = 3$, c'est que $\ell_{i-1} = 3$ sinon on ne s'arrête pas à l'étape i mais $i + 1$. Donc, toujours d'après la proposition 14 appliquée à $i - 1$, pour tous les sommets u, $x_{i-1}(u) \in \{0, \ldots, 7\}$ (puisque $|\text{bin}(x_{i-1}(u)| \leqslant 3$). On vérifie alors facilement que $\text{POSDIFF}(x, y) \leqslant 5$ pour tous les entiers $x \neq y \in \{0, \ldots, 7\}$. (Pour la paire (p, b) qui définit $\text{POSDIFF}(x, y)$ on a au pire $p = 10_2$ et $b = 1$.) Donc $x_i(u) \leqslant 101_2 = 5$. □

On a vu, grâce à la propriété 1, que l'algorithme maintient avec $x_i(u)$ une coloration propre. D'après la proposition 15 il s'agit d'une 6-coloration. On va maintenant borner le nombre de rondes de COLOR6.

On note

$$\log^{(i)} \ell \;=\; \overbrace{\log \log \log \cdots \log}^{i \text{ fois}} \ell$$

la fonction log itérée i fois. On a $\log^{(0)} \ell = \ell$, $\log^{(1)} \ell = \log \ell$, $\log^{(2)} \ell = \log \log \ell$, *etc.*

Proposition 16 *Pour tout $i > 0$, si $\ell_{i-1} > 3$, alors $\ell_i \leqslant 2 + \lceil \log^{(i+1)} n \rceil$.*

Preuve. On va utiliser le fait suivant :

Fait 1 *Soit $f : \mathbb{R} \to \mathbb{R}$ une fonction croissante telle que $f^{-1}(n) \in \mathbb{Z}$ pour tout $n \in \mathbb{Z}$. Alors, pour tout $x \in \mathbb{R}$, $\lceil f(\lceil x \rceil) \rceil = \lceil f(x) \rceil$ et $\lfloor f(\lfloor x \rfloor) \rfloor = \lfloor f(x) \rfloor$.*

En effet, soit $x \in \mathbb{R}$ et $n = \lceil f(x) \rceil \in \mathbb{Z}$. Par la croissance de f et de $\lceil \cdot \rceil$, $f(x) \leqslant n \leqslant \lceil f(\lceil x \rceil) \rceil$. Montrons que $\lceil f(\lceil x \rceil) \rceil \leqslant n$. La fonction f^{-1} est croissante donc $x \leqslant f^{-1}(n)$, d'où $\lceil x \rceil \leqslant \lceil f^{-1}(n) \rceil = f^{-1}(n)$ puisque que ce dernier est un entier. En réappliquant f, il suit que $f(\lceil x \rceil) \leqslant n$. D'où, $\lceil f(\lceil x \rceil) \rceil \leqslant \lceil n \rceil = n$. Par conséquent $n = \lceil f(\lceil x \rceil) \rceil$. Pour la partie entière inférieure, le raisonement est similaire en posant $n = \lfloor f(x) \rfloor$ et en inversant toutes les inégalités.

Le fait 1 s'applique par exemple aux fonctions $x \mapsto x/i$, $x \mapsto x^{1/i}$, $x \mapsto \log^{(i-1)} x$ pour chaque $i \in \mathbb{N}^*$. En particulier, $\lceil \log^{(i)}(\lceil \log n \rceil) \rceil = \lceil \log^{(i+1)} n \rceil$.

On va montrer la proposition 16 par induction, et donc soit

$$P(i) = \text{« si } \ell_{i-1} > 3, \text{ alors } \ell_i \leqslant \lceil \log^{(i+1)} n \rceil. \text{ »}$$

Pour $P(1)$ (il faut $i > 0$, donc l'induction commence à $i = 1$), on a $\ell_1 = 1 + \lceil \log \ell_0 \rceil = 1 + \lceil \log \lceil \log n \rceil \rceil = 1 + \lceil \log \log n \rceil$ en appliquant le fait 1 (on a $\ell_0 > 3$ donc $\log \ell_0$ est bien défini). Or $1 + \lceil \log \log n \rceil \leqslant 2 + \lceil \log^{(2)} n \rceil$. Donc, si $\ell_0 > 3$, on a $\ell_1 \leqslant 2 + \lceil \log^{(2)} n \rceil$. Et $P(1)$ est vraie.

Supposons que $P(i-1)$ soit vraie. On veut prouver $P(i)$ pour $i \geqslant 2$. Si $\ell_{i-1} > 3$, alors $\ell_{i-2} > 3$ (par la propriété 13). Donc, par $P(i-1)$, $\ell_{i-1} \leqslant 2 + \lceil \log^{(i)} n \rceil$. En particulier $3 < \ell_{i-1} \leqslant 2 + \lceil \log^{(i)} n \rceil$. On en déduit que $2 + \lceil \log^{(i)} n \rceil \leqslant 2 \cdot \lceil \log^{(i)} n \rceil$, car pour tout $x \in \mathbb{R}$, si $4 \leqslant 2 + x$, alors $2 + x \leqslant 2x$. Au final, $\ell_{i-1} \leqslant 2 \cdot \lceil \log^{(i)} n \rceil$.

D'après l'algorithme, si $\ell_{i-1} > 3$,

$$\begin{aligned}
\ell_i \;=\; 1 + \lceil \log \ell_{i-1} \rceil \;&\leqslant\; 1 + \left\lceil \log \left(2 \cdot \left\lceil \log^{(i)} n \right\rceil \right) \right\rceil \\
&\leqslant\; 1 + \left\lceil \log(2) + \log \left\lceil \log^{(i)} n \right\rceil \right\rceil \\
&\leqslant\; 2 + \left\lceil \log \left\lceil \log^{(i)} n \right\rceil \right\rceil \\
&\leqslant\; 2 + \left\lceil \log^{(i+1)} n \right\rceil \quad \text{(par le fait 1)}
\end{aligned}$$

ce qui montre que $P(i)$ est vraie. □

Dans la suite on note[1] $\log^* n = \min\{i \geqslant 0 : \log^{(i)} n \leqslant 1\}$. Dit autrement c'est le plus petit entier i tel que

$$n \;\leqslant\; 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\} \, i \, .$$

Autant dire que la fonction $\log^* n$ croît extrêmement lentement. Par exemple,

$$\log^* \left(2^{65536}\right) = \log^* \left(2^{2^{16}}\right) = \log^* \left(2^{2^{2^{2}}}\right) = 5 \, .$$

On rappelle que le nombre de particules de l'univers est estimé à seulement 2^{400}. Raisonablement, le $\log^*$ de n'importe nombre de processeurs ne dépassera jamais 5.

7.4.4 Résumé

Lemme 1 *L'algorithme* Color6 *calcule une* 6*-coloration en temps* $\log^* n$ *pour tout graphe à* n *sommets ayant une* 1*-orientation.*

1. Parfois $\log^* n$ est défini comme le plus petit $i \geqslant 0$ tel que $\log^{(i)} n \leqslant 2$.

Preuve. On a vu dans la proposition 15 que COLOR6 calculait une 6-coloration. Soit t le nombre de boucles « Répéter » réalisées par l'algorithme. C'est le temps (le nombre de rondes) pris par l'algorithme.

Le résultat est vrai si $\ell_0 \leqslant 3$. En effet, d'après la proposition 13, si $\ell_0 \leqslant 3$, alors $\ell_1 = \ell_0$ et donc l'algorithme ne prend alors qu'une seule ronde. Et $\log^* n$ est au moins 1 dès que $n > 1$.

Supposons que $\ell_0 > 3$. Considérons la fin de l'étape $i = (\log^* n) - 1$. Notons que $i > 0$, car si $i = 0$ c'est que $n \leqslant 4$, et dans ce cas $\ell_0 = \lceil \log n \rceil = 2$ ce qui n'est pas nôtre cas. Notons aussi que $2 + \left\lceil \log^{(i+1)} n \right\rceil \leqslant 3$. En effet,

$$\begin{aligned} 2 + \left\lceil \log^{(i+1)} n \right\rceil \;\leqslant\; 3 \quad &\Leftrightarrow \quad \left\lceil \log^{(i+1)} n \right\rceil \;\leqslant\; 1 \\ &\Leftrightarrow \quad \log^{(i+1)} n \;\leqslant\; 1 \quad \Leftrightarrow \quad i+1 \;=\; \log^* n \;. \end{aligned}$$

Si $\ell_{i-1} \leqslant 3$, alors d'après la proposition 13, $\ell_i = \ell_{i-1}$ et l'algorithme s'arrête à la fin de l'étape $i = (\log^* n) - 1$. Donc $t = (\log^* n) - 1$.

Si $\ell_{i-1} > 3$, la proposition 16 s'applique ($i > 0$) et donc $\ell_i \leqslant 2 + \left\lceil \log^{(i+1)} n \right\rceil \leqslant 3$. D'après la proposition 13, $\ell_{i+1} = \ell_i$ et donc l'algorithme s'arrête à la fin de l'étape $i + 1 = \log^* n$.

Dans les deux cas $t \leqslant \log^* n$. □

En fait, dans l'analyse de la validité de l'algorithme COLOR6 on s'aperçoit qu'on n'utilise seulement le fait que les identités des sommets de G définissent initialement une n-coloration. Tout reste valable si l'on part d'une m-coloration avec m quelconque, *a priori* $m < n$. Il suffit de poser $\ell := \lceil \log m \rceil$ dans l'instruction 1 de l'algorithme COLOR6.

7.4.5 De six à trois couleurs

Pour obtenir la 3-coloration finale on procède en deux étapes :

1. on s'arrange pour avoir une 6-coloration où tous les sommets ayant le même père obtiennent la même couleur (procédure SHIFTDOWN).
2. On applique REDUCEPALETTE pour éliminer les couleurs $3, 4, 5$ et ne garder que $0, 1, 2$.

Au total cela rajoute 6 rondes. Si pour toute valeur raisonable de n, $\log^* n \leqslant 5$, c'est donc le passage de 6 à 3 couleurs qui est le plus coûteux, le passage de n à 6 couleurs nécessitant lui que 5 étapes !

La procédure SHIFTDOWN (non explicitée ici) consiste à recolorier chacun des sommets par la couleur de son père. Si le sommet n'a pas de père, il se colorie avec la plus petite couleur différente de lui-même (soit la couleur 0 ou 1). Pour implémenter la procédure

ShiftDown on envoit donc sa couleur à tous ses voisins dont on est leur père (à tous ses fils donc), puis on se recolorier avec la couleur reçue. Visuellement, c'est comme si les couleurs se décalaient d'un étage vers les fils (d'où le nom de ShiftDown).

Après un ShiftDown, chaque sommet u "voit" alors deux couleurs parmi ses voisins : celle de son père et celle de ses fils (puisqu'ils ont tous la même). On peut donc obtenir trois couleurs avec ReducePalette.

Algorithme Color6to3

Pour $k := 3, 4, 5$ faire :

1. ShiftDown
2. ReducePalette$(k+1, k, \text{Color})$

L'ordre d'élimination des couleurs n'est pas important. Cependant, il est très important que tous les processeurs terminent en même temps l'algorithme précédent Color6. C'est bien le cas, la variable ℓ étant identique au cours du temps sur tous les processeurs.

D'après la proposition 12, ReducePalette$(k+1, k, \text{Color})$ prend une ronde et élimine la couleur k.

D'où le résultat final suivant :

Théorème 1 *On peut calculer une* 3*-coloration en temps* $\log^* m + 6$ *pour tout graphe* 1*-orientable possédant une* m*-coloration.*

Deux questions restent en suspens :

1. Comment s'affranchir de la connaissance de n dans l'algorithme Color6 ? On pourrait initialiser $\ell_0 := \text{ID}(u)$ par exemple. Il faut alors modifier la fonction PosDiff, car dans sa forme actuelle il est possible que $\text{PosDiff}(x, y) = y$. Celà pose problème pour recolorier dans ce cas x en y si jamais y ne change plus. Ceci dit, celà se produit dans un nombre limité de cas. En particulier, on vu que celà ne se produit pas lorsque $y \in \{0, 1\}$ (abscence de père).
2. Comment enchaîner avec l'algorithme Color6to3 si tous les sommets ne terminent pas en même temps ?

Les algorithmes distribués qui s'affranchissent de toute information globale sur le graphe (nombre de sommets, degré maximum, ...) sont dits *uniformes*. Dans certaines conditions il est possible de rendre uniforme un algorithme qui initialement ne l'est pas [KSV11]. Voir aussi l'excellent survol [Suo11] sur les algorithmes locaux et le modèle LOCAL.

7.5 Coloration des k-orientations

On peut montrer que les graphes de degré maximum Δ sont $\lceil \Delta/2 \rceil$-orientables. Les graphes planaires sont 3-orientables, et plus généralement, les graphes k-dégénérés sont k-

orientables. (Les graphes k-dégénérés ont la propriété que tout sous-graphe induit possède un sommet de degré au plus k. En enlevant ce sommet, on crée au plus k relations de parentés qui ne peuvent produire de cycle par répétition de cette opération.)

Bien sûr la question est de pouvoir calculer rapidement une k-orientation d'un graphe k-orientable *from scratch*. C'est un autre problème en soit. Il existe des algorithmes distribués sophistiqués pour cela. Le meilleur d'entres eux prend un temps $O(\log n)$ [BE08].

Pour les graphes k-orientables, on s'intéresse à calculer une $(2k+1)$-coloration. Donc une 3-coloration pour les 1-orientables, une 5-coloration pour les 2-orientables, *etc.* En effet, comme on va le voir, pour ces graphes on peut toujours produire une $2k+1$-coloration, et puis il y a des graphes k-orientables sans $2k$-coloration. Par exemple, le graphe complet à $2k+1$ sommets, K_{2k+1}, est k-orientable et de nombre chromatique $2k+1$. En effet, on peut toujours décomposer un K_{2k+1} en $2k$ cycles Hamiltoniens, ce qui a été démontré par Walecki dans les années 1890 (cf. [Als08]). Voir l'exemple de la figure 7.3 pour $k=2$.

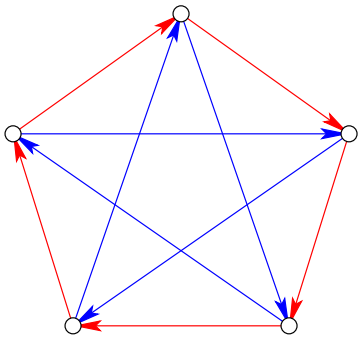


FIGURE 7.3 – Graphe 2-orientable nécessitant une 5-coloration.

Théorème 2 *On peut calculer une $(2k+1)$-coloration en temps $k \log^* m + 2^{O(k)}$ pour tout graphe possédant une k-orientation et une m-coloration.*

Preuve. On suppose ici que chaque sommet u possède k variables $\text{PÈRE}_1(u), \ldots, \text{PÈRE}_k(u)$ représentant les au plus k pères du sommets u (on pose $\text{PÈRE}_i(u) = \perp$ si le i-ème père de u n'existe pas).

[A FINIR]

La couleur d'un sommet u est alors un vecteur $(x_1, \ldots, x_k)$ où un POSDIFF est appliqué sur chaque composante x_i et la couleur y_i reçue de $\text{PÈRE}_i(u)$. ...

Bien sûr, l'algorithme se termine sur un vecteur où chaque composante est dans $\{0, \ldots, 5\}$. Potentiellement, cela fait 6^k vecteurs différents possibles et donc couleurs possibles en tout sur le graphe. La réduction REDUCEPALETTE prend donc malheureusement un temps $6^k - (2k+1) = 2^{O(k)}$ pour réduire à $2k+1$ couleurs.

On obtient $2k+1$ couleurs par REDUCEPALETTE car, parmi ses voisins, on a k pères et k ensembles de sommets dont on est le PÈRE$_i$. Chaque ensemble de sommets (dont on est le PÈRE$_i$) peuvent être colorié par la même couleurs grâce à un SHIFTDOWN. □

Remarquons qu'il est trivial de calculer une Δ-orientation à partir d'un graphe sans orientation particulière initiale. En utilisant le résultat précédent on en déduit une $(2\Delta+1)$-coloration. Δ rondes supplémentaires, grâce à une réduction de $k_0 = 2\Delta+1$ à $k_1 = \Delta+1$ couleurs, permettent d'obtenir une $(\Delta+1)$-coloration. D'où :

Corollaire 2 *On peut calculer une $(\Delta+1)$-coloration en temps $\Delta \log^* m + 2^{O(\Delta)}$ pour tout graphe de degré maximum Δ possédant une m-coloration.*

Notons que Δ et m doivent être connus de tous les processeurs. Il a été conjecturé par [SV93] que tout algorithme procédant itérativement par réduction de couleur (c'est-à-dire qui applique succésivement la même boucle de calcul à une coloration propre – comme la fonction POSDIFF dans COLOR6) ne peut pas produire une $O(\Delta)$-coloration plus rapidement que $\Omega(\Delta \log \Delta)$ rondes.

Le meilleur algorithme connu [BE09] pour les graphes de degré maximum Δ permet de calculer une $(\Delta+1)$-coloration en temps $O(\Delta) + \frac{1}{2}\log^* m$. Cet algorithme ne procède pas, bien sûr, par réduction itérative de couleurs. Il utilise par exemple des colorations qui ne sont pas toutes à fait propres : le sous-graphe induit par chaque classe de couleurs soit de degré maximum borné. (Dans une coloration propre ces sous-graphe doivent être des stables, donc de degré maximum 0.)

7.6 Coloration des cycles, borne inférieure

7.6.1 Coloration des cycles

Nous avons vu que les cycles sont 1-orientables, et à ce titre on peut leur appliquer le résultat de la proposition 1. On peut ainsi calculer une 3-coloration en temps $\log^* n + O(1)$. On va présenter une amélioration de l'algorithme dans le cas des cycles permettant d'aller deux fois plus vite. On va supposer que les cycles sont orientés, c'est-à-dire que les sommets de droit et de gauche sont connus de chaque sommet u par les relations SUCC(u) et PRED(u).

L'idée est d'imaginer qu'on calcule POSDIFF en alternant avec son successeur et son prédecesseur. À chaque ronde, si tous les sommets décident de calculer POSDIFF avec leur successeur, alors la coloration reste propre, car c'est comme si on avait fixer PÈRE(u) = SUCC(u) dans l'algorithme COLOR6. De même, si tous décident de faire POSDIFF avec leur prédécesseur. La coloration reste propre même si on alterne les rondes où le père est le successeur avec celles où le père est le prédécesseur. De manière générale, à chaque ronde

on peut fixer arbitrairement le père de chaque sommet, du moment que cela définisse une 1-orientation. Par exemple, les rondes paires avec son successeur, les rondes impaires avec son prédécesseur. On va voir que le deuxième calcul peut être calculer directement par u, sans avoir à recommuniquer.

Considérons trois sommets consécutifs de couleurs $z - x - y$. En appliquant PosDiff aux successeurs $z \to x \to y$, on obtient les couleurs $z' - x' - y'$ avec $z' = \textsc{PosDiff}(z, x)$ et $x' = \textsc{PosDiff}(x, y)$. C'est une coloration propre. Si maintenant on applique PosDiff aux prédecesseurs $z' \leftarrow x' \leftarrow y'$, on obtient les nouvelles couleurs $z'' - x'' - y''$ avec $x'' = \textsc{PosDiff}(z', x')$. On a donc :

$$x'' \;=\; \textsc{PosDiff}(z', x') = \textsc{PosDiff}(\textsc{PosDiff}(z, x), \textsc{PosDiff}(x, y)) \;.$$

C'est finalement une fonction qui ne dépend que de x, y, z. Ces valeurs sont connues dès la première communication. La seconde étape de communication (avec les prédécesseurs) est en fait inutile pour calculer x''. On peut donc économiser la moitié des communications.

Algorithme ColorRing
(code du sommet u)

1. Poser $x := \mathrm{ID}(u)$ et $\ell := \lceil \log n \rceil$
2. Répéter :
   (a) NewPulse
   (b) $\textsc{Send}(x, \textsc{Succ}(u))$ et $\textsc{Send}(x, \textsc{Pred}(u)))$
   (c) $\textsc{Receive}(y, \textsc{Succ}(u))$ et $\textsc{Receive}(z, \textsc{Pred}(u)))$
   (d) $x := \textsc{PosDiff}(\textsc{PosDiff}(z, x), \textsc{PosDiff}(x, y))$
   (e) $\ell' := \ell$ et $\ell := 1 + \lceil \log (1 + \lceil \log \ell \rceil) \rceil$

   Jusqu'à ce que $\ell = \ell'$
3. Poser $\textsc{Color}(u) := x$
4. $\textsc{ReducePalette}(6, 3, \textsc{Color})$

D'où le résultat qu'on ne démontrera pas :

Proposition 17 *Lalgorithme* ColorRing *produit une* 3*-coloration en temps* $\lceil \frac{1}{2} \log^* m \rceil + 3$ *sur les cycles orientés possédant une* m*-coloration.*

La borne la plus précise connue sur le temps pour un cycle orienté muni d'une m-coloration est de $\lceil \frac{1}{2} \log^* m \rceil + 1$ [Ryb11, Théorème 8.3, p. 71]. Il est cependant possible de faire seulement 3 rondes si les identifiants sont sur des adresses d'au plus 2048 bits ($m = 2^{2048}$). Il faut faire une ronde classique, puis résoudre le problème en deux rondes par une méthode *ad hoc* sur les 24 couleurs qui restent. Notons que d'après la proposition 17, il aurait fallut 6 rondes au lieu de 3 pour $m = 2^{2048}$ (car $\log^* 2^{2048} = 5$).

Notons qu'en pratique, les adresses sont plutôt sur 48 bits, et non 2048. De plus, si les voisins ont des adresses assignées de manière aléatoires uniformes, la probabilité d'avoir au moins un de ses Δ voisins avec la même adresse n'est jamais que $1-(1-1/m)^{\Delta} \approx 1-e^{-\Delta/m}$ ce qui pour $m = 2^{48}$ et $\Delta = 2$ comme pour un cycle est d'environ $7 \cdot 10^{-15}$.

7.6.2 Cycle non orienté

...

7.6.3 Borne inférieure

Dans cette partie on se pose la question de l'optimalité des algorithmes précédant. Peut-on colorier un arbre ou un cycle plus rapidement encore, en $o(\log^* n)$ rondes ? La réponse est non pour les cycles et certains arbres.

Attention ! cela signifie que pour tout algorithme de 3-coloration A, il existe des instances de cycles (ou d'arbres) pour lesquels A prends $\Omega(\log^* n)$ rondes. Ce type de résultat ne dit pas que A ne peut pas aller plus vite dans certains cas. Par exemple, si l'arbre a un diamètre plus petit que $\log^* n$ (comme une étoile), on peut évidemment faire mieux. De même, si les identités des sommets du cycles ont une distribution particulière (par exemple les identités sont $1-2-\cdots-n$) on peut faire mieux (et même sans communication ici !). On peut même imaginer en ensemble d'algorithmes $A_1, A_2, \ldots$ chacun permettant de faire mieux pour certaines instances. Malheureusement, pour une instance donnée, il n'y a pas vraiment de moyens de déterminer rapidement et de manière distribuée le bon algorithme A_i à appliquer.

On va démontrer précisément une borne inférieure pour les cycles, l'idée étant que pour un chemin (qui est un arbre particulier) on ne peut pas aller vraiment plus vite que pour un cycle. On réduit le problème de coloration des cycles à celui des chemins.

Proposition 18 *Soient $P(n)$ et $C(n)$ les complexités respectives en temps pour la 3-coloration des chemins et des cycles orientés à n sommets. Alors $C(n)-1 \leqslant P(n) \leqslant C(n)$.*

Preuve. L'inégalité $P(n) \leqslant C(n)$ est évidente, car l'algorithme des cycles peut toujours être appliqué à un chemin. La coloration sera propre, en modifiant éventuellement le code des deux processeurs extrémités pour qu'il simule la communication avec leur successeur et prédécesseur qui n'existe pas.

Montrons maintenant que $C(n) - 1 \leqslant P(n)$. Pour cela, on imagine qu'on exécute sur un cycle un algorithme de 3-coloration d'un chemin, donc en temps $P(n)$. On considère la portion du cycle $z-x-y-t$ où l'arête xy est celle du cycle qui n'existe pas dans le chemin. On note $c(u)$ la couleur du sommet u à la fin de l'algorithme de coloration du chemin.

On effectue alors une communication supplémentaire, chaque sommet échangeant sa couleur et son identité avec ses voisins. Si un sommet u n'a pas de conflit avec ses

voisins, il s'arête et ne change plus sa couleur. On remarque que les seuls sommets voisins pouvant être en conflits sont x et y. Si $c(x) = c(y)$, alors le sommet de plus petite identité (disons que c'est x) applique FIRSTFREE(x) : il choisit une couleur parmi l'ensemble $\{0, 1, 2\} \setminus \{c(z), c(y)\}$ ($c(z)$ et $c(y)$ sont connues de x). Le seul conflit possible est ainsi éliminé. Donc $C(n) \leqslant P(n) + 1$. □

Donc, une borne inférieure sur $C(n)$ implique une borne inférieure sur celle de $P(n)$. Voici un résultat, du à Linial [Lin87, Lin92] à propos de $C(n)$.

Théorème 3 *Tout algorithme distribué de 4-coloration des cycles orientés à n sommets nécessite au moins $\frac{1}{2}(\log^* n) - 1$ rondes.*

Puisqu'une 3-coloration est aussi une 4-coloration, il suit que la borne inférieure est également valable pour tout algorithme de 3-coloration. Notons que la borne du théorème 3 est très proche de celle de la proposition 17 en prenant $m = n$. Le reste de ce paragraphe est consacrée à la preuve du théorème 3 qui se déroule en trois temps.

Exécution standard. Soit A un algorithme distribué qui dans le modèle LOCAL produit en temps t une coloration propre pour tout cycle orienté à n sommets. On note k le nombre maximum de couleurs produit par A.

On va supposer que l'exécution de A, pour chaque sommet u, se découpe en deux phases (on parle d'*exécution standard*) :

1. Une phase de communication de t rondes, où A collecte toutes les informations disponibles dans le graphe à distance $\leqslant t$ de u.
2. Une phase de calcul, sans plus aucune communication, où A doit produire le résultat final dans la mémoire locale de u.

Il n'est pas difficile de voir que tout algorithme distribué qui s'exécute en t rondes dans le modèle LOCAL possède une exécution standard, et peut en effet être découpé en une phase de communication de t rondes suivi d'une de calcul.

D'abord, un sommet u ne peut recevoir d'information d'un sommet v que s'il se situe à une distance $\leqslant t$. Ensuite, tous les sommets exécutent le même algorithme, éventuellement sur une vue locale différente à cause des identifiants ou du graphe qui n'est pas forcément symétrique. Donc si v est à une distance $d \leqslant t$ de u, alors les calculs de v pendant les $p = t - d$ premières rondes de A peuvent être simulées par u, une fois toutes les informations à distance $\leqslant t$ collectées. Notons encore une fois que les calculs de v au délà des p premières étapes ne peuvent pas être communiquées à u, et donc ne sont pas utilisés par A en u.

Bien sûr, dans une exécution standard, le résultat est le même, mais la taille des messages produits pendant l'exécution n'est pas forcément la même. Des calculs intermédiaires peuvent réduire la taille des messages, alors que la collecte des informations peut produire

des messages très grands. Par exemple, un sommet peut avoir à communiquer les identifiants de ses nombreux voisins à son père. Dans le modèle LOCAL la taille des messages n'a pas d'importance.

On résume parfois le modèle LOCAL comme ceci :

> « Ce qu'on peut calculer en temps t sur un graphe dans le modèle LOCAL est ce qu'on peut calculer sans communication si tous les sommets connaissent leur voisinage[2] à distance t ».

Dans une exécution standard sur des cycles orientés à n sommets, l'algorithme A en u fait donc deux choses :

1. Collecte, pendant la phase de communication, un vecteur appelé *vue* du sommet u. Il s'agit d'une suite de $2t+1$ identifiants $(x_1, \ldots, x_{t+1}, \ldots, x_{2t+1})$ où $x_{t+1} = \mathrm{ID}(u)$, $(x_1, \ldots, x_t)$ sont les identifiants des t voisins de gauche de u et $(x_{t+2}, \ldots, x_{2t+1})$ ceux des t voisins de droite. x_1 et x_{2t+1} sont les identifiants à distance t de u.
2. Produit, pendant la phase de calcul, une couleur pour le sommet u, notée $c_A(u)$. Cette couleur ne dépend donc que de la vue de u, et donc $c_A(u) = f_A(x_1, \ldots, x_{2t+1})$ où $(x_1, \ldots, x_{2t+1})$ est la vue de u.

Graphe de voisinage. On définit le graphe $N_{t,n}$, pour t, n entiers tels que $2t+1 \leqslant n$, comme suit :

- Les sommets sont $(x_1, \ldots, x_{2t+1})$ avec $x_i \in \{0, \ldots, n-1\}$ et $x_i \neq x_j$ pour $i \neq j$.
- Les arêtes sont $(x_1, \ldots, x_{2t+1}) - (x_2, \ldots, x_{2t+1}, x_{2t+2})$.

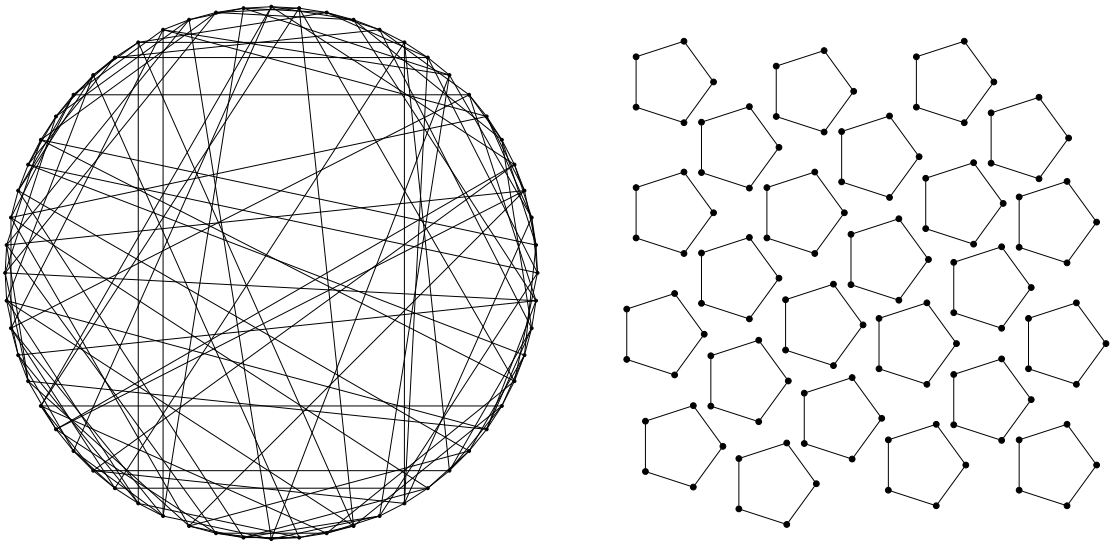
Ce graphe est aussi appellé le *graphe de voisinage* (*neighborhood graph* en Anglais). Informellement, les sommets de $N_{t,n}$ sont les vues possibles d'un sommet d'un cycle collectées après la phase de communication de t rondes. On met une arête dans entre deux vues, s'il existe un cycle où les deux vues peuvent apparaître simultanément sur deux sommets voisins de ce cycle.

Ce n'est pas utile pour la suite, mais remarquons que le graphe $N_{t,n}$ est un sous-graphe induit d'un graphe de De Bruijn de paramètre n et $2t+1$. Dans un graphe de De Bruijn les lettres x_i des sommets ne sont pas forcément différentes deux à deux. Il y a donc plus de sommets, mais la règle d'adjacence est la même.

La stratégie de la preuve est la suivante. Pour borner inférieurement le temps t de A, on va chercher à borner inférieurement le nombre chromatique du graphe $N_{t,n}$. Par construction, on a :

Proposition 19 $\chi(N_{t,n}) \leqslant k$.

2. C'est-à-dire tous les sommets et leurs entrées (généralement l'identifiants), et les l'arêtes de la boule de rayon t, exceptée les arêtes entre sommets à distance exactement t.



(a) (b)

FIGURE 7.4 – (a) Le graphe $N_{1,5}$ possède $60 = n \cdot (n-1) \cdots (n-2t+1)$ sommets de degré $6 = 2 \cdot (n-2t)$. Par exemple, dans ce graphe, les voisins du sommet 012 sont : 120, 123, 125 et par symétrie 201, 301, 401. On peut montrer que $\chi(N_{1,5}) = 4$ (cf. la couverture). En revanche, (b) le graphe $N_{2,5}$, qui a 120 sommets, vérifie clairement $\chi(N_{2,5}) = 3$.

Preuve. On va colorier les sommets de $N_{t,n}$ en utilisant l'algorithme A, et donc montrer que k couleurs suffisent pour $N_{t,n}$. La couleur d'un sommet $(x_1, \ldots, x_{t+1}, \ldots, x_{2t+1})$ de $N_{2t+1,n}$, est tout simplement $f_t(x_1, \ldots, x_{t+1}, \ldots, x_{2t+1}) \in \{0, \ldots, k-1\}$.

Il reste à montrer qu'il s'agit alors d'une coloration propre de $N_{t,n}$. Supposons qu'il existe deux sommets voisins de $N_{t,n}$ ayant la même couleur. Soient $v_1 = (x_1, \ldots, x_{t+1}, \ldots, x_{2t+1})$ et $v_2 = (x_2, \ldots, x_{t+2}, \ldots, x_{2t+2})$ deux sommets voisins de $N_{t,n}$ colorier avec A et de même couleur, donc avec $f_t(v_1) = f_t(v_2)$. On considère le cycle C orienté à n sommets (voir figure 7.6.3) où les $2t+2$ entiers des sommets v_1 et v_2 apparaissent dans l'ordre $(x_1, \ldots, x_{2t+2})$ comme identifiants successifs de sommets de C. Notons qu'il est possible que $x_1 = x_{2t+1}$, si $n = 2t+1$.

$$\cdots - x_1 - x_2 - \cdots - x_t - x_{t+1} - \cdots - x_{2t+1} - x_{2t+2} - \cdots$$

Le sommet d'identité x_{t+1} dans C, disons le sommet u_1, a pour vue le vecteur v_1. Le sommet d'identité x_{t+2} dans C, disons le sommet u_2, a pour vue v_2. Donc les couleurs de u_1 et u_2 dans le cycle C sont $c_A(u_1) = f_A(v_1)$ et $c_A(u_2) = f_A(v_2)$. Cela implique que $c_A(u_1) = c_A(u_2)$ ce qui contredit le fait que A calcule une coloration propre pour C. □

Comme le suggère la figure 7.4, et comme on va le voir ci-dessous, le nombre chromatique de $N_{t,n}$ décroît lorsque t augmente. Grâce à la proposition 19, on cherche donc la plus petite valeur de t qui satisfait $\chi(N_{t,n}) \leqslant k$.

Une façon de paraphraser la proposition 19 est de dire que s'il existe un algorithme A de k-coloration en temps t pour tout cycle orienté à n sommets, alors $\chi(N_{t,n}) \leqslant k$. La contraposée nous dit donc que : si $\chi(N_{t,n}) > k$, alors il n'est existe pas d'algorithme de k-coloration en temps t pour tous les cycles orientés à n sommets.

En fait, la réciproque de la proposition 19 est également vraie. À savoir, si l'on peut colorier $N_{t,n}$ avec k couleurs, alors il existe un algorithme distribué qui calcule en t rondes une k-coloration pour tout cycle orienté à n sommets. On pourrait donc se servir de cet argument pour concevoir des algorithmes distribués de coloration optimaux comme étudiés dans [Ryb11].

Nombre chromatique. Calculer le nombre chromatique de $N_{t,n}$ est un problème difficile en soit. Pour conclure notre preuve on a en fait besoin que d'une minoration. On définit le graphe orienté $B_{t,n}$ comme :

- Les sommets sont $\{x_1, \ldots, x_{2t+1}\}$ avec $0 \leqslant x_1 < \cdots < x_{2t+1} < n$.
- Les arcs sont $\{x_1, \ldots, x_{2t+1}\} \to \{x_2, \ldots, x_{2t+1}, x_{2t+2}\}$.

Le graphe $B_{t,n}$ est donc l'orientation d'un sous-graphe induit de $N_{t,n}$, on ne garde dans $N_{t,n}$ que les sommets $(x_1, \ldots, x_{2t+1})$ avec $x_1 < \cdots < x_{2t+1}$. Il y a beaucoup moins de sommets dans $B_{t,n}$ que dans $N_{t,n}$. En effet, $B_{t,n}$ a $\binom{n}{2t+1} \leqslant 2^{2t+1}$ sommets, le nombre de sous-ensembles $\{x_1, \ldots, x_{2t+1}\}$ de taille $2t+1$ de $\{0, \ldots, n-1\}$, alors que $N_{t,n}$ en possède de l'ordre de $n^{2t+1} \gg 2^{2t+1}$.

Comme on le voit sur la figure 7.5(a), le graphe $B_{1,5}$ est nettement plus petit que le graphe $N_{1,5}$ (5 sommets pour $B_{1,5}$ contre 60 pour $N_{1,5}$. Voir aussi la figure 7.4(a)). Bien évidemment, $\chi(B_{t,n}) \leqslant \chi(N_{t,n})$, l'orientation ne change en rien le nombre chromatique des graphes.

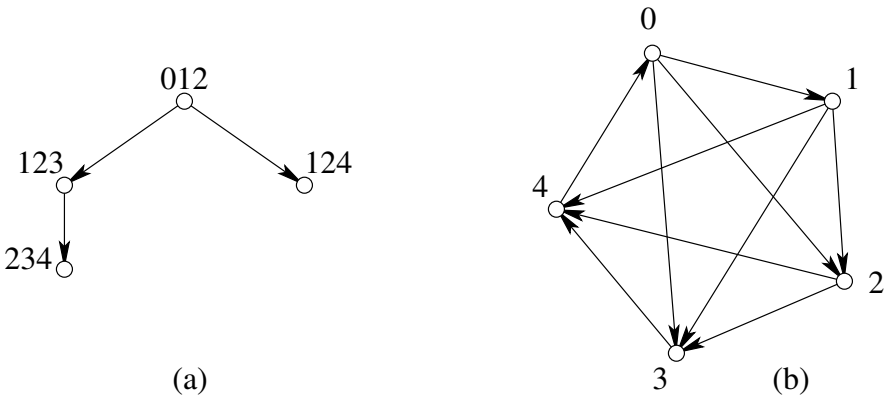


FIGURE 7.5 – (a) Le graphe $B_{1,5}$, et (b) le graphe $B_{0,5}$.

Avant de démontrer la prochaine proposition, nous avons besoin de la définition suivante. Le *line graph* d'un graphe orienté G, noté $\mathcal{L}(G)$, est le graphe orienté dont les sommets sont les arcs de G, et les arcs de $\mathcal{L}(G)$ sont les paires $((u,v),(v,w))$ où (u,v) et (v,w) sont des arcs de G (voir l'exemple de la figure 7.6). Dans la suite on notera plus simple uv un arc allant du sommet u au sommet v.

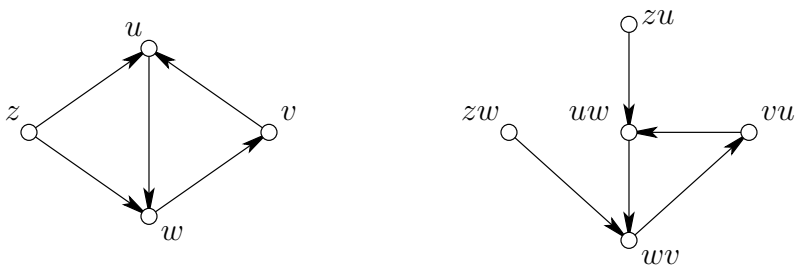


FIGURE 7.6 – Un graphe orienté G à 4 sommets (à gauche) et son *line graph* $\mathcal{L}(G)$ (à droite).

Proposition 20

i. $B_{0,n}$ *est une clique orientée à* n *sommets.*

ii. $B_{t,n} \simeq \mathcal{L}(\mathcal{L}(B_{t-1,n}))$ *(ils sont isomorphes).*

Preuve.

i. Les sommets de $B_{0,n}$ sont les n entiers de $\{0,\dots,n-1\}$, et on met un arc de x à y si $x < y$. Il s'agit donc d'une clique orentiée (voir la figure 7.5(a) pour un exemple avec $n = 5$).

ii. On remarque que les arcs de $B_{t-1,n}$, à savoir $\{x_1,\dots,x_{2t-1}\} \to \{x_2,\dots,x_{2t}\}$, sont en bijection avec les ensembles $\{x_1,\dots,x_{2t}\}$ qui sont les sommets de $\mathcal{L}(B_{t-1,n})$. Par exemple[3], on peut voir l'arc $013 \to 136$ comme le sommet 0136. Et le sommet 1278 comme un l'arc $127 \to 278$.

Ainsi, en appliquant trois fois cette bijection sur un chemin de trois arcs et quatre sommets de $B_{t-1,n}$ (voir ci-dessous), on obtient un chemin de deux arcs et trois sommets de $\mathcal{L}(B_{t-1,n})$. Et en réappliquant la bjection deux autres fois, on obtient un arcs et deux sommets de $\mathcal{L}(\mathcal{L}(B_{t-1,n}))$.

3. Cette correspondance ne marche pas pour $N_{t-1,n}$, car par exemple l'arête $4032-0324$ est peut être en bijection avec 40324, mais 40324 ne pourra jamais faire parti d'un sommet valide d'un $N_{t,n}$ (présence de deux fois la même valeur ! ce qui ne peut pas arriver pour si les x_i sont croissants avec i).

$$\{x_1, \dots, x_{2t-1}\} \to \{x_2, \dots, x_{2t}\} \to \{x_3, \dots, x_{2t+1}\} \to \{x_4, \dots, x_{2t+2}\}$$
$$\{x_1, \dots, x_{2t}\} \to \{x_2, \dots, x_{2t+1}\} \to \{x_3, \dots, x_{2t+2}\}$$
$$\{x_1, \dots, x_{2t+1}\} \to \{x_2, \dots, x_{2t+2}\}$$

Cette dernière bijection n'est ni plus ni moins que la définition des sommets et des arcs de $B_{t,n}$. Donc $B_{t,n}$ et $\mathcal{L}(\mathcal{L}(B_{t-1,n}))$ sont isomorphes. □

Proposition 21 *Pour tout graphe orienté G, $\chi(\mathcal{L}(G)) \geqslant \log \chi(G)$.*

La notation $\mathcal{L}(G)$ pour le *line graph* de G avait donc un nom prédestiné.

Preuve. On considère une k-coloration c optimale de $\mathcal{L}(G)$, c'est-à-dire avec $k = \chi(\mathcal{L}(G))$. On va montrer comment obtenir une 2^k-coloration propre de G. Cela suffit puisqu'alors $\chi(G) \leqslant 2^k = 2^{\chi(\mathcal{L}(G))}$ ce qui implique bien le résultat souhaité.

Notons que la k-coloration des sommets de $\mathcal{L}(G)$ revient à colorier les arcs uv de G. Pour tout sommet u de G, on pose $f(u) = \{c(vu) : vu \in E(G)\}$ c'est-à-dire l'ensemble des couleurs dans $\mathcal{L}(G)$ des arcs entrant de u. On va voir que f définit une coloration propre de G.

Montrons d'abord que $f(u)$ possède au plus 2^k valeurs distinctes. En effet, $f(u) \subseteq \{0, \dots, k-1\}$ est un sous-ensemble de couleurs de $\mathcal{L}(G)$. Et il existe au plus 2^k sous-ensemble distincts.

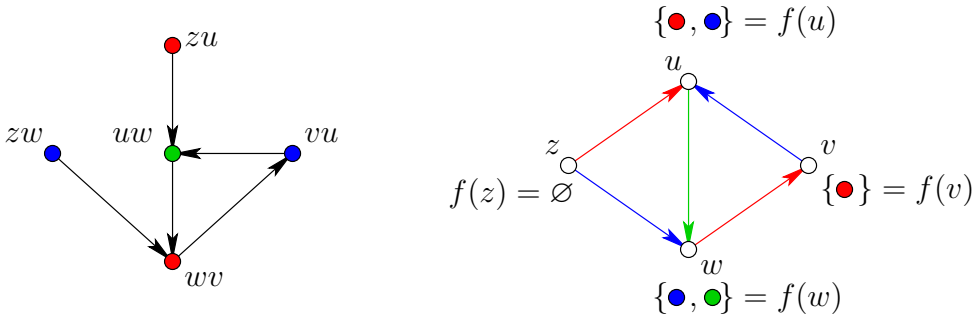


FIGURE 7.7 – Illustration de la coloration de G par celle de $\mathcal{L}(G)$.

Montrons enfin que f est une coloration propre de G. Soit vu un arc de G et $c = c(vu)$ la couleur de cet arc dans $\mathcal{L}(G)$. On a $c \in f(u)$. Supposons que $c \in f(v)$. Alors il existe au moins un arc wv entrant de v de couleur $c(wv) = c = c(vu)$. Or dans $\mathcal{L}(G)$ il existe un arc entre wv et vu : contradiction. Donc $c \notin f(v)$, et donc $f(u) \neq f(v)$. □

Proposition 22 $\chi(B_{t,n}) \geqslant \log^{(2t)} n$.

Preuve. D'après la proposition 20(i), $\chi(B_{0,n}) = n$. En conbinant plusieurs fois la proposition 20(ii) et la proposition 21, on obtient que :

$$\begin{aligned}
\chi(B_{t,n}) &= \chi(\mathcal{L}(\mathcal{L}(B_{t-1,n}))) \geqslant \log\chi(\mathcal{L}(B_{t-1,n})) \geqslant \log\log\chi(B_{t-1,n}) \\
&\geqslant \log^{(2)}\chi(B_{t-1,n}) \\
&\geqslant \log\log\left(\log^{(2)}\chi(B_{t-2,n})\right) = \log^{(4)}\chi(B_{t-2,n}) \\
&\cdots \\
&\geqslant \log^{(2t)}\chi(B_{0,n}) = \log^{(2t)} n \ .
\end{aligned}$$

□

En combinant la proposition 22 et la proposition 19, on conclut que

$$\log^{(2t)} n \ \leqslant \ \chi(B_{t,n}) \ \leqslant \ \chi(N_{t,n}) \ \leqslant \ k \ .$$

Pour $k = 4$, on déduit ainsi que le nombre t de rondes de A doit vérifier :

$$\log^{(2t)} n \ \leqslant \ 4 \ . \tag{7.2}$$

Montrons que l'inégalité (7.2) implique que $t \geqslant \frac{1}{2}(\log^* n) - 1$. Supposons que $t < \frac{1}{2}(\log^* n) - 1$, soit $2t + 2 < \log^* n$. Donc $\log^{(2t+2)} n > 1$ car par définition $\log^* n$ est le plus petit entier i tel que $\log^{(i)} n \leqslant 1$. (Une autre façon de voir est que la fonction $f(i) = \log^{(i)} n$ est décroissante en i.) C'est équivalent à $\log^{(2t)} n > 2^{2^1} = 4$: contradiction avec l'inégalité (7.2).

Donc le temps de l'algorithme de 4-coloration A est $t \geqslant \frac{1}{2}(\log^* n) - 1$. Cela termine la preuve du théorème 3.

7.7 Exercices

Donner le plus petite valeur de n pour que le temps t de la borne inférieure soit $t > 1$? Réponse : $n = 2^{2^{2^2}} + 1 = 65537$.

Bibliographie

[Als08] Brian Alspach. The wonderful Walecki construction. *Bulletin of the Institute of Combinatorics & Its Applications*, 52 :7–20, 2008.

[BE08] Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition. In 27^{th} *Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 25–34. ACM Press, August 2008.

[BE09] Leonid Barenboim and Michael Elkin. Distributed $(\delta+1)$-coloring in linear (in δ) time. In 41^{st} *Annual ACM Symposium on Theory of Computing (STOC)*, pages 111–120. ACM Press, May 2009.

[CV86] Richard Cole and Uzi Vishkin. Deterministic coin tossing and accelerating cascades : micro and macro techniques for designing parallel algorithms. In 18^{th} *Annual ACM Symposium on Theory of Computing (STOC)*, pages 206–219. ACM Press, 1986.

[KSV11] Amos Korman, Jean-Sébastien Sereni, and Laurent Viennot. Toward more localized local algorithms : removing assumptions concerning global knowledge. In 30^{rd} *Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 49–58. ACM Press, June 2011.

[Lin87] Nathan Linial. Distributive graph algorithms - Global solutions from local data. In 28^{th} *Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 331–335. IEEE Computer Society Press, October 1987.

[Lin92] Nathan Linial. Locality in distributed graphs algorithms. *SIAM Journal on Computing*, 21(1) :193–201, 1992.

[PS92] Alessandro Panconesi and Aravind Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In 24^{th} *Annual ACM Symposium on Theory of Computing (STOC)*, pages 581–592. ACM Press, 1992.

[Ryb11] Joel Rybicki. *Exact bounds for distributed graph colouring.* PhD thesis, University of Helsinki, Dept. of Computer Science, May 2011.

[Suo11] Jukka Suomela. Survey of local algorithms. *ACM Computing Survey*, 2011. To appear.

[SV93] Márió Szegedy and Sundar Vishwanathan. Locality based graph coloring. In 25^{th} *Annual ACM Symposium on Theory of Computing (STOC)*, pages 201–207. ACM Press, May 1993.

CHAPITRE 8

Ensembles indépendants maximaux

8.1 Introduction

Un *ensemble indépendant maximal* dans un graphe G (MIS en anglais, pour *Maximal Independent Set*) est un sous-ensemble de sommets M de G tel que : 1) si $x, y \in M$, alors x et y ne sont pas adjacents (M est donc un ensemble indépendant) ; et 2) si l'on ajoute un nouveau sommet à M, alors M n'est plus un ensemble indépendant (il est donc maximal pour l'inclusion).

[EXEMPLE]

Il faut noter que les deux conditions précédentes peuvent être vérifiées localement. Pour l'indépendance, il faut vérifier que tout sommet x dans M n'a aucun voisin dans M, et 2) pour la maximalité, si $x \notin M$, alors il doit avoir au moins un voisin dans M puisque sinon on pourrait rajouter x à M.

Remarques : 1) Un MIS est un ensemble dominant (c'est-à-dire tout $x \notin M$ a un voisin $y \in M$). 2) Dans une coloration (propre) les sommets d'une même couleur forme un ensemble indépêndant (mais pas forcément indépendant).

Coloration, ensemble dominant et MIS sont donc des notions proches et, comme nous allons le voir, il existe des moyens de calculer l'un à partir de l'autre.

L'utilisation des MIS en algorithmique distribuée est importante. On utilise les MIS pour créer du parallélisme. Il est souvent important que des ensembles de sommets puissent calculer indépendamment les uns des autres. Typiquement, pour le calcul d'arbre couvrant de poids minimum (MST), dans l'implémentation de Kruskal, il est assez intéressant que chaque sous arbre (initialement réduit à un sommet) puisse "pousser" indépendamment. Pour ce faire les racines de ces sommets initiateurs doivent être indépendantes pour ne pas se géner.

8.2 Algorithmes de base

En séquentiel ...

Note : calculer un MIS est donc très simple, de même que le calcul d'un ensemble dominant ou d'une coloration. Par contre, si l'on souhaite un MIS de cardinalité maximum (bien faire la différence entre maximal et maximum) le problème devient difficile (NP-complet pour la décision) et même reste difficile à approximer. C'est la même chose pour trouver un ensemble dominant de petite taille où une coloration avec peu de couleurs.

En distribué l'approche consiste à calculer une variable b représentant la décision de joindre ($b = 1$) ou de ne pas joindre ($b = 0$) l'ensemble MIS. Initialement $b = -1$.

Il y a deux façons de faire : MIS_{dfs} et MIS_{rank}. Dans le pire des cas, ils ont la même complexité, mais le second à tendance à être plus rapide dans la pratique. Cependant, MIS_{rank} pré-suppose des identité unique, alors que MIS_{dfs} non.

Dans un chemin, MIS_{rank} peut aller aussi lentement que MIS_{dfs}. Cependant, on peut montrer qu'avec grande probabilité, une permutation aléatoire dans un chemin fait que MIS_{rank} calculera un MIS en temps $O(\log n / \log\log n)$, au lieu de n dans le cas du MIS_{dfs}.

8.3 Ensemble dominants

Définition 7 *Un ensemble S est dominant pour un graphe G si pour tout sommet $x \in V(G)$, soit $x \in S$, ou x a un voisin $y \in S$.*

Rappelons qu'un MIS est un ensemble dominant, mais pas l'inverse !

Ici on s'intéresse aux ensemble dominants de petites cardinalités pour des motivations liés au routage (notion de sommets spéciaux et peu nombreux qui contrôlent tous les autres).

Prop : tout graphe connexe possède un ensemble dominant de taille $\leqslant n/2$, si $n > 1$.

Arbre couvrant + partition en deux couleurs suivant la parité de la distance à la racine.

On s'intéresse à de petits ensemble dominants (de taille $\leqslant n/2$) pour les arbres enracinés (on suppose donc que chacun connaît son père).

...

8.4 Exercices

On définit l'eccentricité d'un graphe G comme $\mathrm{ecc}(G) = \max_u \mathrm{ecc}(u)$. Constuire un graphe G_n et H_n à n sommets tels que $\mathrm{ecc}(G_n) = \mathrm{diam}(G_n)$ et $\mathrm{ecc}(H_n) = \frac{1}{2}\mathrm{diam}(H_n)$.

Soit $S_0 = \{v : \deg(v) \leqslant 12\}$. Montrer que si G est planaire ($m \leqslant 3n - 6$), alors $|S_0| > n/2$. En déduire un algorithme distribué pour calculer un MIS sur tout graphe planaire à n sommets en temps $O(\log n \cdot \log^* n)$.

Supposons qu'il existe un ensemble de sommets $S \subseteq V(G)$ tel que $\forall s \in S$, $\deg(s) \geqslant t$, alors $m = |E(G)| \geqslant t \cdot |S|/2$. Donc si $t \geqslant 4m/n$ alors $|S| \leqslant 2m/t \leqslant n/2$, ce qui est le cas pour $t = 12$.

CHAPITRE 9

Graphes couvrant éparses

Introduction au calcul de *spanner* dans le modèle LOCAL. Ce sont des sous-graphes couvrant les sommets d'un graphe possédant un nombre limité d'arêtes. De plus ils garantissent que les distances dans les sous-graphes ne sont pas trop éloignées des distances originales.

9.1 Introduction

L'utilisation de sous-graphe couvrant, c'est-à-dire qui contient tous les sommets du graphe d'origine, est banale en informatique. Typiquement un arbre couvrant un graphe connexe G à n sommets garantit la connexité alors qu'il ne possède que $n-1$ arêtes. Cela peut être particulièrement intéressant en calcul distribué de minimiser de nombre de liens sans pour autant déconnecter G.

On a vu dans les chapitres précédant que le nombre de messages nécessaire à résoudre certaines tâches peut être réduit si un arbre préexistant était connu des processeurs. Au chapitre 6, on a vu que la connaissance d'un sous-graphe couvrant peu dense pouvait être utilisé pour construire des synchroniseurs.

De manière générale, on souhaite construire un graphe couvrant peu dense préservant les distances d'origines. Bien sûr il n'est pas possible de supprimer des arêtes sans modifier au moins une distance. Néanmoins on souhaite limiter ces modifications. La définition suivante s'applique aussi bien au cas des graphes valués que non valués.

Définition 8 *Soient G un graphe et H un sous-graphe couvrant de G, c'est-à-dire avec $V(G) = V(H)$. L'*étirement *de H est le plus petit réel s tel que $d_H(u,v) \leqslant s \cdot d_G(u,v)$ pour tout sommets $u, v \in V(G)$.*

EXEMPLES

Dans la littérature on appelle aussi *s-spanner* tout sous-graphe couvrant d'étirement s (*stretch* en Anglais).

Dans la suite, on va essentiellement considérer que les graphes sont non valués.

Le problème qu'on se fixe est donc de calculer de manière distribué, dans le modèle LOCAL, des sous-graphes couvrant avec peu d'arêtes et si possible un étirement faible. On se base essentiellement sur l'article [DGPV08].

Formellement, calculer un sous-graphe H signifie que chaque sommet u doit sélectionner un ensemble d'arêtes qui lui sont incidentes, disons $H(u)$. Le graphe calculé est alors le graphe $\bigcup_{u \in V(G)} H(u)$ composé de l'union de tous ces choix.

Notons qu'il n'est pas formellement imposé que les deux sommets extrémités d'une arête s'accordent sur leur décision, c'est-à-dire $uv \in H(u)$ n'implique pas forcément $uv \in H(v)$. Cependant, avec une ronde supplémentaire, on peut toujours faire en sorte que tous les sommets se mettent d'accord (par exemple sur le fait de garder l'arête si l'un des deux l'a sélectionnée, ou le contraire, de ne pas la garder si l'un des deux ne l'a pas garder).

9.2 Calcul d'un 3-spanner

On va voir un algorithme permettant de calculer un sous-graphe couvrant d'étirement 3 avec significativement moins de n^2 arêtes, en fait $2n^{1.5}$. Il suffit de seulement deux rondes. La décision de sélectionner ou non les arêtes est donc locale.

Avant de présenter l'algorithme, remarquons que le problème n'a rien d'évident. Décider localement de supprimer une arête est *a priori* très risqué, puisque par exemple un sommet n'a aucun moyen de deviner s'il est dans un arbre ou pas (ils ne connaissant pas, par exemple le nombre d'arêtes du graphe). Bien sûr pour avoir un l'étirement borné, aucun sommet d'un arbre ne peut supprimer d'arête. D'un autre coté, si aucun sommet ne supprime d'arête on pourrait avoir de l'ordre de n^2 arêtes si le graphe G était très dense.

Jusqu'à récemment, aucun algorithme déterministe n'était connu pour ce problème. Ils existent des algorithmes en deux rondes, mais qui ne garantissent le nombre d'arêtes seulement en moyenne [BKMP05]. Le problème des algorithmes aléatoires de ce type (Monté Carlo) est qu'on est pas certain du résultat, contrairement aux algorithmes de type Las Vegas. Et contrairement au calcul séquentiel, en distribué il n'est pas toujours facile de vérifier la condition (comment vérifier que le nombre d'arêtes est correct sans une diffusion globale ?). En général, en distribué, on ne peut jamais recommencer !

Dans la suite, on rappelle que $B(u, 1)$ représente l'ensemble formé de u et des voisins de u, la boule centrée en u et de rayon 1. Évidemment, $|B(u, 1)| = \deg(u) + 1$.

Algorithme SPAN3
(code du sommet u)

1. NEWPULSE
2. Envoyer son identifiant et recevoir l'identifiant de tous ses voisins.
3. Choisir un ensemble R_u composé de son identifiant et d'un sous-ensemble arbitraire des identifiants reçus tel que $|R_u| = \min\{|B(u, 1)|, \lceil \sqrt{n} \rceil\}$.

4. NewPulse
5. Envoyer R_u et recevoir la sélection R_v de chaque voisin v.
6. Poser $H(u) := \{uv : v \in R_u, v \neq u\}$ et $W := B(u,1) \setminus \{v : u \in R_v\}$.
7. Tant qu'il existe $w \in W$:
   (a) $H(u) := \{uw\} \cup H(u)$.
   (b) $W := W \setminus \{v \in W : R_v \cap R_w \neq \varnothing\}$.

Il y a une variante où u ne connaît pas le nombre $\lceil \sqrt{n} \rceil$, mais qui nécessite une ronde de plus. L'algorithme peut aussi être adapté pour fonctionner avec des poids arbitraires sur les arêtes : il faut choisir R_u parmi ses plus proches voisins et le sommet w parmi les plus proches restant de W.

Théorème 4 *Pour tout graphe G à n sommets, l'algorithme* Span3 *calcule en deux rondes un sous-graphe couvrant de G d'étirement $\leqslant 3$ et avec moins de $2n^{3/2}$ arêtes.*

Preuve. L'algorithme comprends deux rondes de communications.

Étirement. Soit uv une arête de G qui n'est pas dans le sous-graphe couvrant $H = \bigcup_u H(u)$ (figure 9.1(a)). Clairement toutes les arêtes de la forme uw avec $w \neq u$, $u \in R_u \cup R_w$ et $w \in B(u,1)$ existent dans H. Donc si cette arête a été supprimée c'est que v a été supprimé de W à l'étape 7b de l'algorithme (figure 9.1(b)). Si v est supprimé de W à l'étape 7b, c'est que R_v et R_w s'intersectent (figure 9.1(c)). Or dans H toutes les arêtes de v vers R_v et de w vers R_w existent. Donc il existe dans H un chemin de u à v de longueur trois.

Il suit qu'un plus court chemin de x à y dans H est de longueur au plus trois celle d'un plus court chemin P de G, chaque arête de P qui n'est pas dans H étant remplacée par un chemin de H d'au plus trois arêtes.

Nombre d'arêtes. Montrons que tout sommet w de l'ensemble W calculé à l'étape 6 est tel que $|B(w,1)| > \lceil \sqrt{n} \rceil$. En effet, si $|B(w,1)| \leqslant \lceil \sqrt{n} \rceil$, alors $|R_w| = \min\{|B(w,1)|, \lceil \sqrt{n} \rceil\} = |B(w,1)|$. Donc R_w contiendrait tous les voisins de w, en particulier u. Or ce n'est pas le cas, donc $|B(w,1)| > \lceil \sqrt{n} \rceil$. Il suit que $|R_w| = \lceil \sqrt{n} \rceil$.

Les sommets sélectionnés dans la boucle « Tant que » ont, par construction, des ensembles deux à deux disjoints, c'est-à-dire $R_w \cap R_{w'} = \varnothing$ (figure 9.1(e-d)) puisque ceux qui intersectent R_w sont supprimés à jamais de W. Il suit que le nombre de sommets sélectionnés lors de la boucle « Tant que » est au plus $|B(u,2)|/\lceil \sqrt{n} \rceil \leqslant n/\sqrt{n} = \sqrt{n}$ chaque ensemble R_w étant inclus dans $B(u,2)$ qui est de taille au plus n.

Le nombre d'arêtes sélectionnées par u est donc au plus $|R_u| - 1 + \sqrt{n} \leqslant \lceil \sqrt{n} \rceil - 1 + \sqrt{n} < 2\sqrt{n}$. Au total H n'a pas plus de $2n^{3/2}$ arêtes. □

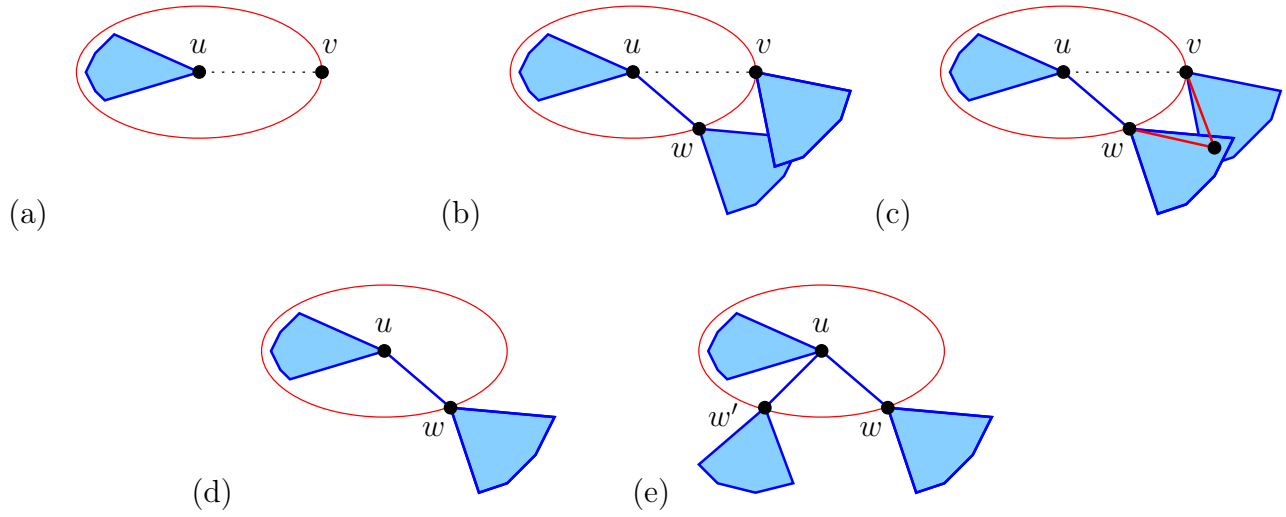


FIGURE 9.1 – Illustration de la preuve du théorème 4.

Exercice : étendre l'algorithme à un algorithme qui ne nécessite pas la connaissance de n. Combien de rondes cela nécessite ? Solution : il faut choisir une sélection de $\left\lceil \sqrt{|B(u,3)|} \right\rceil$ voisins au lieu de $\lceil \sqrt{n} \rceil$. Pour cela il faut au moins 3 rondes.

Problème ouvert : Peut-on réaliser la tâche précédante avec seulement deux rondes ?

Bibliographie

[BKMP05] Surender Baswana, Telikepalli Kavitha, Kurt Mehlhorn, and Seth Pettie. New constructions of (α, β)-spanners and purely additive spanners. In 16^{th} *Symposium on Discrete Algorithms (SODA)*, pages 672–681. ACM-SIAM, January 2005.

[DGPV08] Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. On the locality of distributed sparse spanner construction. In 27^{th} *Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 273–282. ACM Press, August 2008.

0 CHAPITRE

10 Routage et pair-à-pair

10.1 ...

10.2 Exercices

...

1 CHAPITRE 11 Solutions

Solution de l'exercice 4.4.4

```
main(r0,M){
// Diffusion avec écho d'un message M depuis un sommet r0

message X,A;
arête a,père;
int f=0;
booléen terminé=false;
A.data="ack";

if(ID==r0){ // si la source ...

  SENDALL(M);
  FORALL(a){ // on compte les fils
    X=RECEIVEFROM(a);
    if(X.data=="ack") f++;
  }

  while(f>0){ // on attend que les f fils aient terminés
    X=RECEIVE();
    if(X.data=="done") f-;
  }
}
else{ // si pas la source ...

  X=RECEIVE();
  père=X.from; // le premier message détermine le père
  FORALL(a) // on envoie ack ou on diffuse M
    if(a==père) SEND(a,A); else SEND(a,M);
```

```
  FORALL(a) // on compte les ack, ie les fils
    if(a<>père){ // ne pas faire de RECEIVE de son père sinon on est bloqué
      X=RECEIVEFROM(a);
      if (X.data=="ack") f++;
    }

  // si feuille (alors f==0). Le sous-arbre est donc terminé
  while(f>0){ // sinon on attend les f DONE des fils
    X=RECEIVE();
    if (X.data=="done") f-;
  }
  A.data="done";// le sous-arbre est terminé
  SEND(père,A); // on envoie "done" à son père

  A=RECEIVEFROM(père); // on recoit le signale de fin de r0
}

terminé=true; // le sommet courant a fini, il le diffuse aux autres
SENDALL(A); // on peux aussi diffuser seulement sur les fils
// fin du main()
```

Solution de l'exercice 5.4.1

```
main(r0){
// Parcours DFS depuis r0 avec optimisation du temps en O(n)

booléen V[MAX],départ; // V=tableau indiquant les voisins visités
arête a,j,père;
message X,J,M,A;

J.data="jeton";  // passage de jeton
M.data="visité"; // sommet visité
A.data="ack";    // accusé de réception

père=-1;j=0; // j=sommet qui doit recevoir le jeton
FORALL(a) V[a]=false; // aucun voisin n'est visité
départ=true; // au départ

while(j<>père) // le sommet courrant a fini quand le jeton repasse au père
{ if((ID==r0)&&(départ)){// correspond au départ en r0:
    X.data="jeton"; // dans ce cas, on fait comme si l'on recoit "jeton"
    X.from=-1; } // le père de r0 restera à -1
```

```
  else X=RECEIVE(); // sinon on attend la réception d'un message

  if(X.data=="visité"){
    V[X.from]=true; // l'émetteur vient d'être visité
    SEND(X.from,A); } // on accuse le message

  if(X.data=="jeton"){
    if(départ) {// dans tous les cas, c'est la première fois
      départ=false; // sommet courant visité, ce ne sera plus la 1ère fois
      père=X.from; // on choisit son père
      FORALL(a) if(a<>père) SEND(a,M); // on envoie les "visités"
      FORALL(a) if(a<>père) RECEIVEFROM(a);//on ne peut que recevoir des "ack"
      }
    j=père; // calcul le prochain sommet j devant recevoir le jeton
    FORALL(a) if((a<>père)&&(!V[a])) j=a;
    if(j>=0){// si j<0 alors ID==r0 et plus de sommet à visiter
      V[j]=true;
      SEND(j,J);
      }
    }
}}
```

CHAPITRE 12 Travaux pratiques

TP1 - 31 janvier 2007 - Diffusion

1. Installer ViSiDiA : http://www.labri.fr/projet/visidia/down.html

2. Programmer / Compiler et Visualiser le programme ci-dessous (envoie d'un message à ses voisins). Attention, votre programme source doit s'appeler TP1.java, le nom de la nouvelle classe.

```
package visidia.algo;
import visidia.simulation.*;
import visidia.misc.*;
import java.util.*;

public class TP1 extends Algorithm{

  public void init(){

     int deg = getArity();
     int id = getId();

     for(int i = 0; i < deg; i++){
       sendTo(i, new IntegerMessage(id));}

     for(int i = 0; i < deg; i++){
       IntegerMessage mesg = (IntegerMessage)receiveFrom(i);}
  }

  public Object clone() {return new TP1();}
}
```

3. Programmer l'algorithme de diffusion avec détection locale de la terminaison, celui vu en TD.

4. Implémenter et tester sur l'exemple du 2. une fonction `AsendTo()` permettant un envoie de message après un délais aléatoire simulant l'asynchronisme des messages.

TP2 - février 2007 - Diffusion avec détection de la terminaison

....

TP3 & TP4 - 2 mars 2007 - PRIM

En ViSiDiA, on ne peut pas (encore) associer un poids à une arête. Pour cela on va "calculer" un poids unique à partir des identifiants des sommets (qui eux sont uniques). Pour cela, on décide de coder chaque paire $\{i, j\}$ d'entiers naturels (une paire d'identifiants), par l'entier $\omega(i, j) = (i + j)(i + j + 1)/2 + \max\{i, j\}$. On peut vérifier facilement que $\omega(i, j) = \omega(j, i)$ et que si $\{i, j\} \neq \{i', j'\}$, alors $\omega(i, j) \neq \omega('i, j')$.

Pour tout sommet u, on note $\mathrm{ID}(u)$ son unique identifiant (que l'on peut obtenir avec la fonction `getId()`, et $u[i]$ le voisin de u connecté par le port $i = 0...\deg(u)$.

1) Écrire une fonction `CalculeTab(W)` qui, à l'aide d'un cycle de communication entre voisins, remplit un tableau d'entiers `W` $= W_u$ tel que $W_u[i] = \omega(\mathrm{ID}(u), \mathrm{ID}(u[i]))$.

On observe que $W_u[i]$ est unique pour tout i et tout u, et que si $\{u, v\}$ est une arête alors $W_u[i] = W_v[j]$ où $v = u[i]$ et $u = v[j]$. On utilisera cette valeur commune comme *poids* de l'arête $\{u, v\}$.

2) Implémenter l'algorithme PrimDist vu en cours où le poids des arêtes sont données par la fonction `CalculeTab(W)` appelée en chaque sommet.

Principe de PrimDist (rappel) :

- un sommet source (le sommet 0 par exemple) émet un message PUSLE aux sommets de l'arbre (initialement réduit à la source) avec le nouveau sommet à ajouter.
- le nouveau sommet est ajouter à l'arbre courant, puis il envoie un message de mise à jour à ses voisins de sorte qu'il puisse déterminer au retour de ses messages ses voisins déjà dans l'arbre.
- les feuilles calculent l'arête sortante de l'arbre de poids minimum, et diffuse l'information (poids et l'arête en question) à leur père. Les sommets internes font de même jusqu'à la racine.
- la racine calcule le minimum et recommence le cycle avec le PULSE suivant.

Pour l'implémentation, vous pourrez utiliser deux tableaux de booléens `T` et `F` indiquant respectivement si le port i mène à un sommet de l'arbre ou a un fils. Identifiez les types de messages et leur contenu : PULSE (=descente vers l'arête minimum), MIN (=remontée du minimum vers la racine), ADD (=ajoût du nouveau sommet), UPDATE (=mise à jour des voisins du sommet ajouté). Il peut être utile d'avoir une variable `min` donnant l'arête de poids minimum sortante du sommet courant. Ainsi, lorsque le sommet collecte les poids des arêtes sortantes de ses sous-arbres, il mettra `min=-1` si le min provient d'un de ses fils, où bien mettre `min=z` si `z` est l'indentité du sommet qui va potentiellement être choisit comme minimum par la racine. Ainsi, lorsque qu'un sommet de l'arbre recoie le message PULSE avec l'identité du sommet à ajouter, il peut mettre à jour son tableau `T` grâce à `min` et commencer la phase de calcul du minimum sans risque d'erreur.

3) Modifier le programme précédent pour détecter la détection globale de la terminaison, c'est-à-dire une variable `terminée` doit passer à vraie une fois que le MST a été entièrement calculé.

TP4 - 16 mars 2007 - Coloration avec FirstFree

On souhaite implémenter un algorithme de coloration basé sur FirstFree, à savoir on détermine la couleur du sommet courant en choisissant la plus petite couleur qui n'est pas utilisée par ses voisins (cela nécessite donc des échanges de messages).

On supposera que les couleurs sont initialisées à l'identité des sommets avec `getId()`. Pour éviter les conflits, vous utiliserez un parcours des sommets du graphe avec le principe du jeton : seul le sommet ayant le jeton applique FirstFree. Implémenter un tel algorithme de coloration en vous inspirant du programme C-distribué ci-dessous implémentant le parcours DFS vu en cours.

```
main(r0){
// Parcours DFS depuis r0 avec optimisation du temps en O(n)

booléen V[MAX],départ; // V=tableau indiquant les voisins visités
arête a,j,père;
message X,J,M,A;

J.data="jeton";  // passage de jeton
M.data="visité"; // sommet visité
A.data="ack";    // accusé de réception

père=-1;j=0; // j=sommet qui doit recevoir le jeton
FORALL(a) V[a]=false; // aucun voisin n'est visité
départ=true; // au départ

while(j<>père) // le sommet courrant a fini quand le jeton repasse au père
{ if((ID==r0)&&(départ)){// correspond au départ en r0:
    X.data="jeton"; // dans ce cas, on fait comme si l'on recoit "jeton"
    X.from=-1; } // le père de r0 restera à -1
  else X=RECEIVE(); // sinon on attend la réception d'un message

  if(X.data=="visité"){
    V[X.from]=true; // l'émetteur vient d'être visité
    SEND(X.from,A); } // on accuse le message

  if(X.data=="jeton"){
    if(départ) {// dans tous les cas, c'est la première fois
      départ=false; // sommet courant visité, ce ne sera plus la 1ère fois
      père=X.from; // on choisit son père
      FORALL(a) if(a<>père) SEND(a,M); // on envoie les "visités"
      FORALL(a) if(a<>père) RECEIVEFROM(a);//on ne peut que recevoir des "ack"
      }
    j=père; // calcul le prochain sommet j devant recevoir le jeton
    FORALL(a) if((a<>père)&&(!V[a])) j=a;
```

```
    if(j>=0){// si j<0 alors ID==r0 et plus de sommet à visiter
      V[j]=true;
      SEND(j,J);
      }
    }
}}
```

TP5 - 23 mars 2007 - Coloration en log*n

On souhaite implémenter un algorithme de coloration pour les arbres basé sur celui vu en cours s'éxecutant en temps $\log^* n$. Nous allons ici implémenter une version légèrement différente permettant l'exécution en mode asynchrone (voir le code en C-distribué ci-dessous).

Pour cela on suppose donc que le graphe est un arbre (graphe connexe sans cycle) et qu'un sommet est distingué, disons le sommet `r0`, qui sera la racine.

On considère les fonctions suivantes qui seront à programmer :

- `BIN(x)` qui à tout entier `x`$\in \mathbb{N}$ retourne une chaîne binaire de caractères contenant l'écriture binaire de `x`. (Essayez la fonction `toBinaryString()` de la classe `Integer`.)
- `LOG(x)` qui à tout entier `x`$\in \mathbb{N}^*$ retourne $\lceil \log_2 \texttt{x} \rceil$. Notez que cette fonction vaut également la longueur de la chaîne `BIN(x-1)`.
- `NewColor(A,B)` qui à toute paire de chaîne binaire (c'est-à-dire composée des caractères "0" ou "1") retourne la chaîne binaire `BIN(i)+b` où `i` est une position de bit où les chaînes `A` et `B` diffèrent et `b` le caractère `A[i]`. Attention, si les chaînes `A` et `B` sont de longueur différente il faudra, avant de calculer `i` et `b`, compléter la plus petite par des "0" à gauche. Par exemple, `NewColor("1000101","101")="1101"`.
- Pour calculer le nombre de sommets `n` du graphe vous pourrez utiliser (à titre exceptionnel) utiliser l'instruction `getNetSize()`. En fait, il est aussi possible de le calculer ViSiDiA grâce à l'astuce suivante : on demande à chaque processeurs de mettre à jour une variable avec son `ID` de façon à récupérer la valeur maximum dans le graphe. Pour cela il faut utiliser un sémaphore (ici un object quelconque) pour que l'écriture soit exclusive, grâce à l'instruction `synchronized(...){...}`.

  ```
  static Object lock=new Object;
  static n=0;

  public void init(){
  ...
  synchronized(lock) { if(getId()+1>n) n=getId()+1; }
  ...
  }
  ```

  Cependant le problème avec cette méthode est que certains sommets peuvent écrire leur identité et commencer à utiliser `n` alors que le sommet de plus grande identité n'a pas encore écrit la siennne, donnant alors une valeur différente de `n` suivant le sommet. Pour y remédier, il est nécessaire d'utiliser une temporisation assez longue avec `Thread.sleep(t)` où `t` est un temps en miliseconde.

```
main(r0,n){
// Coloration en 6 couleurs d'un arbre à n sommets de racine r0

arête a,père;
message M;
String color;
```

```
int L;

// Calcule un père pour tous
if(ID==r0){
  SENDALL(M);
  père=-1;
  }
else{
  M=RECEIVE(); // on recoie un message de son futur père
  père=M.from; // on stocke son père
  FORALL(a) if(a<>père} SEND(a,M); // on diffuse aux autres
  }

// On calcule 6 couleurs
color=BIN(ID); // couleur intiale du sommet courant
if(père<0) color="0"+color[length(color)-1]; // r0 de couleur "00" ou "01"
L=LOG(n); // L=longueur maximum des couleurs, L<=3 est possible

do{ // il faut faire au moins 1 étape sinon ne marche pas pour n=7 ou 8
  FORALL(a) if(a<>père) SEND(a,color); // on envoie sa couleur à ses fils
  if(père<0){ // la racine ne fait plus rien
    M=RECEIVEFROM(père); // on ne peut recevoir un message que de son père
    color=NewColor(color,M.data); // on calcule la nouvelle couleur
    }
  L=LOG(L)+1; // on recalcule la longueur maximum des couleurs
} while(L>3); // si L<=3 on a une 6 coloration
}
```

1) Implémenter les fonctions `BIN`, `LOG`, et `NewColor`, ainsi que l'algorithme précédent.

2) Cet algorithme donne au plus 6 couleurs. Étendez l'algorithme pour obtenir une 3-coloration.

TP6 - 30 mars 2007 - Ensemble dominant

On dit qu'un ensemble de sommets S d'un graphe G est un ensemble *dominant* si tout sommet de G est soit dans S, soit possède un voisin dans S.

Pour calculer un ensemble dominant en séquentiel on peut utiliser l'algorithme glouton suivant S (on note $N(v)$ l'ensemble des voisins de v dans G) :

1. $U := V(G)$ et $S := \varnothing$
2. tant que $U \neq \varnothing$ faire :
   (a) choisir un sommet $v \in U$
   (b) $U := U \setminus N(v)$
   (c) $S := S \cup \{v\}$

Écrire en programme asynchrone pour calculer un ensemble dominant basé sur la méthode précédente. Vous utiliserez un variable locale booléenne `S` qui pour chaque sommet indiquera son appartenance à l'ensemble dominant. Vous pourrez aussi colorier le sommet suivant les changements de valeur de sa variable `S` (instruction `putProperty("label",c)` avec `c="A"` ou `c="B"`).

Pour l'implémentation distribuée vous devrez bien réfléchir aux messages à envoyer pour être certain qu'un sommet peut effectivement se déclarer dans l'ensemble dominant. Pour se faire vous utiliserez l'idée qu'un sommet se rajoute à l'ensemble dominant que si tous ses voisins d'identité supérieure ne le sont pas, c'est-à-dire que leur variable `S` est à `false`. Pensez éventuellement à écrire d'abord l'algorithme en C-distribué avant de l'implémenter.

TPx - 2006

On a vu en cours un algorithme synchrone 6-COLOR(T) donnant une 6-coloration pour un arbre T.

1) Sur le même principe, écrire un algorithme synchrone COLOR(G) donnant une coloration d'un graphe G quelconque, avec $2^{O(\Delta)}$ couleurs et en temps $O(\log^* n)$, où Δ est une borne sur le degré maximum. (Rappel : pour cela il faut calculer la nouvelle couleur de v en calculant la position des différences entre $c(v)$ et la couleur de son j-ème voicin, $c(v_j)$. Pour les couleurs, pensez à utiliser `putProperty()`. Le degré d'un sommet s'obtient par `getArity()`.

2) Programmer (en synchrone) la procédure REDUCE(m) permettant de réduire le nombre de couleurs de m à $\Delta + 1$.

NB : pour avoir le degré maximum du graphe on demander à chaque processeurs de mettre à jour un varibale avec le max de son degré. Pour cela il utiliser un sémaphore (ici un object quelconque) pour que l'écriture soit exclusive (instruction `synchronized(...)...`).

```
static Object lock=new Object;
static degmax=0;

public void init(){
...
synchronized(lock){if(getArity()>degmax) degmax=getArity();}
...
}
```

TPx - 2006

Un *ensemble indépendant maximal* dans un graph G (MIS en anglais, pour *Maximal Independent Set*) est un sous-ensemble de sommets M de G tel que : 1) si $x, y \in M$, alors x et y ne sont pas adjacents (M est donc un ensemble indépendant) ; et 2) si l'on ajoute un nouveau sommet à M, alors M n'est plus un ensemble indépendant.

Pour construire un MIS, on peut appliquer l'algorithme glouton (séquentiel) suivant :

1. $U := V(G)$; $M := \varnothing$;
2. tant que $U \neq \varnothing$ faire :
2.1. choisir un sommet $v \in U$
2.2. $U := U \setminus N(v)$ (on enlève v et ses voisins)
2.3. $M := M \cup \{v\}$.

Proposer un algorithme distribué synchrone de cet algorithme séquentiel. Vous pourrez colorier en deux couleurs les sommets : ceux qui sont dans le MIS et les autres.

CHAPITRE 13 Projets

Le projet est à rendre par courriel (gavoille@labri.fr) au plus tard ***lundi 16 avril*** *2007 minuit. Vous devez m'envoyer un fichier .java pour l'exercice 2, source que je testerai sous ViSiDiA en mode asynchrone, ainsi qu'un rapport dans un format lisible non compressé (.ps, .pdf, ou .txt – pas de .doc, .sdw ou de .zip). Si vous pensez que cet utile, vous pouvez également m'envoyer un fichier de graphe d'exemple sur lequel votre algorithme fonctionne.*

Le rapport devra contenir au plus ***3 pages*** *(sans la page de garde). Il décrira le plus clairement possible la méthode ou algorithme utilisé, les remarques et vos commentaires. Votre programme devra être suffisament commenté.*

Pour les calculs de complexité (donnés avec la notation $O()$), vous veillerez à ce que les algorithmes et les complexités donnés dans votre rapport correspondent à ceux de vos programmes. Il n'y a pas une, mais plusieurs réponses possibles. La notation tiendra compte de la correspondance entre le rapport et votre programme, ainsi que de la qualité de l'argumentation.

Exercice 1

Dans le cours nous avons vu deux algorithmes asynchrones pour la construction d'arbre en largeur d'abord (BFS). L'un est basé sur Dijkstra, l'autre sur Bellman-Ford dont la complexité en temps est proportionelle au diamètre du graphe.

a) Écrire en C-distribué l'algorithme permettant de construire un arbre en largeur d'abord de racine r_0 selon Bellman-Ford avec la détection locale de la terminaison globale. Plus précisément, chaque sommet u doit gérer les trois variables suivantes :

`L :` doit, à la fin de l'algorithme, contenir la distance de u à r_0, c'est-à-dire le nombre minimum d'arêtes d'un chemin connectant u à r_0 dans le graphe.

`père :` doit contenir l'arête incidente à u menant au père dans l'arbre BFS, et `-1` pour la racine.

`terminé :` booléen initialisé à `false` qui doit passer à `true` seulement après que la variable `L` soit correcte pour tous les sommets du graphe, c'est-à-dire contienne la distance du sommet à la racine.

b) Quels sont les complexités en temps et en nombre de messages de votre algorithme.

Exercice 2

Soit G un graphe connexe à n sommets et S un sous-ensemble de sommets de G. On dit que S est un ensemble *dominant* si tout sommet de G est soit dans S, soit possède un voisin dans S.

On souhaite construire un algorithme asynchrone qui permette de construire rapidement pour G un ensemble dominant S de « petite » taille, typiquement avec au plus $n/2$ sommets.

L'ensemble S est représenté de manière distribuée par une variable locale `S`. On notera $\texttt{S}_u$ le booléen qui doit valoir `true` si et seulement si, à la fin de l'algorithme, $u \in S$. On souhaite donc calculer $\texttt{S}_u$ pour tout sommet u de G.

L'idée de l'algorithme est la suivante :

A. on construit une forêt couvrante G comprenant beaucoup de composantes, c'est-à-dire d'arbres que l'on supposera enracinés.

B. puis en parallèle sur chaque arbre T de la forêt, calculer l'ensemble P des sommets de T à distance paire (dans T) de r, et l'ensemble I des sommets de T à distance impaire (dans T) de r, où r représente la racine de T. On note M le plus petit des deux ensembles P et I. On pose finalement $\texttt{S}_u$ =`true` si et seulement si $u \in M$.

Pour calculer la forêt de la partie **A**, chaque sommet choisit comme père son voisin ayant la plus petite identité, avec la convention que si deux voisins se choissent mutuellement, alors celui de plus petite identité devient la racine.

Pour la partie **B**, et à chaque sommet u de l'arbre T, on associe un compteur $C(u) = (x, y)$ composé de deux valeurs : x étant le nombre de descendants de u à distance paire de u (u compris) et y le nombre de descendants de u à distance impaire de u. Donc pour toutes les feuilles u de T on a $C(u) = (1, 0)$. On remarque deux choses : 1) le compteur de la racine $C(r)$ peut être calculé à partir des compteurs des feuilles en remontant vers la racine r ; 2) Si $C(r)$ est connu de u, alors $\texttt{S}_u$ peut calculer correctement.

a) Fournir un programme ViSiDiA implémentant l'algorithme précédent calculant un petit ensemble dominant.

b) Estimer les complexités en temps et en nombre de messages de votre algorithme. Préciser la taille de vos messages. Validez ces complexités par des expériences. Discutez éventuellement des résultats obtenus.

c) Démontrer que l'ensemble S ainsi construit est de taille $|S| \leqslant n/2$.

Le projet est à rendre par courriel (gavoille@labri.fr) au plus tard ***jeudi 27 avril*** *2006 minuit. Vous devez m'envoyer vos fichiers sources java (1 par exercice) que je testerai sous ViSiDiA (en synchrone), ainsi qu'un rapport sous forme* ***lisible*** *(.ps, .pdf, ou .txt). Si vous pensez que cet util, vous pouvez également m'envoyer un fichier de graphe d'exemple sur lequel votre algorithme fonctionne.*

Le rapport : il devra contenir au plus ***3 pages****. Il décrira le plus clairement possible les algorithmes utilisés et vous donnerez pour chacun d'eux la complexité MESSAGE et TEMPS, avec la notation $O()$. Vous veillerez à ce que les algorithmes et les complexités donnés dans votre rapport correspondent à ceux de vos programmes. Il n'y a pas une, mais plusieurs réponses possibles. La notation tiendra compte de la correspondance entre le rapport et votre programme.*

Excercice 1 : taux de croissance

Soit G un graphe. Pour tout sommet u de G et tout entier $r \geqslant 0$, on note $B(u,r)$ l'ensemble des sommets à distance au plus r de u, c'est-à-dire la boule de rayon r centrée en u. Notez que $B(u,0) = \{u\}$. Le *taux de croissance* du sommet u est le plus petit entier c tel que pour tout $r \geqslant 0$, $|B(u,r)|/|B(u,r/2)| \leqslant c$. (Rappel : si B est un ensemble, $|B|$ est sa cardinalité)

Écrire un programme en ViSiDiA en mode synchrone qui calcul le taux des sommets d'un graphe. À la fin de l'exécution, chaque sommet devra contenir dans son étiquette "label" son taux de croissance.

Excercice 2 : dimension doublante

Soit G un graphe. La *dimension doublante* de G est le plus petit entier k tel pour chaque sommet u de G et chaque entier $r \geqslant 0$, il existe un ensemble S d'au plus k sommets de G tels que pour tout sommet v de $B(u,r)$ il existe un sommet $w \in S$ tel que $d(v,w) \leqslant r/2$. (On rappelle que $dv, w)$ est la distance dans le graphe entre v et w). Dit autrement, la boule centrée en u de rayon r est couverte par (ou contenue dans) au plus k boules de rayon deux fois moins.

Calculer la dimension doublante est NP-complet. On propose alors d'approximer la dimension doublante par la "méthode des r-nets". Un r-net S est un sous-ensemble de sommets de G maximal pour la propriété suivante : deux sommets distincts de S sont à une distance $> r$ (c'est-à-dire que si l'on ajoute un seul sommet à S alors il perd cette

propriété). Un ensemble indépendant maximal (MIS) est exactement un 1-net. Le calcul d'un r-net peut se faire par un algorithme glouton (séquentiel) suivant :

```
U := V(G); M := 0
Tant que U<>0 faire:
1. choisir un sommet u dans U
2. M := M+{u}
3. U := U-B(u,r)
Retourner M
```

Pour approximer la dimension doublante, on calcule simplement un $r/2$-net S, pour tous les $r \geqslant 0$, puis on évalue pour chaque sommet u la quantité $|B(u,r) \cap S|$, et on prend la valeur maximum atteinte pour un sommet. Donc l'algorithme séquentiel d'approximation de la dimension doublante est le suivant :

```
k := 0
Pour tout r de 0 à n faire:
1. Calculer un r/2-net S
2. Pour tout u de V(G) faire: k := max{k,|B(u,r) INTER S|}
Retourner k
```

Écrire un programme en ViSiDiA en mode synchrone qui calcul une approximation de la dimension doublante d'un graphe par la méthode des r-net. A la fin de l'exécution, tous les sommets devront contenir cette valeur commune dans une de leur étiquette ("label" par exemple). Si vous avez besoin de distinguer un sommet, vous utiliserez le sommet 0. Dans la construction, n'hesiter pas à colorier les sommets d'un $r/2$-net.

Rappels :

`putProperty("label", String)` permet de changer l'étiquette du sommet, et donc d'afficher n'importe quel résultat en modifiant `String`. Par exemple, vous pourrez afficher le nombre de sommets à une certaine distance.

`putProperty("toto", String)` permet de définir une nouvelle étiquette, `toto`, pour un sommet qui sera visualisable en sélectionnant le sommet, puis en cliquant sur l'icône information "I" en bleu du simulateur.