Coloration

Master 2 Informatique - UFR S.A.T

Pr. Ousmane THIARE

ousmane.thiare@ugb.edu.sn
http://www.ousmanethiare.com/

16 avril 2020

Chapitre 6 : 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.

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 processeurs. 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 (1920x1080 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.

Une coloration propre d'un graphe G est une fonction c : $V(G) \rightarrow N$ 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 $c(u) \in \{0, \cdots, k-1\}$ pour tout sommet u de G.
Colorier un graphe revient à partitionner les sommets en ensembles indépendants. Les ensembles indépendants 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 $\chi(G) \leq 2$ 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 $\chi(G) \leq 4$.

Dans ce chapitre, on ne va pas s'intéresser 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 $2^{O(\sqrt{logn})}$, 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 $\Delta < 2^{O(\sqrt{logn})}$, 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.

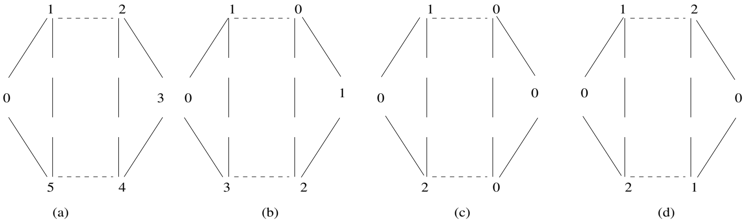
Parmi les hypothèses du modèle LOCAL, chaque sommet u possède une identité unique, ID(u). Donc c(u)=ID(u) est trivialement une coloration propre, certes qui utilise potentiellement beaucoup de couleurs. Le but va être de réduire ce nombre tout en maintenant une coloration propre.

En séquentiel, il y a une technique très simple pour réduire le nombre de couleurs. Initialement, c(u) :=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.

Plus formellement, on définit FirstFree(u) comme étant le plus plus petit entier $k \geq 0$ 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 $FirstFree(u) \in \{0, \cdots, deg(u)\}$. En séquentiel, on fait donc :

Algorithme FirstFit(G)

1. Pour tout sommet u, poser c(u) :=ID(u).
2. Pour tout sommet u, poser c(u) :=FirstFree(u).



Exemple de coloration suivant la règle Firstfree. En (a) coloration initiale. En (b) résultat en appliquant l'algorithme séquentiel avec l'ordre 0, 1, ..., 5. En (c) résultat en appliquant FirstFree(u). 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

Pour tout graphe G, $\chi(G) \leq \Delta(G) + 1$.

Preuve :
Il suffit d'analyser l'algorithme séquentiel FirstFit. 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), $c(u) \in \{0, \cdots, deg(u)\}$ (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 précédente. En terme de degré maximum, c'est la meilleure borne possible puisque pour une clique K_n à n sommets, on a $\chi(K_n) = n$ alors que $\Delta(K_n) = n - 1$. Autrement dit, $\chi(K_n) = \Delta(K_n) + 1$.

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 1.3(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'idéal 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 on verra des exemples plus tard où c'est différent.

Definition

Le **passage** d'un processus p à travers la session X (noté $\langle p, X \rangle$) est un intervalle $[t_1, t_2]$ (de temps) pendant lequel le processus p exécute sa section critique. Un passage est initié en t_1 et accompli en t_2.

Definition

Soit T l'ensemble des passages. Une **ronde** (round en anglais) R_Y de T est l'ensemble maximal des passages consécutifs de T qui sont des passages à travers la session Y.

On suppose donc qu'au départ c est une $k_0 - coloration$ et que les valeurs k_0, k_1 sont connus de tous les processeurs.
Algorithme ReducePalette(k_0, k_1) (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 $c(u) = min\{t \geq 0 : t \neq \{c_1, \cdots, c_{deg(u)}\}$

L'instruction NewPulse permet en mode synchrone de démarrer une nouvelle ronde, c'est-à-dire un nouveau cycle Send/Receive/calcule. Dans un système asynchrone, cette instruction peut être vue comme une synchronisation, un appel à une routine du synchroniseur par exemple (voir le chapitre 5).

Proposition

Si $k_0 \geq k_1 \geq \Delta(G) + 1$, l'algorithme ReducePalette(k_0, k_1) construit en $k_0 - k_1$ rondes une $k_1 -$ coloration pour G ayant une $k_0 -$ coloration.

Preuve : Il faut exactement $k_0 - k_1$ rondes pour exécuter ReducePalette(k_0, k_1) 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.

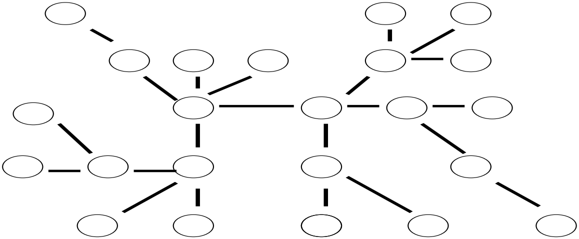
Preuve :
Les sommets u sur lesquels FirstFree est appliquée ont une couleur initiale $c(u) = k \geq k_1 \geq \Delta(G) + 1$. Après cette application sur u, $c(u) \leq deg(u) \leq \Delta(G) \leq k_1 - 1$.
On a donc construit une $k_1 - coloration$ pour G, en temps $k_0 - k_1$.
On peut donc construire une $(\Delta(G) + 1) - coloration$ en temps synchrone $O(n)$. On va voir comment faire beaucoup mieux, dans certains cas tout au moins.
On va s'intéresser ici à la coloration des arbres. On va présenter un algorithme très rapide pour calculer un 3-coloration.

Une 2-coloration serait envisageable pour les arbres, 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.

Definition

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 sont clairement 1-orientables : il suffit de fixer un sommet comme racine et la relation 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). 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.



Un graphe 1−orientable

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 $Pre(u) = \perp$.

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.

Algorithme pour les 1-orientations
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 $\lceil logn \rceil$ bits. En effet, pour représenter un entier de $[0, 2^k[$ il faut k bits, et k=$\lceil logn \rceil$ est le plus petit entier tel que $n \leq 2^k$.

Algorithme pour les 1-orientations

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

Algorithme pour les 1-orientations

Algorithme Color6 (code du sommet u)

1. Poser x :=ID(u) et l :=$\lceil logn \rceil$
2. Si $Pere(u) = \perp$, alors y :=1si x=0 et y := 0 sinon.
3. Répéter :
   (a) NewPulse
   (b) Send(x, v) à tout voisin v de u
   (c) Si $Pere(u) \neq \perp$, alors Receive(y, Pere(u))

Algorithme pour les 1-orientations

Algorithme Color6 (code du sommet u)

(d) t :=l

(e) x :=PosDiff(x, y)

(f) l :=1+$\lceil log l \rceil$

Jusqu'à ce que l=t

4. Color(u) := x

On note log x le logarithme en base 2 de x, et $\lceil x \rceil$ la partie entière supérieure de x.

La fonction PosDiff

Dans la suite on notera par bin(x) l'écriture binaire de l'entier x. On note par |s| la longueur de la chaîne s. On supposera qu'on écrit bin(x) de droite à gauche (donc les bits de poids faible à droit). On note s[i] le i-ème caractère de la chaîne s, $i \in [0, |s|[$. Ainsi on écrit bin(6) = 110, mais bin(6)[0] = 0, bin(6)[1] = bin(6)[2] = 1.

La fonction PosDiff(x,y), définie pour des entiers $x \neq y \in N$, 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 (par exemple la première) telle que $B_x[p] \neq B_y[p]$.
2. PosDiff(x, y) est alors la valeur dont l'écriture binaire est $bin(p) \circ B_x[p]$.

La fonction PosDiff

Exemple (en choisissant pour p la plus petite valeur possible) :

- $PosDiff(10, 4) = PosDiff(1010_2, 100_2) = 11_2 = 3$ car p=1 et b=1.
- $PosDiff(6, 22) = PosDiff(110_2, 10110_2) = 1000_2 = 8$ car $p = 4 = 100_2$ et b = 0.
- $PosDiff(4, 0) = PosDiff(100_2, 0_2) = 101_2 = 5$ car $p = 2 = 10_2$ et b = 1.

Par construction $|bin(PosDiff(x, y))| = 1 + \lceil log|bin(max\{x, y\})|\rceil$. La fonction PosDiff vérifie la propriété importante suivante :

La fonction PosDiff

Propriété

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

Preuve : Supposons que $x \neq y$ et $y \neq z$. Soit (p,b) la position et le bit obtenus dans le calcul de PosDiff(x,y) et (p', b') dans le calcul de PosDiff(y,z). Si PosDiff(x,y)=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 $b' = B_y[b'] = B_y[p]$. Mais x et y diffèrent à la position p, donc $B_x[p] \neq B_y[p]$ et $b \neq b'$: contradiction. Donc $PosDiff(x, y) \neq PosDiff(y, z)$.

Cette propriété garantit que la coloration reste propre tant que chaque sommet de couleur x utilise PosDiff(x, y) avec la couleur y de son père. Il faut quand même le vérifier pour la racine qui n'a pas de père, mais seulement un père virtuel de couleur fixe $y \in \{0, 1\}$ (ce père virtuel ne fait pas de PosDiff). On remarque que pour tout x, PosDiff(x, 1) = 0 et PosDiff(x, 0) = 1. Donc après la première ronde la racine ne change plus de couleur qui devient 0 ou 1.

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».On a $x_0(u) = ID(u)$, $l_0 = \lceil logn \rceil$, $l_1 = 1 + \lceil log\lceil logn \rceil \rceil$, etc.
La proposition suivante va nous montrer que l'algorithme termine toujours.

Proposition

Si $l_{i-1} > 3$, alors $l_i < l_{i-1}$. Sinon $l_i = l_{i-1}$.

Introduction
Définition du problème
Réduction de palette
Coloration des arbres
Analyse de l'algorithme
Résumé
De six à trois couleurs
Coloration des k-orientations
Coloration des cycles, borne inférieure

Preuve : On a $l_i = 1 + \lceil logl_{i-1} \rceil \leq 2 + logl_{i-1}$. La proposition est donc vraie dès que $2 + logl_{i-1} < l_{i-1}$, c'est-à-dire lorsque $4l_{i-1} < 2l_{i-1}$. C'est vrai si $l_{i-1} \geq 5$. Pour les valeurs inférieures, on le vérifie exhaustivement :

l_{i-1}	1	2	3	4	5
$1 + \lceil logl_{i-1} \rceil$	2	2	3	3	4

C'est donc vrai dès que $_{li-1} > 3$.

Preuve :
Si $l_{i-1} = 3$,alors $l_i = 1 + \lceil log 3 \rceil = 3$. Si $l_{i-1} = 2$, alors $l_i = 1 + \lceil log 2 \rceil = 2$. Et si $l_{i-1} = 1$, alors $l_i = 1 + \lceil log 1 \rceil = 1$. Il n'est pas possible d'avoir $l_{i-1} = 0$, car $n \geq 2$ (si n=1 il n'y a pas d'arête dans le graphe).

On déduit de la proposition précédente que Color6 termine toujours puisque soit $l_0 > 3$ et alors la valeur de l diminue strictement. Soit $l_{i-1} \leq 3$, et alors la valeur suivante li reste aussi à l_{i-1}. Donc on sort toujours de la boucle « Répéter », le test « l = t » étant alors vrai. Montrons maintenant qu'il calcule une 6-coloration. Pour ceci, montrons que l_i borne supérieurement l'écriture binaire de toutes les couleurs du graphes. Dit autrement :

Proposition

À chaque étape i et sommet u, $|bin(x_i(u))| \leq l_i$.

Preuve : Soit $c_i = max_u x_i(u)$ le maximum de $x_i(u)$ pris sur tous les sommets du graphe. Comme $|bin(x_i(u))| \leq |bin(c_i)|$, il suffit de démontrer la proprosition pour c_i. On le fait par induction sur i.
C'est vrai pour i=0, $c_i < n$ et donc $|bin(c_i)| \leq \lceil logn \rceil = l_0$.
Supposons que $|bin(c_i)| \leq l_i$. On a
$ci + 1 = PosDiff(c_i, x_i(u))$ pour un certain sommet u avec $x_i(u) \neq c_i$. Par construction,
$|bin(c_i + 1)| \leq 1 + \lceil log|bin(c_i)| \rceil$ car $c_i \leq x_i(u)$. Par induction, $|bin(c_i)| \leq l_i$. Donc
$|bin(c_{i+1})| \leq 1 + \lceil logl_i \rceil = l_{i+1}$ par définition de l_{i+1}.

Proposition

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

Preuve : D'après l'avant dernière proposition, l'algorithme se termine avec $l_i \leq 3$. Si $l_i \leq 2$, alors d'après la dernière proposition, on a $|bin(x_i(u))| \leq 2$. Donc $x_i(u) \in \{0, \cdots, 3\}$. Si $l_i = 3$, c'est que $l_{i-1} = 3$ sinon on ne s'arrête pas à l'étape i mais i+1. Donc, pour tous les sommets u, $x_{i-1}(u) \in \{0, \cdots, 7\}$ puisque $|bin(x_{i-1}(u)| \leq 3$. On vérifie facilement que $PosDiff(x, y) \leq 5$ pour tout entier $x \neq y \in \{0, \cdots, 7\}$- pour la paire (p,b) qui définit PosDiff(x,y) on a au pire $p = 10_2$ et b=1. Donc $x_i(u) \leq 5$.

On a vu que l'algorithme, grâce à la propriété 1, que l'algorithme maintient avec $x_i(u)$ une coloration propre. D'après la dernière proposition il s'agit d'une 6-coloration. On note

$$log^{(i)} l = \overbrace{logloglog \cdots log}\, l$$

La fonction log itérée i fois. On a $log^{(0)} l = l$

Proposition

Pour tout i>0, si $l_{i-1} > 3$, *alors* $l_i \leq 2 + \lceil log^{(i+1)} n \rceil$.

Preuve : Exercice...

Lemme

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

Preuve : On a vu dans la dernière proposition que Color6 calculait une 6-coloration. Soit t le nombre de boucles « Répéter » réalisées par l'algorithme. C'est le temps pris par l'algorithme.
Le résultat est vrai si $l_0 \leq 3$. En effet, si $l_0 = \lceil logn \rceil \leq 3$, l'algorithme prend alors une seule ronde. Et $log^* n$ est au moins 1 dès que n>1. (Parfois $log^* n$ est défini comme le plus petit $i \geq 0$ tel que $log^{(i)} \leq 2$).

Preuve :
Supposons que $l_0 > 3$. Dans ce cas si le plus petit entier tel que $l_i \leq 3$ est i=t-1. En effet, d'après la proposition 13 et l'algorithme, la valeur de l diminue jusqu'à atteindre une valeur ≤ 3 et reste une étape supplémentaire à la même valeur et sort de la boucle.
A FINIR

On a $l_t \leq 3$. Le nombre de rondes de l'algorithme est précisément t.

$2 + \lceil log^{(i+1)} n \rceil \leq 3$

$\Leftrightarrow 2 + \lceil log^{(i+1)} n \rceil \leq 3$

$\Leftrightarrow \lceil log^{(i+1)} n \rceil \leq 1$

$\Leftrightarrow log^{(i+1)} n \leq 1$

$\Leftrightarrow i + 1 = log^* n$

En posant $t = (log^* n) - 1$, on a donc que $2 + \lceil log^{(i+1)} n \rceil \leq 3$, et d'après la dernière proposition, $l_t \leq 2 + \lceil log^{(i+1)} n \rceil \leq 3$.

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 $l := \lceil logm \rceil$ dans l'instruction 1 de l'algorithme Color6.

Pour obtenir une 3-coloration 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 5, 4, 3 pour ne garder que 2, 1, 0. Au total cela rajoute 6 rondes.
A FINIR
Algorithme Color6to3 (code du sommet u)
Pour k := 5, 4, 3 faire :
1. ShiftDown
2. ReducePalette(k+1,k)
D'après la proposition 12, ReducePalette(k+1, k) prend une ronde et élimine une couleur, la couleur k.
D'où le résultat final suivant :

Théorème

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

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 $O(logn)$.

Théorème

On peut calculer une (k+2)-coloration en temps $klog^* m + 2^{O(k)}$ pour tout graphe k-orientable possédant une m-coloration.

Preuve : A faire...

Remarquons cependant qu'il est trivial de calculer une $\Delta - \textit{orientation}$. En utilisant le résultat précédent on en déduit une $(\Delta + 2) - \textit{coloration}$. Une étape supplémentaire, grâce à $\textit{ReducePalette}(\Delta + 2, \Delta + 1)$, permet d'obtenir une $(\Delta + 1) - \textit{coloration}$. D'où :

Théorème

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.

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 droite et de gauche sont connus de chaque sommet u par les relations Succ(u) et Pred(u).

Coloration des cycles

L'idée est de calculer PosDiff en alternant avec son successeur et son prédécesseur. 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. L'astuce est que lorsqu'un sommet u échange sa couleur avec ses voisins, il est en mesure de simuler le calcul de PosDiff entre l'un de ses voisins v et son Père(v) (qui est u bien sûr).

Algorithme ColorRing (code du sommet u)

1. Poser x :=ID(u) et $l := \lceil logn \rceil$
2. Répéter :
   (a) NewPulse
   (b) Send(x, Succ(u)) et Send(x, Pred(u)))
   (c) Receive(z, Succ(u)) et Receive(y, Pred(u)))
   (d) t :=l
   (e) x :=PosDiff(PosDiff(x,z),PosDiff(y,x))
   (f) $l := 1 + \lceil log(1 + \lceil logl \rceil) \rceil$

Jusqu'à ce que l=t

3. Poser Color(u) := x
4. ReducePalette(6,3)

Coloration des cycles

D'où le résultat :

Proposition

L'algorithme ColorRing produit une 3-coloration en temps $\frac{1}{2} log^* n + O(1)$ sur les cycles orientés à n sommets.

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.

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.

Borne inférieure

Proposition

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 $C(n) - 1 \leq P(n) \leq C(n)$.

Preuve : L'inégalité $P(n) \leq C(n)$ 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.

Borne inférieure

Montrons maintenant que $C(n) - 1 \leq P(n)$. 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'est 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.

Borne inférieure

On remarque que les seuls sommets voisins pouvant être en confits 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 $\{0, 1, 2\} - \{c(z), c(y)\}$ (c(z) et c(y) sont connues de x). Le seul conflit possible est ainsi éliminé. Donc $C(n) \leq P(n) + 1$.

Théorème

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

Bibliographie

[*BE*08] Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition. In 27th Annual ACM Symposium on Principles of Distributed Computing (PODC), pages 25–34. ACM Press, August 2008.

[*BE*09] Leonid Barenboim and Michael Elkin. Distributed $(\delta + 1)$-coloring in linear (in δ) time. In 41st Annual ACM Symposium on Theory of Computing (STOC), pages 111–120. ACM Press, May 2009.

Bibliographie

[*CV*86] Richard Cole and Uzi Vishkin. Deterministic coin tossing and accelerating cas- cades : micro and macro techniques for designing parallel algorithms. In 18th Annual ACM Symposium on Theory of Computing (STOC), pages 206–219. ACM Press, 1986.
[*Ryb*11] Joel Rybicki. Exact bounds for distributed graph colouring. PhD thesis, University of Helsinki, Dept. of Computer Science, May 2011.

Bibliographie

[*Lin*92] Nathan Linial. Locality in distributed graphs algorithms. SIAM Journal on Computing, 21(1) :193–201, 1992.
[*PS*92] Alessandro Panconesi and Aravind Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In 24th Annual ACM Symposium on Theory of Computing (STOC), pages 581–592. ACM Press, 1992.