Machines P-RAM

Résumé: Dans ce TD, nous regardons divers algorithmes sur P-RAM. Si le saut de pointeurs est une technique fondamentale, il est intéressant de remarquer que la plupart des algorithmes n'ont rien à voir avec leur équivalent séquentiel et se résument rarement à un bête saut de pointeurs.

1 Sélection dans une liste

▷ **Question 1.** Soit L une liste contenant n objet coloriés soit en bleu, soit en rouge. Concevoir un algorithme EREW efficace qui sépare les éléments bleus des éléments rouges.

Réponse. Il suffit d'utiliser la technique de saut de pointeur vu en cours. Chaque processeur détermine le processeur bleu qui le suit dans la liste en temps $O(\log n)$.

```
EXTRAIT-BLEUS
 1: Pour tout i en parallèle
 2:    Si suivant(i) = NIL Ou couleur(suivant(i)) = bleu Alors
 3:       fini(i) ←Vrai
 4:       bleu(i) ←suivant(i)
 5: Tant que il existe un nœud i tel que fini(i) = Faux
 6:    Pour tout i en parallèle
 7:       Si fini(i) = Faux Alors
 8:          fini(i) ← fini(suivant(i))
 9:          Si fini(i) = True Alors bleu(i) ← bleu(suivant(i))
10:          suivant(i) ← suivant(suivant(i))
```

Cet algorithme est bien EREW mais cela demande une petite explication. Il fonctionne comme l'algorithme du saut de pointeur en parallèle sur plusieurs listes dont le nœud terminal est bleu ou nil et que l'on conserve un pointeur sur la fin de sa liste, donc sur le prochain élément bleu. Chaque saut de pointeur se fait sur des listes indépendantes et n'interfère pas avec les autres. À la fin de l'algorithme, la liste des éléments bleu commence avec le premier élément de la liste initiale si ce dernier est bleu et avec son successeur bleu sinon. □

2 Composantes connexes

On souhaite concevoir un algorithme CRCW qui permet de calculer les composantes connexes d'un graphe dont les sommets sont numérotés. Plus précisément, on cherche un algorithme qui renvoie un tableau C de taille n tel que $C(i) = C(j) = k$ si et seulement si i et j sont dans la même composante connexe et k est le plus petit indice des sommets de cette composante.

Définition 1. À toute étape de l'algorithme, on appellera pseudo-sommet étiqueté par i l'ensemble de sommets $j, k, l, \cdots \in V$ tels que $C(j) = C(k) = C(l) = \cdots = i$. On assimilera le pseudo-sommet i étiqueté par i au sommet étiqueté par i.

Un des invariants de l'algorithme est que le plus petit indice des sommets constituant un pseudo-sommet étiqueté par i est i et que les sommets appartenant à un pseudo-sommet sont dans la même composante connexe. Cette assertion est donc vraie si on initialise C tel que pour tout $i \in V = [\![1, n]\!] : C(i) = i$. Ceci signifie que chaque processeur se considère comme sommet de référence de sa composante connexe. L'objectif de l'algorithme est de modifier leur point de vue…

Définition 2. Une arborescence k-cyclique ($k \geqslant 0$) est un graphe orienté faiblement connexe tel que :
- tout sommet a un degré sortant égal à 1 et
- il existe exactement un circuit de longueur $k + 1$.

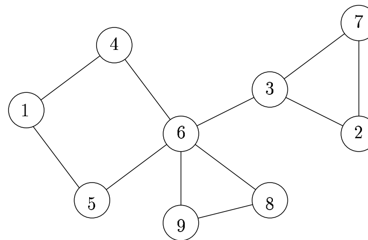
On appelle étoile une arborescence 0-cyclique

L'invariant précédent est donc que le graphe orienté $(V, \{(i, C(i)) \mid i \in V\})$ est constitué d'étoiles. On peut donc identifier pseudo-sommet et étoiles, le centre de l'étoile étant l'indice du pseudo-sommet. Le calcul des composantes connexes s'effectue en enchaînant plusieurs fois de suite les deux fonctions suivantes :

Gather
1: **Pour tout** $i \in S$ **en parallèle**
2: $T(i) \leftarrow \min\{C(j) \mid \{i,j\} \in A, C(j) \neq C(i)\}$
{si l'ensemble est vide, on associe $C(i)$}
3: **Pour tout** $i \in S$ **en parallèle**
4: $T(i) \leftarrow \min\{T(j) \mid C(j) = i, T(j) \neq i\}$
{si l'ensemble est vide, on associe $C(i)$}

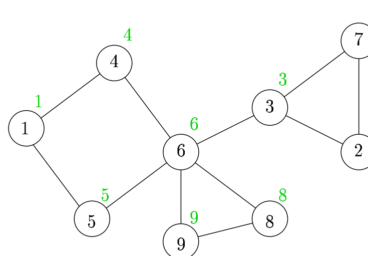
Jump
5: **Pour tout** $i \in S$ **en parallèle**
6: $B(i) \leftarrow T(i)$
7: **Répéter** $\log n$ **fois**
8: **Pour tout** $i \in S$ **en parallèle**
9: $T(i) \leftarrow T(T(i))$
10: **Pour tout** $i \in S$ **en parallèle**
11: $C(i) \leftarrow \min\{B(T(i)), T(i)\}$

▷ **Question 2.** Appliquer la fonction Gather au graphe suivant.

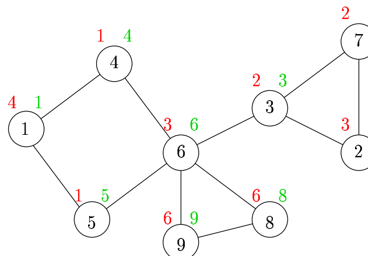


Puis la fonction Jump , puis la fonction Gather, et ainsi de suite.... Il sera instructif d'observer l'effet des opérations sur les graphes orientés $(V, \{(i, T(i)) \mid i \in V\})$ et $(V, \{(i, C(i)) \mid i \in V\})$.

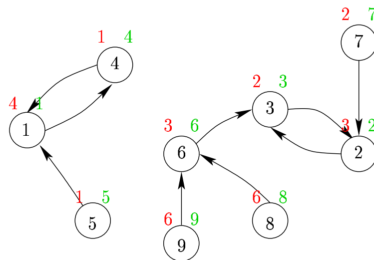
Réponse. Avant le début de l'algorithme, les valeurs de C sont initialisées comme ci-dessous :



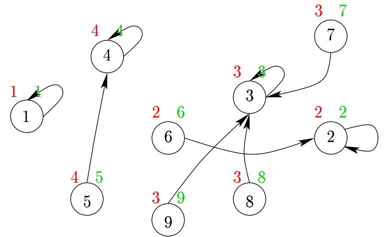
Après l'application de la fonction Gather, les valeurs de T et de C sont les suivantes :



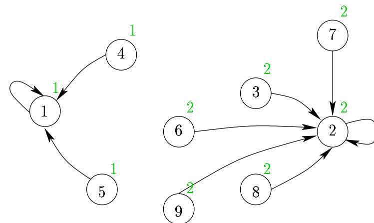
Si on considère le graphe orienté $(V, \{(i, T(i)) \mid i \in V\})$, on peut s'apercevoir qu'il n'est constitué que d'arborescences 1-cycliques :



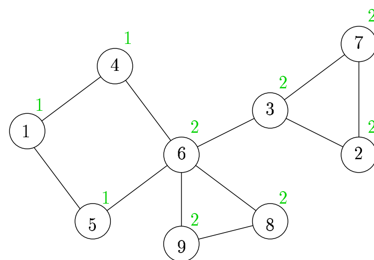
Après le saut de pointeur de la fonction Jump, il ne reste plus que des étoiles :



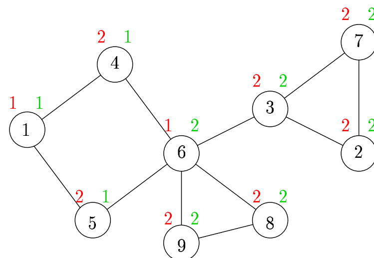
qui sont fusionnées dans le graphe $(V, \{(i, C(i)) \mid i \in V\})$ lors de la dernière opération de Jump :



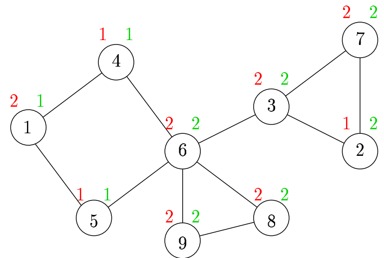
On notera qu'il y n'y a donc plus que deux pseudo-sommets dans la composante connexes. On se retrouve donc dans la situation suivante au début de l'appel à la fonction Gather :



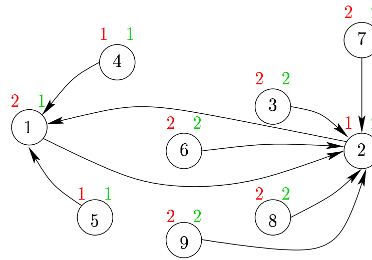
Après la première étape, T est mis à jour comme ci-dessous :



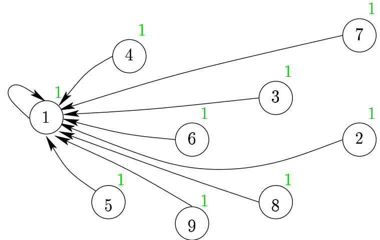
et enfin comme ceci :



On se retrouve donc avec le graphe orienté $(V, \{(i, T(i)) \mid i \in V\})$ constitué d'arborescence 1-cycliques :



Les pseudo-sommets 1 et 2 sont donc fusionnés à la fin de l'appel à Jump :



.

On a bien calculé les composantes connexes du graphe. □

▹ **Question 3.** Montrer qu'après l'application de la fonction Gather, les composantes connexes contenant plusieurs pseudo-sommets induisent des arborescences 1-cycliques dans le graphe orienté $(V, \{(i, T(i)) \mid i \in V\})$. On notera également que le plus petit pseudo-sommet d'une arborescences 1-cyclique appartient au cycle.

Réponse. Tout d'abord, il est clair que lorsqu'une composante connexe ne contient qu'un seul pseudo-sommet, l'étoile correspondante est transférée dans T sans modification.

Si une composante connexe contient plusieurs pseudo-sommets, par contre, T décrira un ensemble d'arborescences 1-cycliques contenues dans cette composante. En effet, tout pseudo-sommet de cette composante contient au moins un sommet adjacent à un sommet d'un autre pseudo-sommet. Gather fait pointer le représentant de chaque pseudo-sommets –via T– vers le représentant d'un autre pseudo-sommet, tout en laissant pointer les autres sommets vers leur pseudo-sommet initial. En clair, si deux groupes se touchent, leurs représentants se retrouvent liés dans le graphe orienté induit par T et les autres sommets continuent de pointer vers leur représentant respectif. Chaque composante de ce graphe orienté doit contenir au moins une boucle car le degré sortant de chaque sommet vaut 1. Il y a au plus une boucle car sinon il y aurait un pseudo-sommet avec deux valeurs pour T. Enfin la boucle en question ne peut être que de longueur 2 sinon (si elle était de longueur 1) i et $T(i)$ seraient identiques ou (si elle était de longueur supérieure à 2) il existerait un sommet i sur la boucle tel que $T(i)$ n'est pas le plus petit indice des pseudo-sommets adjacents au pseudo-sommet i. □

▹ **Question 4.** Montrez que la la fonction Jump transforme une arborescence 1-cyclique en étoile (ou pseudo-sommet).

Réponse. Cette étape fusionne tous les sommets d'une même arborescence 1-cyclique en une étoile indexée par le sommet de plus petit indice en utilisant la technique de saut de pointeur. En effet, étant donnée la configuration d'une étoile, à l'issu des sauts de pointeurs, chaque sommet à pour valeur de T l'une des anciennes valeurs d'un des sommets de la boucle. La dernière étape permet donc d'assigner à tous les sommets la plus petite valeur des sommets des l'arborescence à laquelle ils appartiennent. □

▹ **Question 5.** Montrez qu'après $\log n$ enchaînements des fonctions Gather et Jump, les composantes connexes du graphe sont représentés par les pseudo-sommets induits par C.

Réponse. Il suffit de montrer que le nombre de pseudo-sommets diminue au moins de moitié à chaque étape. Concentrons nous sur les représentants des pseudo-sommets et intéressons nous au graphe induit par T sur ces sommets. Dans un tel graphe, deux pseudo-sommets i et j sont connectés si et seulement si il existe deux sommets k et l connectés dans le graphe initial et tels que $C(k) = i$ et $C(l) = j$. Dans l'exemple précédent, cela revient à avoir un graphe composé des pseudo-sommets 1 et 2 reliés par une arête après la

première application de GATHER. La fonction JUMP fusionne tous les pseudo-sommets ainsi reliés en un seul pseudo-sommet. Ainsi le nombre de pseudo-sommets dans une même composante connexe diminue au moins de moitié à chaque étape et $\log n$ enchaînements de GATHER et JUMP suffisent à calculer les composantes connexes. □

▷ **Question 6.** Quelle est la complexité de l'algorithme ? Combien de processeurs avez-vous utilisé ?

Réponse. Premièrement, on peut remarquer que la boucle séquentielle de la fonction JUMP implique que le temps de calcul total est au moins $O(\log^2 n)$, et ce quelque soit le nombre de processeurs. On va d'abord montrer que ce temps peut être atteint avec $O(n^2)$ processeurs.

On peut déjà remarquer qu'avec autant de processeurs, la première et la dernière boucle de JUMP prennent un temps $O(1)$ et le saut de pointeur un temps $O(\log n)$. En fait, $O(n)$ processeurs suffisent pour arriver à un tel temps de calcul de la fonction JUMP. Si on veut arriver à un temps total de l'ordre de $O(\log^2 n)$, on va donc devoir montrer que la fonction GATHER peut s'exécuter en temps $O(\log n)$.

Le calcul du maximum de n valeurs peut se faire en temps $O(1)$ avec n^2 processeurs. Cependant, ce sont les maxima de plusieurs ensembles que l'on veut calculer. Il faut donc ruser un peu.

1: **Pour tout** $i \in S$ **en parallèle**
2: $T(i) \leftarrow \min\{C(j) \mid \{i,j\} \in A, C(j) \neq C(i)\}$
{si l'ensemble est vide, on associe $C(i)$}

La boucle précédente peut-être transformé en le code suivant

3: **Pour tout** $i, j \in S$ **en parallèle**
4: **Si** $\{i,j\} \in A$ **Et** $C(i) \neq C(j)$ **Alors** $Temp(i,j) \leftarrow C(j)$ **Sinon** $Temp(i,j) \leftarrow \infty$
5: **Pour tout** $i \in S$ **en parallèle**
6: $Temp(i,1) \leftarrow \min\{Temp(i,j) \mid j \in S\}$
7: **Pour tout** $i \in S$ **en parallèle**
8: **Si** $Temp(i,1) = \infty$ **Alors** $T(i) \leftarrow C(j)$ **Sinon** $T(i) \leftarrow Temp(i,j)$

La première boucle se fait clairement en temps $O(1)$ avec $O(n^2)$ processeurs (en réalité avec $O(|A|)$ processeurs) sur une CRCW. Les deux boucles suivantes se font en temps $O(\log n)$ avec $O(n)$ processeurs en divisant pour régner...
La seconde boucle de la fonction JUMP se traitant de façon similaire, le calcul des composantes connexes s'effectue donc bien en temps $O(\log^2 |S|)$ avec $O(|S| + |A|)$. Cependant, la fonction JUMP gaspille des ressources. Le théorème de Brent nous permet donc de diminuer le nombre de processeurs à $\left(\frac{n^2}{\log n}\right)$ (en réalité à $O\left(\frac{|A|}{\log |S|} + |S|\right)$ processeurs) sans modifier le temps de calcul. □

3 Rupture de symétrie déterministe

On veut concevoir un algorithme EREW qui permet de sélectionner «beaucoup» d'objets dans une liste chaînée sans jamais choisir deux objets adjacents dans la liste. Chaque objet est associé à un processeur mais le numéro du processeur n'est pas relié à l'ordre des éléments dans la liste (ça serait trop facile...).

▷ **Question 7.** Concevoir un algorithme qui permet sélectionner un nombre optimal d'objets en temps $O(\log n)$ sur une EREW.

Réponse. Il suffit d'utiliser l'algorithme vu en cours pour la distance à la fin de la liste. Chaque processeur détermine sa distance à la fin de la liste en temps $O(\log n)$. Si cette distance est paire, son objet est sélectionné, sinon il ne l'est pas. □

Dans la suite, on va montrer qu'il est possible d'extraire «beaucoup» d'éléments en un temps $O(\log^* n)$ où

$$\log^* n = \min\left\{i \mid \log^i n \leqslant n\right\}.$$

Dans l'expression précédente, $\log^i$ représente la composition de i fois la fonction log. La fonction $\log^*$ a une croissance très lente, puisque $\log^*(2^{65536}) = 5$.

Nous allons maintenant préciser le sens de «beaucoup» d'éléments a partir de la notion d'ensemble indépendant maximal.

Définition 3. Un ensemble de sommets V' d'un graphe $G = (V, E)$ est indépendant ssi

$$\forall (a, b) \in E, \text{ au plus un élément de } \{a, b\} \text{ est dans } V'.$$

Un ensemble de sommets V' d'un graphe $G = (V, E)$ est indépendant et maximal (attention pas maximum) ssi l'ajout de tout sommet à V' en fait un ensemble non indépendant.

Dans ce qui suit, notre objectif est d'arriver à extraire un ensemble indépendant maximal de la liste en temps $O(\log^* n)$. Pour cela, on va commencer par colorier la liste (avec 6 couleurs). On va construire un algorithme qui part d'une n-coloration (où la couleur de chaque objet est déterminée par la couleur du processeur qui lui est associe) et qui diminue à chaque étape le nombre de couleurs utilisées.

▷ **Question 8.** Donner un algorithme astucieux pour diminuer le nombre de couleurs utilisées tout en gardant une coloration (on pourra raisonner sur le codage binaire des couleurs).

Réponse. Pour obtenir une coloration en un temps $O(\log^* n)$, il faut que le nombre de couleurs nécessaire r_{k+1} après l'étape $k + 1$ soit de l'ordre de $\log r_k$.
Considérons deux nœuds successifs a et b à l'étape k dont les couleurs sont respectivement codées par

$$\mathcal{C}^k(a) =< a_{r_k-1}, a_{r_k-2}, \ldots, a_0 > \text{ et } \mathcal{C}^k(b) =< b_{r_k-1}, b_{r_k-2}, \ldots, b_0 > .$$

À l'étape $k + 1$, on définit le codage le codage de a par

$$\mathcal{C}^{k+1}(a) =<< i >, a_i >,$$

où i est le plus petit indice tel que $a_i \neq b_i$ et $< i >$ est le codage binaire de i.
Comme $i \in [0, r_k - 1]$, on vérifie que la longueur du codage binaire de i est $\leqslant \lceil \log r_k \rceil$. Ainsi, si on note

$$\mathcal{C}^{k+1}(a) =< a_{r_{k+1}-1}, a_{r_{k+1}-2}, \ldots, a_0 >,$$

le codage de a après la $k + 1$ étape, on a

$$r_{k+1} = \lceil \log r_k \rceil + 1.$$

Comme la couleur d'un nœud est définie à partir de la couleur de son successeur, il est nécessaire de définir la couleur du dernier nœud d de façon particulière, et on pose

$$\mathcal{C}^{k+1}(d) =< 0^{r_{k+1}}, d_0 > .$$

Pour vérifier la correction de l'algorithme, il suffit de vérifier qu'on obtient bien un coloriage à chaque étape, c'est à dire que si deux nœuds successifs a et b ont une couleur différente à l'étape k, ils ont une couleur différente à l'étape $k + 1$. La preuve est relativement aisée.
En effet, si $\mathcal{C}^k(a)$ et $\mathcal{C}^k(b)$ sont différents, le plus petit indice i où les deux codages différent est bien défini. Soit $\mathcal{C}^{k+1}(a) =<< i >, a_i >$ et $\mathcal{C}^{k+1}(b) =<< j >, b_j >$ les codages des couleurs de a et b après l'étape $k + 1$. Si $i \neq j$ alors les codages sont différents et si $i = j$ alors $a_i \neq b_i$ par construction.
On a donc construit un algorithme qui permet à chaque étape d'obtenir un nouveau coloriage des nœuds de la liste. Les questions suivantes permettront de préciser la complexité de cet algorithme. □

▷ **Question 9.** Pourquoi obtient-on 6 couleurs ? Montrer qu'on obtient 6 couleurs après $O(\log^* n)$ étapes.

Réponse. Si r_k désigne le nombre de couleurs utilisées pour le codage a l'étape k, on a vu que

$$r_{k+1} = \lceil \log r_k \rceil.$$

Si r_k vaut 3, on vérifie que r_{k+1} vaut également 3, et le nombre de couleurs utilisées stagne. Plus précisément, comme la position à laquelle les codages des couleurs de deux nœuds successifs peut être 0, 1 ou 2, les seuls codages qu'on peut obtenir sont $< 000 >, < 001 >, < 010 >, < 011 >, < 100 >, < 101 >$, ce qui conduit bien à 6 couleurs différentes.
Pour obtenir le 6 coloriage promis, on va utiliser l'algorithme suivant :

```
6-COLORIAGE
1: Tant que r ⩾ 5
2:    Effectuer une étape de l'algorithme de recoloriage
3: Effectuer 3 étapes de l'algorithme de coloriage
```

Les trois étapes finales nous assurent qu'on obtient bien un 6-coloriage. Pour étudier la complexité de l'algorithme, il est nécessaire de montrer le lemme suivant
Dans la boucle **Tant que** on a $r_k \geqslant 5$ et $r_k \leqslant \lceil \log n \rceil + 2$.
Cette propriété est vérifiée de façon immédiate pour $k = 1$, puisque $r_1 \leqslant \lceil \log n \rceil$. Supposons $r_k \geqslant 5$ et $r_k \leqslant \lceil \log^k n \rceil + 2$. On vérifie que

$$\begin{aligned}
r_{k+1} &= \lceil \log r_k \rceil + 1 \\
&\leqslant \lceil \log(\lceil \log^k n \rceil + 2) \rceil + 1 \\
&\leqslant \lceil \log(\log^k n + 3) \rceil + 1 \\
&\leqslant \lceil \log(2 \log^k n) \rceil + 1 \quad (\log^k n \geqslant 3) \\
&\leqslant \lceil \log(\log^k n) + 1 \rceil + 1 \\
&\leqslant \lceil \log^{k+1} n \rceil + 2
\end{aligned}$$

Soit $m = \log^* n$. On vérifie que $\log^{m-1} n + 2 \leqslant 4$ et donc le nombre d'itérations de la boucle **Tant que** est inférieur à $m - 2$. Le nombre d'étapes total de recoloriage de l'algorithme est $m + 1$. □