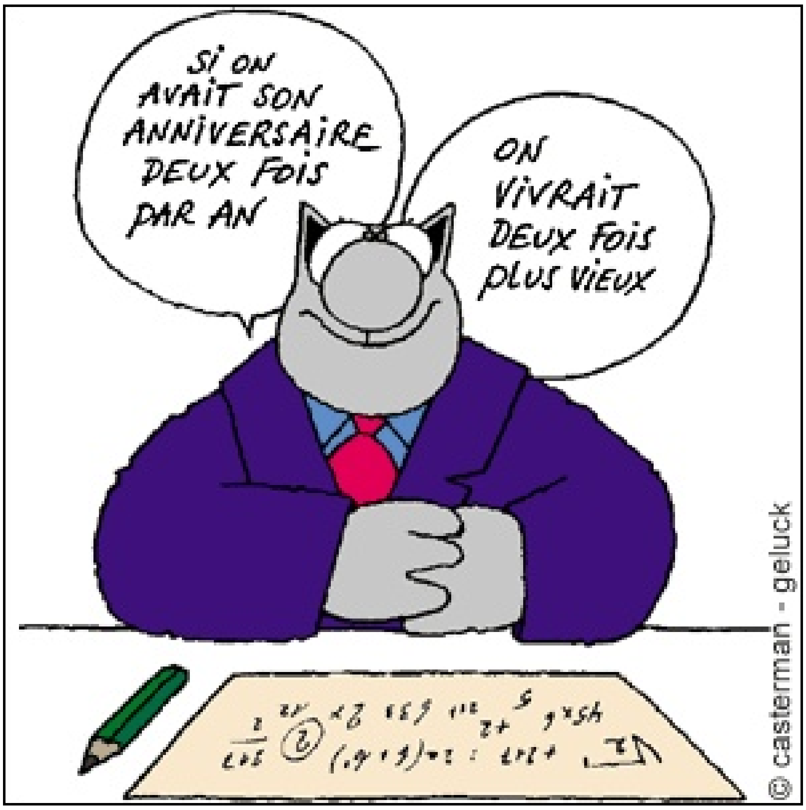
Algorithmique Avancée et Parallélisme

Adeline Bailly - GM4
Avec l'aide de Pauline Hubert & Thibault Lasnier



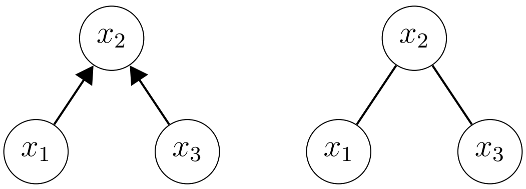
Année 2012-2013

Table des matières

Chapitre 1

Rappels sur les graphes

Définition 1.1 *Un graphe est composé de sommets et d'adjacents. Ou de sommets et d'arcs.*



1.1 Matrice d'adjacentes

Orienté :

$$\begin{array}{c|ccc} & x_1 & x_2 & x_3 \\ \hline x_1 & 0 & 1 & 0 \\ x_2 & 0 & 0 & 0 \\ x_3 & 0 & 1 & 0 \end{array}$$

Non Orienté :

$$\begin{array}{ccc} x_1 & x_2 & x_3 \end{array}$$
$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Lorsque le graphe n'est pas orienté, alors la matrice est symétrique. Si orienté, non symétrique (sauf rares exceptions).

1.2 Matrice d'incidences

$$\begin{array}{c|cc} & x_1x_2 & x_2x_3 \\ \hline x_1 & -1 & 0 \\ x_2 & 1 & 1 \\ x_3 & 0 & -1 \end{array}$$

-1 : départ
1 : arrivée

1.3 Liste de successeurs

(de prédécesseurs ou de voisins)

$S(x_i)$ = liste des successeurs (d'adjacence)
$P(x_i)$ = liste des prédécesseurs (pas nécessaire - dépend de l'utilisation)

1.4 Tables de successeurs

Exemple : Codage de matrice creuse.

Définition 1.2 *Chemin -*

- *Liste de sommets*
- *Liste d'arcs*

Remarque 1.1 *Chemin ($a \rightarrow b \rightarrow c$) $\neq$ Chaîne ($a \rightarrow b \leftarrow c$)*

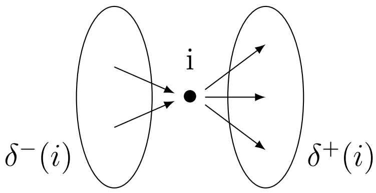
Définition 1.3 *Connexité -*
$\exists$ *une chaîne telle que*

- *Le graphe $G(X, U)$ est connexe ssi $\forall x, y \in X$, il existe une chaîne entre x et y*
- *Le graphe $G(X, U)$ (orienté) est fortement connexe ssi $\forall x, y \in X$, il existe un chemin entre x et y*

Chapitre 2

Exercices

Exercice 2.1 *Soit une représentation d'un graphe orienté par liste de successeurs. Combien de temps prend le calcul des demi-degrés extérieurs ? intérieurs ?*



$d^+(i) = Card(\delta^+(i))$
$d^-(i) = Card(\delta^-(i))$

On rappelle : n le nombre de sommets, m le nombre d'arêtes.

$d^+(i)$ = demi-degrès extérieurs $\rightarrow O(m)$
$d^-(i)$ = demi-degrès intérieurs $\rightarrow \cancel{O(m \times n)}$ (naïf)

En non-naïf (ie $\sim O(m)$) pour $d^-(i)$:

```
Pour tous les sommets
   Pour tout j dans S(i)
      d+(i) <- d+(j)+1
      d-(j) <- d-(j)+1
   FinPour
FinPour
```

Exercice 2.2 *Etant donné un graphe non orienté, proposer un algorithme pour détecter s'il est biparti. On suppose le graphe connexe*

Parcours en profondeur (Depth first search) $\sim O(m)$
On rappelle : n le nombre de sommets, m le nombre d'arêtes.

```
DFS(x)
Debut
   Marquer (x) {* Colorier - Hors algo - A faire avant *}
   Pour tout y voisin de x
      Si y est non marqué
         Marquer (y) {* Colorier 2nde couleur *}
         DFS(y)
      FinSi
   FinPour
Fin
```

Algo/td1_algo1.algo

La même chose mais sur 2 algo :

```
{* On suppose x rouge *}

DFS_rouge(x)
Debut
    Pour tout y voisin de x
        Si y est non colorié
            Colorier (y) en vert
            DFS_vert(y)
        Sinon
            Si y est rouge
                Return "Erreur"
                STOP
            FinSi
        FinSi
    FinPour
Fin
```

Algo/td1_algo2.algo

```
{* On suppose x vert *}

DFS_vert(x)
Debut
    Pour tout y voisin de x
        Si y est non colorié
            Colorier (y) en rouge
            DFS_rouge(y)
        Sinon
            Si y est vert
                Return "Erreur"
                STOP
            FinSi
        FinSi
    FinPour
Fin
```

Algo/td1_algo3.algo

Si on veut une seule procédure, on peut créer une procédure avec les couleurs en paramètre.

Exercice 2.3 *Etant donné un graphe orienté, proposer un algorithme pour détecter les différentes composantes connexes.*

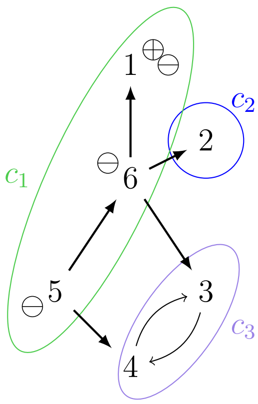
→ Les composantes fortement connexes en fait.

Algorithme de haut niveau pour trouver la composante fortement connexe d'un sommet x_0

1. Marquer x_0 avec $\oplus$ et $\ominus$
2. Marquer $\oplus$ tout successeur non déjà non marqué $\oplus$ d'un sommet marqué $\oplus$ (*Parcours de graphe*)

3. Marquer $\ominus$ tout prédécesseur non déjà non marqué $\ominus$ d'un sommet marqué $\ominus$ (*Parcours de graphe inverse*)

A la fin, les sommets marqués $\oplus$ et $\ominus$ forment la composante fortement connexe de x_0. Ici, l'algorithme a été appliqué pour $x_0 = 1$.



Cherchons la composante fortement connexe du sommet 1 : $c_1 = \{1, 5, 6\}$. Puis le sommet 2 : $c_2 = \{2\}$; enfin pour le sommet 3 : $c_3 = \{3, 4\}$.

On obtient le graphe réduit suivant :
$$\begin{tikzcd} & c_2 \\ c_1 \arrow[ur] \arrow[dr] & \\ & c_3 \end{tikzcd}$$

Chapitre 3

Diviser et construire (Divide & Conquer)

3.1 Principe

Diviser le problème en sous problèmes plus petits qui chacun facilitent la résolution du problème initial.

3.2 Exemple

Multiplication des grands entiers (Tableau d'entiers, redéfinition des opérations de base)

X et Y entiers écrits en chiffre en base 2

Multiplication "classique" : $O(n^2)$ opérations élémentaires (addition ou multiplication de chiffre)

3.3 Stratégie

Couper chaque nombre en 2 parties.

X :	A	B	$X = A.2^{n/2} + B$ (n pair)
Y :	C	D	$Y = C.2^{n/2} + D$

$$XY = AC.2^n + [(A-B)(D-C) + AC + BD].2^{n/2} + BD$$

$AC, AD, BC, BD \sim O(\frac{n^2}{4}) \Rightarrow XY \sim O(n^2)$

$T(n)$: temps de calcul pour multiplier 2 nombres de taille n
$T(1) = 1$

$\boxed{T(n) = 3T(\frac{n}{2}) + Kn}$ *A Montrer en exercice*

$T(n) = O(n^{log_2(3)})$ avec $log_2(3) \sim 1,59 < 2$

3.4 Algorithme de Straßen

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{bmatrix}$$

$S_1 = (B - D)(G + H)$ (etc)
_ cf https://moodle.insa-rouen.fr/mod/resource/view.php?id=13021

$T(n) = 7\ T(\frac{n}{2}) + d.n^2$ multiplication de matrice d'ordre n
$T(n) = O(n^{log_2(7)})$ avec $log_2(7) \sim 2.89$

3.5 Exercice : Problème du sac à dos

$$\text{Problème du sac à dos : } \begin{cases} Max \sum_{i=1}^{n} v_i x_i \\ \sum_{i=1}^{n} w_i x_i \leq B \\ x_i \in \{0, 1\} \end{cases}$$

(v : valeur de l'objet. w : son poids)

1. Proposer un exemple tel qu'un algorithme glouton ne fonctionne pas.
   - Algo glouton : Pour chaque objet pris dans l'ordre, si on peut (contrainte de capacité du sac), on ajoute l'objet
2. Ecrire un algo générant toutes les solutions possibles. Donner sa complexité

Correction

$$(P_{ij}) \quad \begin{cases} Max \sum_{k=1}^{i} v_k x_k \\ \sum_{k=1}^{i} w_k x_k \leq j \\ x_k \in \{0, 1\} \qquad \forall k \in [|1, i|] \end{cases}$$

Soit P_{ij} le gain maximum pour le problème réduit aux i premiers objets avec une capacité du sac j.

$P_{ij} =?$
$$P_{1j} = \begin{cases} 0 \quad si\ w_1 > j \\ v_1 \quad si\ w_1 \leq j \end{cases}$$

$P_{i-1,j}$ par rapport à P_{ij} ?

Si la meilleure solution pour (P_{ij}) ne contient pas l'objet i alors $P_{ij} = P_{i-1,j}$.
Sinon cette meilleure solution contient l'objet i alors $P_{ij} = v_i + P_{i-1,j-w_i}$ **si** $j - w_i > 0$.

D'où $\boxed{P_{ij} = Max(P_{i-1,j}\ ; v_i + P_{i-1,j-w_i})}$.

A noter que $P_{ij} = 0 \quad si\ i = 0 \quad ou\ j = 0$.

Algorithme (non polynomial) : $(\sim O(nB))$
Pseudo-polynomial car B non borné.

```
Pour i=1 à n     P[i,0] <- 0;
Pour i=1 à B     P[0,i] <- 0;

Pour i=1 à n
    Pour j=1 à B
        P[i,j] <- P[i-1,j];
        Si j >= w_i alors
            Si P[i-1,j-w_i] + v_i > P[i-1,j] alors
                P[i,j] <- P[i-1,j-w_i] + v_i;
            FinSi
        FinSi
    FinPour
FinPour
```

Algo/td1_algo4.algo

3.6 Exemple avec données

E1 :

$$\begin{array}{c} i \\ v_i \\ w_i \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 6 & 18 & 22 & 28 \\ 1 & 2 & 5 & 6 & 5 \end{pmatrix} \end{array}$$

Déterminer la meilleure solution pour 5 objets ($n = 5$) et $B = 11$ (On applique l'algo juste au dessus). $P[5, 11] = ?$

0	$\boxed{0}$	0	0	0	0	0	0	0	0	0	0	0
1	$\boxed{0}$	1	1	1	1	1	1	1	1	1	1	1
2	$\boxed{0}$	1	6	7	7	7	7	7	7	7	7	7
3	$\boxed{0}$	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	$\boxed{22}$	24	28	29	29	40
5	0	1	6	7	7	28	29	34	35	35	46	$\boxed{50}$

Le chemin parcouru correspond aux nombres encadrés.

Fichier de données pour l'algorithme :

```
n B

v1 w1
v2 w2
..  ..
vn wn
```

Algo/td1_data.txt

Chapitre 4

Exercice

Exercice 4.1 $T = (V, E)$ ***arbre*** *de degré borné par 3.* $\quad n = |V|$. *Montrer qu'il existe une arête dont la suppression donne 2 arbres ayant au plus* $\frac{2n+1}{3}$ *sommets.*
Donner un algorithme en temps linéaire.

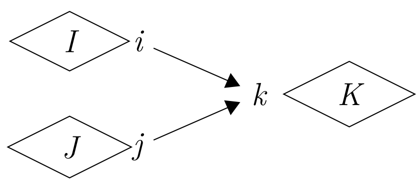
Algorithme linéaire à trouver $\Rightarrow$ idée du parcours de graphe. Et mettre à jour les tailles des composantes connexes si on retire l'arête. *TODO : A programmer*.

Parcourir l'arbre et déterminer récursivement pour chaque arête ij la taille de chaque composante obtenue en supprimant l'arête ij : $T(i)$ et $T(j)$.

Pour chaque arête ij, on oriente ij dans la direction de la plus petite composante :
Si $T(i) < T(j) \quad i \rightarrow j$
Si $T(i) > T(j) \quad i \leftarrow j$
Si $T(i) = T(j)$, on a le choix

Obtient-on une arborescence ? Il faut vérifier $\forall i \quad d^-(i) \leq 1$ ie au plus un arc entrant par sommet.

Démonstration 4.1 *En effet, si* $i \rightarrow k \quad j \rightarrow k$
avec I*,* J*,* K *sous arbres orientés*



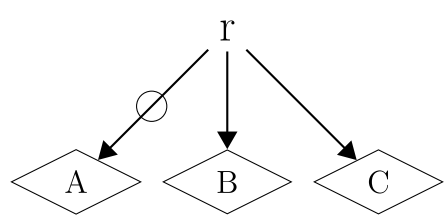
Par construction, $|I| \geq |J| + |K|$ *;* $|J| \geq |I| + |K| \quad \Rightarrow |K| = 0$ *Impossible.*

On a donc une arborescences : il existe alors une racine r. On prend l'arête reliant r à sa plus grande arborescence.

Soient A, B, C les sous arborescences de r

$|A| \geq |B| \geq |C|$
$|A| \leq \dfrac{n}{2} \leq \dfrac{2n+1}{3}$
$|B| \leq |A|$ et $|C| \leq |A|$ donc $|B| + |C| \leq 2\,|A|$

et $n - 1 = |A| + |B| + |C| \geq 3\dfrac{|B| + |C|}{2}$
d'où $|B| + |C| + 1 \leq \dfrac{2n+1}{3}$



Chapitre 5

Union-Find

Union (u, v) : Fusionne les ensembles contenant les éléments canoniques u et v
Find (u) : Renvoie un élément canonique de l'ensemble contenant u

Choix de la structure de données :

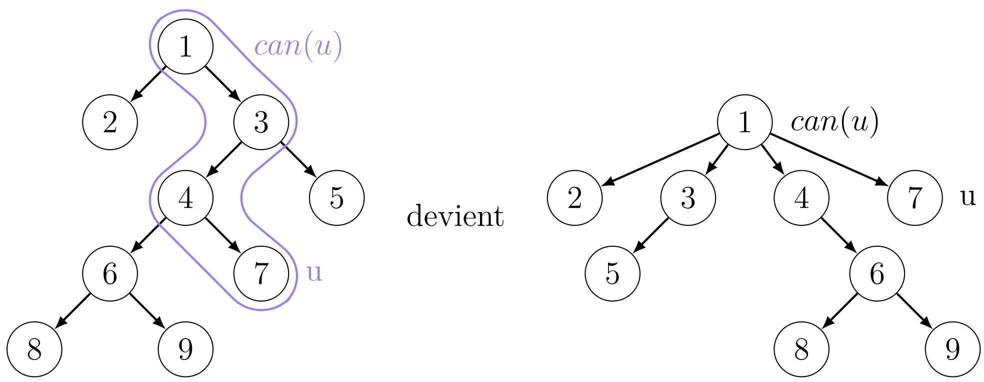
Chaque ensemble est représenté par une arborescence (par une liste d'éléments) ayant pour sommets les éléments de l'ensemble, la racine étant l'élément canonique.

Chaque élément u a un père (ou une mère selon votre degré de féminisme) dans l'arborescence : $p(u)$.

On utilise deux heuristiques pour accélérer les algorithmes $union$ et $find$. (pour l'efficacité, pas pour le fonctionnement de l'algo)

- **union** : Toujours prendre la racine de la plus petite arborescence comme enfant de la plus grande (On a une variable par sommet donnant la taille de la sous-arborescence)
- **find** : Quand on a trouvé $can(u)$ (l'élément canonique de u, ie la racine de la sous-arborescence), on parcourt à nouveau tous les sommets v du chemin de $can(u)$ à u. On appelle ce procédé **Compression de chemin**

Exemple :



Théorème 5.1 *n éléments et m séquences union find. $O((m+n)\alpha(n))$*
$\alpha(n)$: inverse de la fonction de Ackermann

5.1 Fonction de Ackermann

$A_0(x) = x + 1$
$A_{k+1}(x) = A_k^x(x) = A_k \circ \ldots \circ A_k(x)$ (x fois)

$A_k^0 =$ identité
$A_k^{i+1} = A_k \circ A_k^i$

Propriété 5.1 *A_k est croissante* $\quad x \leq y \Rightarrow A_k(x) \leq A_k(y)$

$A_0(x) = x + 1$
$A_1 = A_0^x = 2x$
$A_2(x) = A_1^x(x) = x\ 2^x \geq 2^x$
$A_3(x) = A_2^x(x) \geq 2^{2^{\cdot^{\cdot^{2}}}}$ (x fois) $= 2 \uparrow x$
$A_4(x) = A_3^x(x) \geq 2 \uparrow (2 \uparrow \ldots (2 \uparrow 2) \ldots) = 2 \uparrow\uparrow x$

Définition 5.1 *Fonction de Ackermann : $A(k) = A_k(x)$*
L'inverse de la fonction de Ackermann : $\alpha(n)$: plus petit k tel que $A(k) \geq n$

En pratique : $\alpha(n) \leq 4$

TODO : Programmer l'algorithme de Kruskal avec Union-Find (avec ou sans compression de chemin) – (Try with compression de chemin)

5.2 Plus longue sous-suite commune

Définition 5.2 *$Z = z_1 z_2 \ldots z_k$ est une sous-suite de $X = x_1 x_2 \ldots x_n$ s'il existe une suite d'indices croissants $i_1 \ldots i_k$ telle que : $\forall j = i \ldots k \quad x_{i_j} = z_j$*

Exemple 1 : Z = BCDB ; X = ABCBDAB ⇒ X = A **BC** B **D** A **B** (Z en gras)
Exemple 2 : X = ABCBDAB ; Y = BDCABA ⇒ Z = BCAB ou Z = BCBA conviennent

Notation : X_i le ième préfixe de X : $X_i = x_1 \ldots x_i$; $\quad X_0$ mot vide

Théorème 5.2 *Sous-structure optimale -*
Soient $X = x_1 \ldots x_n$ et $Y = y_1 \ldots y_m$ 2 mots et $Z = z_1 \ldots z_k$ une plus longue sous-suite commune (plssc) de X et Y alors :

1. *Si $x_n = y_m$ alors $z_k = x_n = y_m$ et z_{k-1} est 1 plus longue sous-suite de X_{n-1} et Y_{m-1}*
2. *Si $x_n \neq y_m$ alors $(z_k \neq x_n) \Rightarrow$ Z est 1 plus longue sous-suite commune de X_{n-1} et Y*
3. *Si $x_n \neq y_m$ alors $(z_k \neq y_m) \Rightarrow$ Z est 1 plus longue sous-suite commune de X et Y_{m-1}*

Démonstration

1. Si $(z_k \neq x_n)$, on ajoute $x_n = y_m$ à la fin de Z, on a toujours une sous-suite commune ⇒ contradiction (!) donc $z_k = x_n = y_m$
2. ...

$C[i,j]$: la longueur d'une plus longue sous-suite commune à X_i et Y_j.

$$c[i,j] = \begin{cases} 0 \ \ si\ i=0 \ \ ou\ j=0 \\ c[i-1,j-1]+1 \ \ si\ ((i>0) \ \ et\ (j>0) \ \ et\ (x_i = y_j)) \\ MAX(c[i-1,j], c[i,j-1] \ \ si\ ((i>0) \ \ et\ (j>0) \ \ et\ (x_i \neq y_j)) \end{cases}$$

L'algorithme pour trouver z ($\sim O(nm)$) ($b(i,j)$ représentant l'origine) :

```
n <- longueur (x)
m <- longueur (y)

Pour i=1 à n    c[i,0] <- 0;
Pour i=1 à m    c[0,i] <- 0;

Pour i=1 à n
    Pour j=1 à m
        Si (x_i = y_j) alors
            c[i,j] <- c[i-1,j-1] +1;
            b[i,j] <- 'Flèche en haut à gauche'
        Sinon
            Si c[i-1,j] >= c[i,j-1] alors
                c[i,j] <- c[i-1,j]
                b[i,j] <- 'Flèche en haut'
            Sinon
                c[i,j] <- c[i,j-1]
                b[i,j] <- 'Flèche à gauche'
            FinSi
        FinSi
    FinPour
FinPour
```

Algo/td2_algo1.algo

Exemple avec X= ABCBDAB et Y = BDCABA

0	0	0	0	0	0	0	0
1	0	$\uparrow 0$	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\nwarrow 1$
2	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$
3	0	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\uparrow 2$	$\uparrow 2$
4	0	$\nwarrow 1$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\leftarrow 3$
5	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\uparrow 3$
6	0	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 3$	$\nwarrow 4$
7	0	$\nwarrow 1$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\nwarrow 4$	$\uparrow 4$

Exercice 5.1 *Déterminer une plus longue sous-suite de 10010101 et 010110110.*

Longueur 6. Plus longue sous-suite commune : 100110

Exercice 5.2 *Montrer comment construire une plus longue sous-suite à partir de la table c et des suites X et Y en $O(n+m)$, sans utiliser "b".*

OK. Retrouver la sous-suite commune en $O(3(m+n))$ c'est la première solution qui vient sans le tableau de flèche : tester la valeur à gauche, en haut, et en diagonale en haut à gauche.

Exercice 5.3 *Montrer comment calculer une plus longue sous-suite avec seulement* $2 \times MIN(n,m)$ *cases dans la table c + une constante. Idem sans le* $\times 2$.

TODO - A mijoter

Exercice 5.4 *Donner un algorithme en* $O(n^2)$ *pour trouver une plus longue sous-suite croissante de nombres d'une suite de nombres.*

TODO

5.3 Algorithme de Prim

$G = (X, U)$ non orienté
$x_0 \in X$
$T = (X(T), U(T)) \subset G$
$T \leftarrow \{x_0, \varnothing\}$

```
TANT QU'ON PEUT
   Ajouter à T l'arête de plus petit poids reliant
    un sommet de X(T) un sommet de X\X(T)
FIN TQ
```

Au début : $X(T) = \{x_0\}$ $\quad U(T) = \varnothing$

$\forall i \in X$
$min(i)$: poids minimal des arêtes entre i et $X(T)$
$x(i)$: sommet de $X(T)$ tel que $i\ x(i)$ est de poids $min(i)$

Au début : $min(i) \leftarrow p(ix_0)$ si l'arête $i\ x_0$ existe, ∞ sinon.

A chaque itération :
    On ajoute i à $X(T)$ tel que $i = \underset{i \in X/X(T)}{\text{argmin}}\ min(i)$
    On met à jour $min(j)$ et $x(j)$ pour les $j \in X/X(T)$
        Si $p(ij) < min(j)$ Alors
            $min(j) \leftarrow p(ij)$
            $x(j) \leftarrow i$
        FinSi

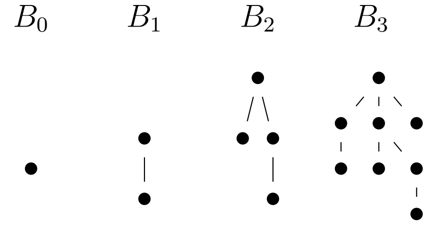
Chapitre 6

Tas binomiaux - Binomial heap

Jean Vuillemin - 1979

6.1 Définitions & Propriétés

Définition 6.1 *Un arbre binomial est défini récursivement.*
Pour $i \geq 1$, B_i est formé d'une racine i et de i enfants : $B_0 \dots B_{i-1}$



Remarque 6.1 $|B_i| = 2^i$
Nombre de sommets de niveau k de B_n : $C(n, k)$

Définition 6.2 *Un arbre binomial est ordonné si chaque sous-arborescence a son minimum à sa racine (ie : Chaque élément est plus grand que son père)*

Définition 6.3 *Un tas binomial est un ensemble d'arbres binomiaux ordonnés, il ne peut y avoir au plus qu'un B_i par entier naturel i*

Propriété 6.1 *Une valeur minimale est à la racine d'un des arbres binomiaux du tas binomial*

Propriété 6.2 *Un tas binomial de n sommets contient au plus $log(n+1)$ arbres binomiaux*

6.2 Opérations sur les tas binomiaux

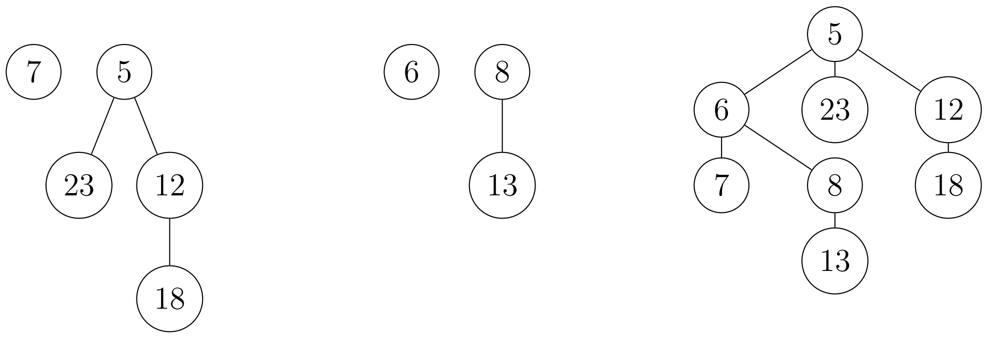
- création(i) : créé un tas contenant l'élément i $\sim O(1)$
- trouverMin(h) : renvoie le plus petit élément de h (h pour *heap*) $\sim O(log(n))$ ou $\sim O(1)$ si on a un pointeur vers le minimum
- insérer(h, i) : ajoute l'élément i au tas h $\sim O(log(n))$
- effacerMin(h) : efface l'élément de plus petite valeur de h $\sim O(log(n))$

- fusionner(h_1, h_2) : à partir des tas h_1 et h_2, donne un nouveau tas contenant tous les éléments de h_1 et h_2 $\quad \sim O(log(n))$
- décrémenter(h, i, δ) : diminuer de δ la valeur de l'élément i du tas h
- effacer(h, i) : enlever l'élément i du tas h

Remarque 6.2 *insérer(h, i) = fusionner(h, création(i))*

Exemple de fusion

Tas h_1 Tas h_2 Fusion(h_1,h_2)



Fusion de deux tas h_1 et h_2 :

On procède par i croissant.
Pour chaque i

- Si h_1 et h_2 n'ont pas de B_i : On forme un B_i en cas de retenue
- Si h_1 et h_2 ont tous les 2 un B_i : On forme un B_i et les 2 B_i de h_1 et h_2 forment une retenue B_{i+1}
- Si h_1 ou h_2 a un B_i mais pas l'autre : On le garde sauf s'il y avait une retenue auquel cas on forme un B_{i+1} comme retenue

Complexité amortie = "en moyenne sur un grand nombre d'éxécutions"
Quelle est la complexité amortie des opérations sur les tas binomiaux ?

Chapitre 7

Métaheuristique

7.1 Définitions

Définition 7.1 *Heuristique -*
En grec : Bonne direction - Algorithme approché sans garantie de performance

Définition 7.2 *Meta -*
Du grec : Au delà - Applicable à une large gamme de problèmes différents

Si besoin : http://fr.wikipedia.org/wiki/Métaheuristique

Deux types d'algorithmes approchés
- Par Constructions : Algorithme Glouton (ex : Kruskal)
- Par transformations

7.2 Recherche locale

Soit V. $x \in S \mapsto V(x) \subset S$.
Chaque élément de $V(x)$ est accessible à partir de x par une transformation "élémentaire".
$x =$ vecteur indicateur des sommets. **Exemple** : Partitionnement de graphe

$$t_i \ : \ x \mapsto t_i(x) = y \qquad \left\{ \begin{array}{rclcl} y_i & = & 1 - x_i & \textit{pour} & i = j \\ y_j & = & x_j & \textit{pour} & j \neq i \end{array} \right.$$

Recherche locale :
- Méthode itérative
- On cherche x^{k+1} dans $V(x^k)$

Stratégie du meilleur voisin

$k \leftarrow 0$
x^0 solution initiale
Répéter
  $x^{k+1} \leftarrow \underset{x \in V(x^k)}{argmin}\ f(x)$
  $k \leftarrow k + 1$
TantQue $(f(x^k) < f(x^{k-1}))$ /* ou $k > K$ */

Au mieux, on obtient un optimum local.

7.3 Recherche à profondeur variable

$\bar{x}$: Solution initiale
L : Liste de transformations

REPETER
    $x^0 \leftarrow \bar{x}$
    $\hat{x} \leftarrow \bar{x}$
    $L \leftarrow \varnothing$
    bool $\leftarrow$ Vrai
    POUR $j = 0$ à $n - 1$
        $x^{j+1} \leftarrow \underset{x \in V_L(x^j) \backslash x^j}{Argmin} \; f(x)$
        SI $f(x^{j+1}) < f(\hat{x})$ ALORS $\hat{x} \leftarrow x^{j+1}$ FINSI
        $L \leftarrow L \cup \{t(x^j, x^{j+1})\}$
    FIN POUR
    SI $(\hat{x} < \bar{x})$ ALORS $\bar{x} \leftarrow \hat{x}$ FINSI
JUSQU'A (Test Arrêt)

- $V_L(x)$: Voisinage des solutions accessibles à partir de x par les transformations élémentaires sauf celles qui sont dans L.
- L : Liste "tabou"

7.4 Recuit simulé

(Métaheuristique stochastique - Algorithme de Métropolis)

$\bar{x} \leftarrow x^0$
$k \leftarrow 0$
TANT QUE (Condition d'arrêt)
    Choisir $y \in V(x^k)$ (Aléatoirement)
    $p \leftarrow Min\{1, exp(-\dfrac{f(y) - f(x^k)}{\theta_k})\}$
    $x^{k+1} \leftarrow \begin{cases} y & \text{avec proba} \quad p \\ x^k & \text{avec proba} \quad 1 - p \end{cases}$
    SI $f(x^{k+1}) < f(\bar{x})$ ALORS $\bar{x} \leftarrow x^{k+1}$ FINSI
    $k \leftarrow k + 1$
FIN TantQue

p : Proba qu'on fasse la transformation

7.5 Algorithmes génétiques

Algorithme de population

$P_0 = \{x^1, x^2, \ldots, x^m\}$ *Population*
$k \leftarrow 0$
TANT QUE (Condition d'arrêt)

Sélection

Obtenir $(y^1, z^1), \ldots, (y^p, z^p)$, des paires de solutions par opérateur de solution

$P_{k+1} \leftarrow \mathcal{S}(P_k)$ (peut-être $\varnothing$ ou P_k)

POUR $j = 1$ à p

Croissement : $x^j \leftarrow \chi(y^j, z^j)$

Mutation : $x^j \leftarrow \mu(x^j)$

$P_{k+1} \leftarrow P_{k+1} \cup \{x^j\}$

FIN POUR

FIN TantQue

Chapitre 8

Exercice

8.1 Multiplication de matrices rectangulaires

$A_1, \ldots, A_n$ matrices rectangulaires
A_k est de taille $p_k \times q_k$
$A_k A_{k+1} \quad q_k = p_{k+1}$

1. Nombre d'opérations élémentaires ? (addition & multiplication)
   $\underbrace{C}_{p\times r} = \underbrace{A}_{p\times q} \times \underbrace{B}_{q\times r}$
   $c_{ij} = \sum_{k=1}^{q} a_{ik}\, b_{kj} \Rightarrow \quad \sim O(pqr)$

2. Exemple :
   $A_1 = 10 \times 100$
   $A_2 = 100 \times 5$
   $A_3 = 5 \times 50$

   Donner le nombre de multiplications scalaires pour effectuer :
   $((A_1\ A_2)\ A_3) \rightarrow (10 \times 100 \times 5) + (10 \times 5 \times 50) \quad = 7500$
   $(A_1\ (A_2\ A_3)) \rightarrow (100 \times 5 \times 50) + (10 \times 100 \times 50) = 75\,000$

3. Soit $P(n)$ le nombre de façons de parenthéser une suite de n matrices.
   Montrer que $P(n) = \begin{cases} 1 & si \quad n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & si \quad n \geq 2 \end{cases}$

   $P(1) = 1 \quad P(2) = 1 \quad P(3) = 2 \quad P(4) = 5 \quad P(5) = 14 \quad P(6) = 42$
   $(A_1 A_2 \ldots A_k)\ (A_{k+1} \ldots A_n) \rightarrow \sum_{k=1}^{n-1} P(k)P(n-k)$

4. Proposer un algorithme pour trouver la meilleure façon de placer les parenthèses et donner sa complexité.

5. Application numérique :
   $A_1 = 30{\times}35 \quad A_2 = 35{\times}15 \quad A_3 = 15{\times}\ 5 \quad A_4 =\ 5{\times}10 \quad A_5 = 10{\times}20 \quad A_6 = 20{\times}25$

TODO : Coder l'algorithme pour le parenthésage + Donner sa complexité

a_{ij} le plus petit nombre d'opérations élémentaires pour calculer $A_i \dots A_j$ avec A_i matrice de taille $p_i \times p_{i+1}$

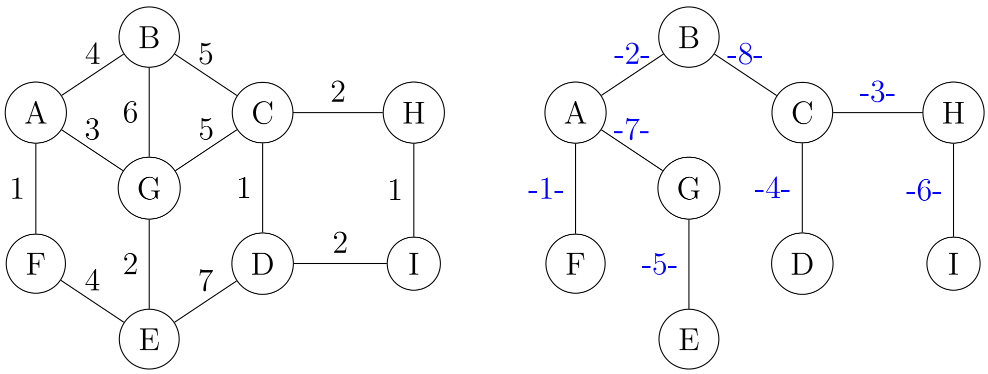
$a_{ij} = \min\limits_{k=1 \dots j-1} (m_{ik} + m_{k+1,j} + p_i p_{k+1} p_{j+1})$
$m_{ii} = 0$
$m_{i,i+1} = p_i p_{i+1} p_{i+2}$

8.2 Algorithme de Sollin

G graphe non orienté
$\mathcal{A}$: graphe partiel de G sans arêtes

TANT QUE il y a plus d'une composante connexe
Soit C une composante de $\mathcal{A}$ ajouter à $\mathcal{A}$ une arête de poids minimal reliant C à une autre composante connexe de $\mathcal{A}$
FIN TantQue
Résultat : $\mathcal{A}$

A programmer avec union-find + Paralléliser



8.2.1 L'algorithme

```
Début
    Pour i = 1 à
        X_i = {i}
    FinPour
    T <- null
    TantQue |T| < n-1 Faire
        {* Initialisation *}
        Pour chaque arbre i Faire
            plusProche(i) <- +inf
        FinPour
        Pour chaque arc (vw) Faire
            Si FIND(v) != FIND(w) Alors
```

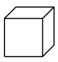
```
            Si poids(vw) < plusProche(FIND(v)) Alors
               plusProche(FIND(v)) <- poids(vw)
               arete(FIND(v)) <- vw
            FinSi
         Finsi
      FinPour
      Pour chaque arbre i Faire
         vw <- arete(i)
         Si FIND(v) != FIND(w) Alors
            T <- T U {vw}
            UNION(FIND(v), FIND(w))
         FinSi
      FinPour
   FinTantQue
Fin
```

Algo/td5_algo1.algo

Définition 8.1 *Ressource critique –*
Ressource partagée mais dont la modification peut entraîner un disfonctionnement du code dans une architecture parallèle.

Définition 8.2 *Section critique –*
Partie du code commune à plusieurs processus dont l'exécution simultanée peut provoquer un disfonctionnement.

8.3 Cubes / pyramides

Soit n cubes élémentaires. 
On souhaite les ranger uniquement en 2 types de tas :

1. en cube : $(k \times k \times k)$ exemple $3 \times 3 \times 3$
2. en pyramide : 1 puis 4 puis 9 ...
   Nombre de cubes dans une pyramide de k étages : $\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$

On souhaite connaître le nombre minimum de tas.

Ranger les valeurs des tas possibles dans un tableau : `t[]`

$i \leftarrow 0$
TANT QUE $(i^3 \leq n)$
$t_{cube}[i] \leftarrow i^3$
$i++$
FTQ
tailleCube $\leftarrow$ i
m $i \leftarrow 0$
TANT QUE $(\frac{i(i+1)(2i+1)}{6} \leq n)$
$t_{pyra}[i] \leftarrow \frac{i(i+1)(2i+1)}{6}$
$i++$

FTQ
taillePyra $\leftarrow$ i
Construite t à partir de t_{cube} et t_{pyra}

$p[i]$: le nombre minimum de tas pour i entier élémentaire. $p[0] = 0$

$p[k] \leq t[1] + p[k-1]$ with $t[1] = 1$
$p[k] \leq 1 + p[k - t[2]]$ cas où on prend un tas de taille $t[2]$ et la meilleure solution pour ranger $(k - t[2])$ cubes élémentaires.
$p[k] \leq 1 + p[k - t[3]]$ cas où on prend un tas de taille $t[3]$ et la meilleure solution pour ranger $(k - t[3])$ cubes élémentaires.
$\vdots$
$p[k] \leq 1 + p[k - t[l]]$ avec l le plus grand indice tel que $t[l] \leq k$
$\sim O(n^{\frac{4}{3}})$
$\Rightarrow p[k] = 1 + \min\limits_{1 \leq i \leq l} p[k - t[i]]$

$p[0] = 0$
$p[1] = 1$
Pour $i = 2$ à n faire
    $k \leftarrow 1$
    $p[i] \leftarrow p[i-1] + 1$
    Tant Que $t[k] \leq i$
        Si $p[i - t[k]] + 1 < p[i]$ Alors
            $p[i] \leftarrow p[i - t[k]] + 1$
        Finsi
    FinTantque
FinPour
$\sim O(n^{\frac{1}{3}})$ (complexité du TQ)

Chapitre 9

Algorithmique parallèle

9.1 Définitions

Définition 9.1 *Système parallèle -*
Un système implanté sur un ordinateur avec 2 ou plusieurs processus qui fonctionne concurremment

Définition 9.2 *Processus -*
Un processus est une partie d'un programme qui s'exécute séquentiellement mais qui peut s'exécuter en parallèle avec d'autres processus du même programme

Définition 9.3 *Multiprogrammation -*
L'exécution en simultanée de processus concurrent sur un seul processeur

Définition 9.4 *Multiprocessing -*
Exécution de processus concurrents. Des processus distincts peuvent tous accéder à une mémoire partagée

Définition 9.5 *Algorithmique distribuée -*
exécution de processus concurrent sur des processeurs dictincts (différents) qui communiquent par un mécanisme d'échange de messages. Chaque processeur possède sa mémoire locale mais il n'y a pas de mémoire partagée

9.2 Deux grandes approches

9.2.1 Vectorisation (pipeline)

Le traitement se fait en plusieurs phases *chaînées*

Exemple : Addition de 2 vecteurs A et B de nombres flottants
$C = A + B$
$c_i = a_i + b_i \quad 1 \leq i \leq n$

En machine $x = \alpha \times 2^p, \quad y = \beta \times 2^q$

L'addition $a_i + b_i$ peut être partagée en 7 opérations élémentaires :

1. Contrôle des signes

2. Comparaison des exposants
3. Alignement des virgules
4. Addition des mantisses
5. Calcul de la partie exposant
6. Normalisation
7. Calcul et recherche de l'adresse du résultat

9.2.2 Multi-tâches

Plusieurs tâches indépendantes.
Exemple : Multiplication de matrices

- Séquentiel :

```
Pour i = 1 à 100
    Pour j = 1 à 100
        c [ i j ] < 0 ;
        Pour k = 1 à 100
            c [ i j ] < c [ i j ] + a [ i k ] + b [ k j ]
        FinPour
    FinPour
FinPour
```

Algo/td4_algo1.algo

- Sur 4 processeurs :

```
n . proc = 3
l = proc_id ( ) 1
Pour i = 25 l +1 à 2 5 ( l +1)
    Pour j = 1 à 100
        c [ i j ] < 0 ;
        Pour k = 1 à 100
            c [ i j ] < c [ i j ] + a [ i k ] + b [ k j ]
        FinPour
    FinPour
FinPour
```

Algo/td4_algo2.algo

9.3 Mesure des performances

Définition 9.6 *Gain* - $\dfrac{T(1)}{T(p)}$ *avec* $T(i)$ = *temps de résolution pour i processeurs*

Définition 9.7 *Efficacité* - $\dfrac{Gain}{p}$

Théorème 9.1 *Loi de Amdahl* - $S = \dfrac{1}{(1-f)+\frac{f}{p}}$
f : fraction du code qui est parallélisé
p : nombre de processeurs

Exemple : Changer les 4 pneus d'une voiture

Avec 1 réparateur.
• Description du travail - 10 min
• Sortir les pneus neufs - 10 min
• Mettre la voiture sur l'élévateur - 10 min
• Changer les 4 pneus - 60 min (15 min / pneu)
• Descendre la voiture - 10 min
• Jeter les vieux pneus - 10 min
• Facturer - 10 min
Total : 2 heures.

Avec 4 réparateurs.
• Description du travail - 10 min
• Sortir les pneus neufs & Mettre la voiture sur l'élévateur - 10 min
• Changer les 4 pneus - 15 min (1 réparateur / pneu)
• Descendre la voiture & Jeter les vieux pneus & Facturer - 10 min
Total : 45 min !

$$\text{Gain} = \frac{120}{45} = 2,66666\ldots \qquad \text{Efficacité} = \frac{2}{3} = 0,66666\ldots$$

9.4 Compléxité

9.4.1 Architecture

Définition 9.8 *PRAM - Parallel Random Access Machine*
Ensemble de processeurs avec mémoire commune partagée.
Chaque processeur peut avoir une mémoire locale.
Un accès à la mémoire = une unité de temps.

• CRCW : Concurrent Read, Concurrent Write : chaque processeur peut lire et écrire n'importe où dans la mémoire à tout moment.
• CREW : Concurrent Read, Exclusive Write : chaque processeur peut lire n'importe quel endroit de la mémoire à tout instant, mais aucune écriture simultanée de deux processeur à un même endroit n'est possible.
• ERCW : Exclusive read, Concurrent Write : never considered. (Aucun sens)
• EREW : Exclusive read, Exclusive Write : chaque processeur ne peut lire ou écrire à un endroit de la mémoire que si aucun autre processeur n'y accède à ce moment-là.

Dans les architectures des machines vectorielles, on a des machines SIMD (*Simple Intruction on Multiple Data*) ou encore MIMD (*Multiple Instructions on Multiple Data*).

Circuits booléens / arithmétiques (ie un graphe orienté sans circuits)
• Noeuds d'entrée (input) - Noeuds de sortie (output)
• Opérateurs de bases sur les bits associés aux noeuds
• Nombre de noeuds ≈ nombre de processeurs dans un PRAM
• **Profondeur** : Longueur du plus long chemin (en terme d'arcs) d'un sommet de l'input vers un sommet de l'output ≈ temps de calcul

9.4.2 Nick's Class

Définition 9.9 *Nick's Class - NC - Nick Pippenger*
Classe des problèmes efficacement parallélisableS (← définition pour les ASI)
NC est l'ensemble des problèmes pouvant être résolus sur un PRAM en temps polylogarithmique ($log^{O(1)}(n)$) en utilisant un nombre polynomial de processeurs ($n^{O(1)}$)

A-t-on $NC = \mathcal{P}$? Problème ouvert

9.4.3 Multiplication matricielle

2 matrices carrées A, B d'ordre n. n^3 processeurs. Temps : $1 + log(n)$

$C = A \times B \Leftrightarrow c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}$

- Calculer en parallèle les produits $a_{ik}\, b_{kj} \quad \forall i, j, k$ (1 pas)
- Allouer n processeurs à chaque paire (i, j) et calculer la somme en $log(n)$ pas

9.4.4 Calcul parallèle des préfixes

$x_0, \ldots, x_{n-1}$
Opération binaire $\otimes$ associative (pas forcément commutative)
$y_i = x_n \otimes \ldots \otimes x_i$

On construit le circuit arithmétique suivant :
n entrées → reçoivent les x_i
n sorties → y_i

Chaque processeur p transmet sa forme au processeur $p+1$ qui opère sur deux données.
Idem de p à $p2$ / de p à $p+4 \rightarrow log(n)$

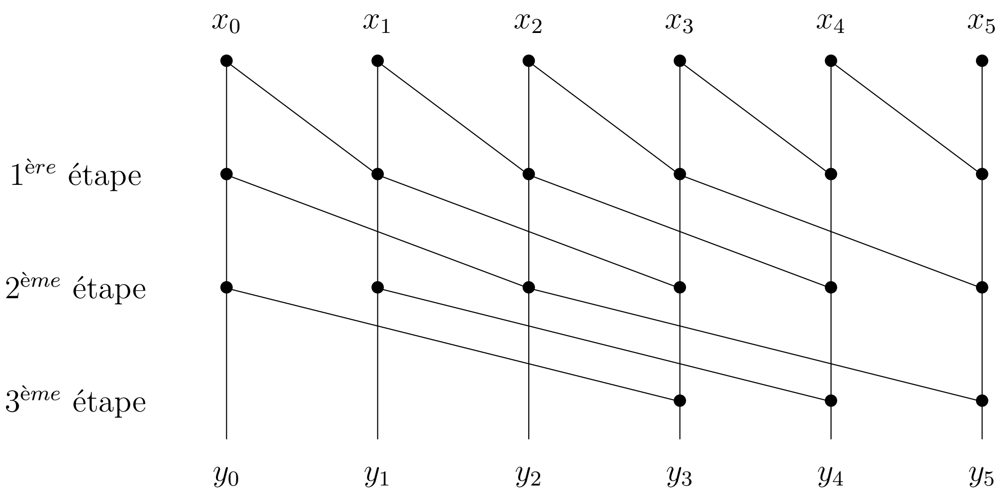


Figure 9.1 – Calcul en parallèle des préfixes (tous)

9.5 Arithmétique entière dans $\mathcal{NC}$

9.5.1 Addition

n digits (bits) : en $log(n)$ avec n processeurs

- Calcul de la chaîne des retenues avec préfixe parallèle
- Calcul de la somme : XOR des 2 opérandes et des retenues

Chaîne de retenue : r
Bit de poids faible : 0
Si les $i^{èmes}$ bits des opérandes sont 0 : $r_{i+1} = 0$
Si les $i^{èmes}$ bits des opérandes sont 1 : $r_{i+1} = 1$
Si les $i^{èmes}$ bits des opérandes sont 0 et 1 : $r_{i+1} = r_i$

$\{0, 1, p\}$ muni de l'opération associative « . » $\quad 0.x = 0 \quad 1.x = 1 \quad p.x = x$
p pour propagation (de la retenue).

.	0	1	p
0	0	0	0
1	1	1	1
p	0	1	p

Exemple *(Calcul parallèle des préfixes avec la loi « . »)*

a		1	0	0	1	0	1	0	1	1	1	0	1	0	1	1
b		1	1	0	1	0	1	0	0	1	0	1	0	0	0	1
$r =$	1	p	0	1	0	1	0	p	1	p	p	p	0	p	1	0
retenue	1	0	0	1	0	1	0	1	1	0	0	0	0	1	1	0
a+b	1	0	1	1	0	1	0	1	0	0	1	1	1	1	0	0

Pour trouver le r : On met un 0 pour le 1^{er} ; ensuite si on a $(1,1) \Rightarrow 1$, si on a $(1,0) - (0,1) \Rightarrow p$, et si on a $(0,0) \Rightarrow 0$ (pour la retenue suivante).
Pour le calcul de r, $\sim O(1)$ avec n processeurs.

Pour le calcul de la vraie retenue, on reprend le cablage de la figure 9.1, page 26, mais en partant de la gauche.

9.5.2 Multiplication

2 nombres à n chiffres.
Façon scolaire : n sommes $\sim O(log^2(n))$ avec $O(n^2)$ processeurs.

$$\begin{array}{cccccccccc}
 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
+ & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
+ & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
\hline
 & & & & & & & & 1 & 0 \\
 & & & & & & & 0 & 1 & \\
 & & & & & & 1 & 1 & & \\
 & & & & & 1 & 0 & & & \\
 & . & . & . & . & . & & & & \\
\hline
\end{array}$$

En temps constant : Chaque processeur additionne 3 chiffres et donne comme résultat un nombre à 2 chiffres.

On obtient 2 nombres de n chiffres dont l'addition donne le même résultat que l'addition des 3 nombres de départ *(en blue + purple)*.

On groupe les nombres par 3 à l'étape k : $(\frac{2}{3})^k + n$ nombres à additionner.
(Quitte à arrondir n à la puissance de 3 supérieure)

$\sim O(log(n))$ étapes
Ensuite on additionne les 2 nombres obtenus en $O(log(n))$ avec $O(n)$ processeurs.

Complexité de la multiplication : $log(n)$ avec $O(n)$ processeurs.

9.5.3 Division

$\frac{s}{t}$: $\quad s = qt + r \quad 0 \le r < t$

Méthode de Newton : $x_{i+1} = x_i - \dfrac{f(x_i)}{f'(x_i)}$

Calcul de $\frac{1}{t}$:

$f(x) = t - \frac{1}{x}$. On cherche x tel que $f(x) = 0$.
$f'(x) = \frac{1}{x^2} \quad x_{i+1} = 2x_i - tx_i^2$
$x_0 = 2^a$ tel que $x_0 \in]\frac{1}{2t}, \frac{1}{t}]$ $\quad (\frac{1}{x} \in [t, 2t[)$

Chaque opération se fait en $\sim O(log(n))$ avec $O(n)$ processeurs.
L'erreur initiale : $\frac{1}{2t}$ (Largeur de l'intervalle de départ)
On montre que la convergence est quadratique.

$0 \le 1 - tx_0 < \frac{1}{2}$
$i \ge 0 \qquad 1 - tx_{i+1} = (1 - tx_i)^2$

Par récurrence : $(1 - tx_i) = (1 - tx_0)^{2^i}$
D'où $\frac{1}{t} - x_i < \frac{1}{2^{2^i} t}$
$\frac{1}{2t} < x_0 \le x_1 \le \ldots \le \frac{1}{t}$
Le nombre d'itérations est $k = [\log \log |\frac{s}{t}|]$
$x_k \le \frac{1}{t}$ et $1 - tx_k < \frac{t}{s} \quad$ (Rq : $\frac{1}{t} - x_k \le \frac{1}{s} \Rightarrow s \times erreur < 1$)
Donc $0 \le \frac{s}{t} - sx_k < 1$
La partie entière de $\frac{s}{t}$ est soit $\lfloor sx_k \rfloor$ soit $\lceil sx_k \rceil$. On trouve le reste en soustrayant.

$log(\frac{s}{t})$ est de l'ordre de n

$\log\log|\frac{s}{t}| = O(log(n))$

Donc la complexité de la division par cette méthode est en $log^2(n)$ avec $O(n)$ processeurs.

Les opérations de base (+, −, ×, division) sont en temps polynomial avec $O(n)$ processeurs.

9.6 L'algèbre linéaire dans $\mathcal{NC}$

9.6.1 Opérations « élémentaires »

- **Produit scalaire** : $< a, b >$ où a, b sont des vecteurs, $a = (a_1, \ldots, a_n)$, $b = (b_1, \ldots, b_n)$
  En $O(log(n))$ étapes arithmétiques avec $O(n)$ processeurs.
  $\hookrightarrow$ Calcul parallèle des a_ib_i
  $\hookrightarrow$ Somme des produits de façon arborescente
- **Multiplication de matrices** : $A(m \times n)$ et $B(n \times p)$
  Calcul de AB : $O(log(n))$ avec $O(mnp)$ processeurs
  mp coefficients calculés par produit scalaire
- **Puissance de** A : $A^k \ \ (k = 1, \ldots, n)$ avec A matrice $n \times n$
  $O(log^2(n))$ avec $O(n^4)$ processeurs : calcul des préfixes paralèlles de $(A, A, \ldots, A)$ avec comme opération associative la multiplication de matrices

9.6.2 Inversion de matrices triangulaires inférieures

A matrice $n \times n$

$$A = \begin{bmatrix} B & 0 \\ C & D \end{bmatrix} \begin{matrix} \}\frac{n}{2} \\ \}\frac{n}{2} \end{matrix}$$

On calcule B^{-1} et D^{-1} récursivement en parallèle et $A^{-1} = \begin{bmatrix} B^{-1} & 0 \\ -D^{-1}CB^{-1} & D^{-1} \end{bmatrix}$.

Temps de calcul parallèle $\quad T(n) = T(\frac{n}{2}) + 2M(\frac{n}{2})$
$T(\frac{n}{2})$: Temps pour inverser B et D en parallèle
$2M(\frac{n}{2})$: Temps pour calculer $-D^{-1}CB^{-1}$
Avec $O(n^3)$ processeurs, on a $M(n) = O(log^2(n))$, d'où $T(n) = O(log^2(n))$

9.6.3 Récurrences linéaires

a_{ij} et c_i donnés

$$\begin{aligned}
x_1 &= c_1 \\
x_2 &= a_{21}x_1 + c_2 \\
x_3 &= a_{31}x_1 + a_{22}x_2 + c_3 \\
&\vdots \\
x_n &= a_{n1}x_1 + \cdots + a_{n,n-1}x_{n-1} + c_n
\end{aligned}$$

Exemple : Suite de Fibonacci
$F_0 = 1 \quad F_1 = 1 \quad F_{n+2} = F_{n+1} + F_n$
$c_1 = c_2 = 1 \quad c_i = 0$ pour $i \geq 3$
$a_{i,i-1} = a_{i,i-2} = 1$ pour $i \geq 3$
$a_{ij} = 0$ sinon

Notons $a_{ij} = 0$ pour $j \geq i \quad A = (a_{ij})_{1 \leq i,j \leq n} \quad x = (x_i)_{1 \leq i \leq n} \quad c = (c_i)_{1 \leq i \leq n}$

Le système équivaut à $Ax + c = x$. Soit $c = (I - A)x$
(Rq : $(I - A)$ triangulaire inférieure avec une diagonale de 1)

$x = (I - A)^{-1}c$ avec la méthode du 9.6.2 donc en $O(log^2(n))$ avec $O(n^2)$ processeurs.

9.6.4 Polynôme caractéristique

$$\begin{aligned} det\ (xI - A) &= x^n - s_1 x^{n-1} + s_2 x^{n-2} - \cdots \pm s_n \\ &= \prod_{i=1}^{n} (x_i - \lambda_i) \end{aligned}$$

$$\begin{aligned} s_1 &= tr(A) \\ &= \sum_{i=1}^{n} \lambda_i \quad \text{Calculable dans } \mathcal{NC}. \\ &= \sum_{i=1}^{n} a_{ii} \end{aligned}$$

Remarquons que $tr(A^m) = \sum_{i=1}^{n} \lambda_i^m$

$s_n = det(A) = \prod_{i=1}^{n} \lambda_i$

$$s_k = \sum_{1 \leq i_1 \leq i_2 \leq \cdots \leq i_k \leq n\ ,\ j \notin \{i_1, \ldots, i_k\}} \lambda_{i_1} \ldots \lambda_{i_k} \lambda_j^m$$

On a $f_k^0 = (n - k)s_k$

D'où
$$\begin{aligned} s_k tr(A^m) &= \left(\sum_{1 \leq i_1 \leq \cdots \leq i_k \leq n} \lambda_{i_1} \ldots \lambda_{i_k}\right)\left(\sum_{i=1}^{n} \lambda_i^m\right) \\ &= \sum_{1 \leq i_1 \leq \cdots \leq i_k \leq n\ ,\ j \notin \{i_1, \ldots, i_k\}} \lambda_{i_1} \ldots \lambda_{i_k} \lambda_j^m + \sum_{1 \leq i_1 \leq \cdots \leq i_k \leq n\ ,\ j \in \{i_1, \ldots, i_k\}} \lambda_{i_1} \ldots \lambda_{i_k} \lambda_j^m \\ &= f_k^m + f_{k-1}^{m+1} \end{aligned}$$

Donc $s_k tr(A^0) - s_{k-1} tr(A^1) + s_{k-2} tr(A^2) \cdots \pm s_1 tr(A^{k-1}) \pm tr(A^k)$
$$\begin{aligned} &= (f_k^0 + f_{k-1^1} - (f_{k-1}^1 + f_{k-1}^2) \cdots \pm (f_1^{k-1} + f_0^k) \pm f_0^k \\ &= f_k^0 \\ &= (n - k)s_k \end{aligned}$$

$$\boxed{s_k = \frac{1}{k}(s_{k-1} tr(A^1) - s_{k-2} tr(A^2) \cdots \pm tr(A^k))}$$

$tr(A^m)$ calculable dans $\mathcal{NC}$
On fait le calcul par récurrence en utilisant 9.6.3.

9.6.5 Inversion de matrice régulière : algo Csanky (1976)

Théorème 9.2 *Cayley-Hamilton –*
Toute matrice A vérifie son équation caractéristique :

$$A^n - s_1 A^{n-1} + s_2 A^{n-2} \cdots \pm s_{n-1} A \pm I = 0$$

On multiplie par A^{-1} et on arrange les termes : $A^{-1} = \frac{1}{s_n}(s_{n-1}I - s_{n-2}A \pm s_1 A^{n-1})$

On calcule les s_k comme au 9.6.4 (dans $\mathcal{NC}$)
Polynôme matriciel : $O(log\ n)$ avec $O(n^3)$ processeurs
A^{-1} : $O(log^2(n))$ avec $O(n^4)$ processeurs

9.7 Open MP

http://www.idris.fr/data/cours/parallel/openmp/OpenMP_cours.html
https://moodle.insa-rouen.fr/mod/resource/view.php?id=10895