Algorithmique et programmation

Claire Mathieu (hauts du DI 10), Jacques Stern, Zenthao Li (hauts du DI), Damien Vergnaud (-1 crypto)

Quatre notes pour l'évaluation : deux examens (début décembre, janvier), un devoir à la maison (distribué début octobre pour fin octobre), et un projet de programmation en C (à faire en novembre et décembre, soutenance fin décembre).

Livre de référence : CLRS (Cormen, Leisersan, Rivest, Stein), aka « le Cormen ».

2013-09-23

3 méthodes de conception d'algorithmes

1. Algorithmes gloutons
2. Diviser pour reigner
3. Programmaiton dynamique

Algorithmes gloutons

Un *problème* est donné par une spécification : une entrée et une sortie.

Étudions le problème de sélection d'activité. En entrée : n activités, où pour $1 \leqslant i \leqslant n$, $a_i = [s_i, f_i[$ (s_i est l'instant de début, f_i est l'instant de fin). On pose $S = \{a_i\}$. En sortie : un sous-ensemble A de S tel que :

- Toutes les activités de A sont compatibles, ie $\forall a_i, a_j \in A, [s_i, f_i[\cap [s_j, f_j[= \varnothing$ (si $i \neq j$)
- $|A|$ est maximal

Un *algorithme* est une façon de résoudre le problème. Un algorithme est *correct* s'il respecte la spécification du problème.

Pour notre problème, algorithme naif :

$A \leftarrow \varnothing$
Pour tout sous-ensemble T de S,
    si les activités de T sont compatibles entre elles,
    et si $|T| > |A|$
    alors faire $A \leftarrow T$.
Enfin, le résultat est A.

Cet algorithme est évidemment correct.

Défaut de l'algorithme : sa complexité en temps (il est trop lent). En effet :

- Pour tester si les activités de T sont compatibles, il faut un temps en $O(|T|^2) = O(n^2)$ ($n = |S|$).
- Pour tester tous les ensembles, on a un temps en $O(2^n)$.

Le temps total de l'algorithme est donc de l'ordre de $O(n^2\, 2^n)$: temps exponentiel (cette analyse est pessimiste, mais l'algorithme est néanmoins en $\Omega(2^n)$).

On dit qu'un algorithme est de complexité *raisonnable* si sa complexité est polynomiale en la taille de l'entrée.

Remarque. On s'intéresse toujours à la complexité asymptotique, ie quand $n \to +\infty$. De plus, c'est un ordre de grandeur (on abandonne le facteur constant, qui dépend des caractéristiques de la machine). On s'intéresse aussi à la complexité dans le pire des cas, car on veut un algorithme efficace pour toutes les entrées possibles.

Algorithme glouton pour ce même problème :

Trier les activités par ordre de $\begin{cases} \text{(incorrect) durée } \nearrow \\ \text{(?) fin } \nearrow \\ \text{lexicographique (fin } \nearrow\text{, début } \searrow) \\ \text{(incorrect) lexicographique (début, durée) } \nearrow \end{cases}$
$A \leftarrow \varnothing$
Pour chaque activité s de S,
  Si s est compatible avec A, alors $A \leftarrow A \cup \{s\}$.
Résultat : A

Cet algorithme n'est pas correct de base (cela dépend de l'ordre dans lequel les objets sont pris, ...)

Théorème. L'algorithme est correct lorsque l'on trie les activités par ordre lexicographique (fin croissante, début décroissant).

Preuve. Par récurrence sur n.

On a $f_1 \leqslant \cdots \leqslant f_n$.

On pose $S' = S \setminus \{s_k \,|\, a_k < f_1\}$.

On a $\{a_1\} \cup \mathrm{opt}(S') = \mathrm{opt}(S)$ car $\mathrm{opt}(S)$ contient au plus une activité à l'instant $f_1 - \varepsilon$, donc $|\mathrm{opt}(S)| \leqslant |\mathrm{opt}(S')| + 1$.

Complexité. $O(n^2)$ pour un algorithme naif, $O(n \log(n))$ pour un algorithme intelligent. (on mémorise l'instant de fin f de la dernière activité de A. s est compatible avec A si et seulement si $d_s \geqslant f$).

CLRS 16.1, 16.4

Définition. (S, I) est un matroïde si S est un ensemble fini non vide et I est une collection (un ensemble) de sous-ensemles de S telle que :

- I est non vide
- Si $A \in I$, alors tout sous-ensemble de A est dans I.
- Si $A, B \in I$ avec $|A| < |B|$, alors $\exists x \in B \setminus A \,|\, A \cup \{x\} \in I$.

Si $A \in I$, on dit que A est un *indépendant*.

Exemple. S est un ensemble de vecteurs, I est l'ensemble des familles libres de S.

Exemple. Soit G un graphe. S est l'ensemble des arêtes de G, $A \in I$ ssi A est acyclique. $\varnothing \in I$ donc $I \neq \varnothing$.

On munit notre graphe d'une fonction poids, ie d'une fonction $\omega : \begin{matrix} S \to \mathbb{R}_+^* \\ x \mapsto \omega_x > 0 \end{matrix}$, et on étudie l'algorithme suivant :

Entrée. matroïde (S, I) et poids ω.

Sortie. $A \in I$ maximisant le total des poids.

Ce problème peut se résoudre par un algorithme glouton :

Trier S par poids décroissants
$A \leftarrow \varnothing$

Pour chaque élément $x \in S$ dans l'ordre :
Si $A \cup \{x\} \in I$, alors $A \leftarrow A \cup \{x\}$.
Résultat : A

Théorème. cet algorithme est correct.

Preuve. On suppose que les arrêtes sont triées ($\omega(x_1) \geqslant \omega(x_2) \geqslant \cdots \geqslant \omega(x_n) > 0$).

Soit A_i l'ensemble A après le i-ème passage dans la boucle « pour ».

Invariant : $\forall i \geqslant 0, \exists A^*$ optimal qui contient A_i

Par récurrence sur i. (laissé en exercice au lecteur)

Diviser pour reigner (CLRS 4.2, 4.5)

Problème. Multiplication de matrices. En entrée, deux matrices carrées A et B de taille n. En sortie, la matrice $C = A\,B$.

Algorithme naïf : pour tout $1 \leqslant i, j \leqslant n$, $c_{i,j} \leftarrow \sum_k a_{i,k}\, b_{k,j}$. Complexité : $O(n^3)$.

Le précepte « diviser pour reigner » induit comme possibilité l'utilisation d'une décomposition par blocs. On décompose : $A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}$ et $B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$. On a alors :

$$A\,B = \begin{pmatrix} A_{1,1}\,B_{1,1} + A_{1,2}\,B_{2,1} & A_{1,1}\,B_{1,2} + A_{1,2}\,B_{2,2} \\ A_{2,1}\,B_{1,1} + A_{2,2}\,B_{2,1} & A_{2,1}\,B_{1,2} + A_{2,2}\,B_{2,2} \end{pmatrix}$$

Algorithme. (supposons $n = 2^p$; si ce n'est pas le cas, on agrandit la matrice jusqu'à la puissance de deux suivante)

Mult(A, B) :
Si $n = 1$, le résultat est trivial. Sinon :
Décomposer A et B en blocs.
Résultat : $A\,B = \ldots$ (faire les calculs avec appels récursifs).

Cet algorithme est bien correct... Pour la complexité : on écrit une récurrence (on note $C(n)$ le nombre d'opérations pour multiplier deux matrices carrées de taille n).

$$C(1) = 1$$

$$n > 1, C(n) = 8\,C\Big(\frac{n}{2}\Big) + \Theta(n^2)$$

Résolution. $C(n) = \Theta(n^2) + 8\,\Theta\big(\big(\frac{n}{2}\big)^2\big) + 8^2\,C\big(\frac{n}{2^2}\big) = n^2 + 8\big(\frac{n}{2}\big)^2 + 8^2\big(\frac{n}{2^2}\big)^2 + \cdots + 8^i\big(\frac{n}{2^i}\big)^2 + \cdots$ jusqu'à $\frac{n}{2^i} = 1$, ie $i = \log n$; $C(n) = \sum_{i \leqslant \log n} 8^i \frac{n^2}{2^{2i}} = n^2 \sum_{i \leqslant \log n} 2^i = n^3$.

Si on arrive à remplacer le 8 par un 7 dans la formule de récurrence, $C(n) = \sum_{i \leqslant \log n} 7^i \frac{n^2}{2^{2i}} = n^2\big(\frac{7}{4}\big)^{\log n} = n^{\log_2 7} \ll n^3$.

On arrive à faire ça en calculant les 7 produits suivants (algorithme de Strassen) :

$$\begin{aligned}
P_1 &= A_{1,1}\,(B_{1,2} - B_{2,2}) \\
P_2 &= (A_{1,1} + A_{1,2})\,B_{2,2} \\
P_3 &= (A_{2,1} + A_{2,2})\,B_{1,1} \\
P_4 &= A_{2,2}\,(B_{2,1} - B_{1,1}) \\
P_5 &= (A_{1,1} + A_{2,2})\,(B_{1,1} + B_{2,2}) \\
P_6 &= (A_{1,2} - A_{2,2})\,(B_{2,1} + B_{2,2}) \\
P_7 &= (A_{1,1} - A_{2,1})\,(B_{1,1} + B_{1,2})
\end{aligned}$$

On a alors :

$$A\,B \;=\; \begin{pmatrix} P_5+P_6-P_2+P_4 & P_1+P_2 \\ P_3+P_4 & P_5+P_1-P_3-P_7 \end{pmatrix}$$

Dans le cas général, on a une récurrence du type : $\begin{cases} T(1)=1 \\ T(n)=a\,T(n/b)+f(n) \end{cases}$.

Si $f(n)=O(n^{\log_b a-\varepsilon})$, on trouve $T(n)=\Theta(n^{\log_b a})$.

Si $f(n)=\Theta(n^{\log_b a})$, on trouve $T(n)=\Theta(n^{\log_b a}\log n)$

Si $f(n)=\Omega(n^{\log_b a+\varepsilon})$ et $a\,f(n/b)\leqslant c\,f(n)$, $c<1$, $n\to+\infty$, on trouve $T(n)=\Theta(f(n))$

Par exemple, pour certains tris, on trouve $T(n)=2\,T\big(\frac{n}{2}\big)+O(n)$, ce qui donne $T(n)=O(n\log n)$.

Programmation dynamique

Problème. Alignement de deux suites.

On a un alphabet $\Sigma=\{a,b,c,d\}$ (par exemple).

En entrée, deux suites $s_1s_2\cdots s_n$ et $t_1t_2\cdots t_m$ sur Σ et une fonction score : $(\Sigma\cup\{-\})^2\to\mathbb{R}^+$. En sortie, $\bar{s}$ et $\bar{t}$ sur $\Sigma\cup\{-\}$ tel que $\bar{s}$ est obtenu à partir de s par insertion de « $-$ », $\bar{t}$ est obtenu à partir de t par insertion de « $-$ », et minimisant $\text{cout}(\bar{s}\,,\bar{t}\,)=\sum_i \text{score}(\bar{s}_i,\bar{t}_i)$.

Par exemple, $s=abaace$ et $t=aaade$, on peut sortir $\bar{s}=abaace$ et $\bar{t}=a-aade$.

Il faut commencer par trouver une approche par récurrence. On a trois possibilités :

- aligner s_1 et $-$
- aligner $-$ et t_1
- aligner s_1 et t_1

Pour un algorithme glouton : on choisit la possibilité minimale, puis on continue. Cet algorithme ne fonctionne pas du tout.

En fait, il faut choisir le minimum parmi $\text{score}(s_1,-)+\text{align}(s_2\cdots s_n,t)$, $\text{score}(-,t_1)+\text{align}(s,t_2\cdots t_n)$ ou $\text{score}(s_1,s_2)+\text{align}(s_2\cdots s_n,t_2\cdots t_m)$, valeur que l'on appelle $\text{align}(s,t)$. (il y a aussi des cas de base à traîter si $n=0$ ou $m=0$).

La complexité est alors : $T(n,m)=T(n-1,m)+T(n,m-1)+T(n-1,m-1)+O(1)\geqslant 3\,T(n-1,m-1)+O(1)\geqslant 3^n$, c'est donc un temps exponentiel.

Pour éviter d'avoir à caculer inutilement les mêmes valeurs des résultats de align sur les différents sous-problèmes qui apparaissent, on les enregistre dans un tableau à deux dimensions T tel que $T_{i,j}=\text{align}(s_i\cdots s_n,t_j\cdots t_m)$. On a alors une complexité temporelle et mémoire en $O(n\,m)$.

Attention à l'ordre dans lequel on calcule les différents éléments du tableau T ! On ne peut réutiliser que des éléments qui ont déjà été calclués (par exemple, boucler pour i de n à 1 et pour j de m à 1). Attention aussi aux termes correspondant aux cas de base (ici c'est pour $i=n+1$ et $j=m+1$).

$T[n+1,m+1]\leftarrow 0$
Pour j de 1 à m, $T[n+1,j]\leftarrow ...$
Pour i de 1 à n, $T[i,m+1]\leftarrow ...$
Pour i de n à 1
    Pour j de m à 1
        $T[i,j]\leftarrow \min\{...\}$

Résultat : $T[1,1]$

Pour retrouver les chaînes d'alignement minimal, on peut soit enregistrer ce qu'on fait à chaque étape, soit remonter le tableau en regardant à quoi correspond les cases par lesquelles on passe.

Complément : bin-packing

Entrée. n objets $u_1, ..., u_n$ de tailles $0 \leqslant s_1, ..., s_n \leqslant 1$.

Sortie. Partition des n objets telle que

- chaque partie a comme taille totale au plus 1, ie $\sum_{u_k \in A_i} s_k \leqslant 1$
- on a un nombre minimal de parties

Ce problème n'a pas d'algorithme polynomial si $P \neq \text{NP}$. Il reste deux possibilités : trouver un algorithme non polynomial, ou trouver un algorithme polynomial mais incorrect (approché, non optimal). On peut trouver un algorithme approché donnant un résultat où le nombre de boîtes est inférieur à $1.01 \times \text{OPT} + 1$.

Cas particuliers.

- $\forall i, s_i = \frac{1}{3} \vee s_i = \frac{2}{3}$.
- $\forall i, s_i$ est un multiple de $\frac{1}{10}$.

  On compte le nombre d'éléments pour chaque taille ($n_9 =$ nombre d'objets de taille 9/10, etc.). On compte le nombre de types de boîtes : il y en a $c = 2^{10} = O(1)$. De plus, au total, on devra utiliser au plus n boîtes. Le nombre de possibilités pour les boîtes est donc $c-1$ parmi $n+c-1$ (noté C_{n+c-1}^{c-1} ou $\binom{n+c-1}{c-1}$), qui est de l'ordre n^c, c'est donc polynomial.

- $\forall i,$ on arrondit s_i au dixième supérieur, puis on se retrouve dans le cas précédent.

  Si on applique ça au premier cas, 1/3 devient 0.4 et 2/3 devient 0.7, ces deux objets ne tiennent donc plus ensemble! Et c'est toujours problématique quelle que soit la précision de l'arrondi.

  Dans certain cas on peut trouver un arrondi bien pensé. On peut aussi proposer d'augmenter un peu la capacité des boîtes.

2013-09-30

Analyse en moyenne du tri rapide (CLRS 7.4)

Problème du tri.

Entrée. n nombres $a_1, ..., a_n$.

Sortie. Les nombres triés dans l'ordre croissant.

Types de tris : sélection, insertion, fusion, tas, rapide. Complexités respectives dans le pire des cas : n^2, n^2, $n \log n$, $n \log n$, n^2. Complexités moyennes respectives : n^2, n^2, $n \log n$, $n \log n$, $n \log n$. Les constantes de l'algo quicksort sont également plus petites.

Fonctionnement du tri rapide : on utilise le principe « diviser pour reigner ».

1. Travail préliminaire, on découpe le tableau en deux moitiés L et R.

2. Appeler récursivement l'algorithme de tri rapide sur les deux parties du tableau.

Si après (1) tous les éléments de L sont $\leqslant$ à tous les éléments de R, alors après (2), le tableau est trié.

Algorithme. Tri rapide $a[l, ..., r]$

Si $r \leqslant l$,
alors le tableau est trié
Sinon,
Partager le tableau en $[l, i]$ et $[i+1, r]$de telle façon que $[l, i] \neq [l, r]$ et $[i+1, r] \neq [l, r]$ et :

$$\forall x \in [l, i], \forall y \in [i+1, r], a[x] \leqslant a|y]$$

Tri rapide de $a[l, ..., i]$
Tri rapide de $a[i+1, ..., r]$

Correct. Preuve par récurrence sur la taille $r - l$ du sous-tableau à trier.

Pour le partage du tableau, on prend la valeur $a[l]$ comme « pivot », on met à gauche toutes les valeurs plus petites, à droite toutes celles plus grandes, ce qui se fait en temps linéaire. Ça peut se faire en place en faisant un double parcours du tableau, de gauche à droite en même temps que de droit à gauche. (dès que l'on rencontre deux éléments qui sont dans la mauvaise moitié tous les deux, on les échanges. sinon, on déplace nos pointeurs un par un jusqu'à ce que la condition se réalise.)

Complexité du tri rapide sur un tableau trié : c'est un $\Theta(n^2)$! En effet, il faut résoudre la récurrence $T(n) = \Theta(n) + O(1) + T(n-1)$, ie $\begin{cases} T(n) = c\,n + T(n-1) \\ T(1) = c' \end{cases}$.

Théorème. La complexité en moyenne du tri rapide est $O(n \log n)$, lorsque l'ordre des données d'entrée est une permutation aléatoire uniforme de $\mathfrak{S}_n$ (toutes les $n!$ permutations de $\mathfrak{S}_n$ ont la même probabilité $1/n!$).

Preuve. À un facteur constant près, la complexité de l'algorithme correspond au nombre de comparaisons entre des éléments du tableau.

À une étape donnée, toutes les comparaisons se font entre les éléments du tableau $a[\cdot]$ et le pivot $a[1]$.

Supposons que l'ordre trié soit $a_{\sigma(1)} \leqslant a_{\sigma(2)} \leqslant \cdots \leqslant a_{\sigma(n)}$, avec $\sigma \in \mathfrak{S}_n$.

Remarque : on ne compare $a_{\sigma(i)}$ à $a_{\sigma(j)}$ que 0 ou 1 fois.

Nombre de comparaisons : $\sum_{i,j} X_{i,j}$, où $X_{i,j} = \begin{cases} 1 \text{ si } a_{\sigma(i)} \text{ est comparé à } a_{\sigma(j)} \\ 0 \text{ sinon} \end{cases}$

On a donc : $\boldsymbol{E}(\text{nb comparaisons}) = \boldsymbol{E}\big(\sum_{1 \leqslant i < j \leqslant n} X_{i,j}\big) = \sum_{1 \leqslant i < j \leqslant n} \boldsymbol{E}(X_{i,j}) = \sum_{i,j} \boldsymbol{P}(a_{\sigma(i)}$ est comparé à $a_{\sigma(j)})$

À partir du tableau initial, il y a trois possibilités : soit $a_{\sigma(i)}$ ou $a_{\sigma(j)}$ est le pivot, soit $a_{\sigma(i)}$ et $a_{\sigma(j)}$ sont de part et d'autre du pivot, soit $a_{\sigma(i)}$ et $a_{\sigma(j)}$ sont tous les deux du même côté du pivot. Dans le premier cas, $X_{i,j} = 1$. Dans le second, $X_{i,j} = 0$. Dans le dernier cas, on ne sait pas dire : il faut regarder l'appel récursif et faire la même étude.

Regardons la branche des appels récursifs sur la partie de a qui contient $a_{\sigma(i)}$ et $a_{\sigma(j)}$. $X_{i,j}$ est déterminé une fois qu'on choisit un pivot qui est :

- ou bien $a_{\sigma(i)}$ ou $a_{\sigma(j)}$, auquel cas on va avoir $X_{i,j} = 1$;
- ou bien $a_{\sigma(k)}$, avec $i < k < j$, auquel cas $X_{i,j} = 0$.

Au total il y a $2 + (j - i - 1)$ possibilités, dont 2 donnant $X_{i,j} = 1$ et les autres donnant $X_{i,j} = 0$.

On a donc $\boldsymbol{E}(X_{i,j}) = \frac{2}{j-i+1}$. On peut maintenant calculer : $T(n) = \sum_{1 \leqslant i < j \leqslant n} \frac{2}{j-i+1}$

Soit $k = j - i$, $T(n) = 2 \sum_{k=1}^{n-1} \sum_{i=1}^{n-k} \frac{1}{k+1} = 2 \sum_{k=1}^{n-1} \frac{n-k}{k+1} \leqslant 2\, n \sum_{k=1}^{n-1} \frac{1}{k+1} \leqslant 2\, n \ln n = O(n \ln n)$.

□

Pour éviter le pire cas, on peut effectuer un *tri rapide randomisé*. Première méthode : mélanger le tableau avec une permutation aléatoire de $\mathfrak{S}_n$ (faisable en temps linéaire) puis exécuter le tri rapide classique. Deuxème méthode : tri rapide où l'on choisit un pivot aléatoire parmi le sous-tableau à trier à chaque étape. Les deux méthodes sont identiques d'un point de vue probabiliste.

Tirer une permutation aléatoire en temps linéaire.

On étudie $\varphi(i) = \text{card}\ \{(j,\ \sigma(j)) : j > i \quad \text{et} \quad \sigma(j) > \sigma(i)\}$. Par exemple sur la permutation $(1,2,3,4,5,6,7) \mapsto (4,2,6,7,5,1,3)$, on a le tableau :

i	1	2	3	4	5	6	7
$\varphi(i)$	3	4	1	0	0	1	0

Tableau 1.

(Faire des jolis graphes ! Graphe (1) : $\sigma(i)$ en fonction de i, on dessine les quadrants dans lesquels on fait le comptage pour φ. Graphe (2) : $\varphi(i)$ et cases pour possibilités pour $\varphi(i)$ en fonction de i.)

On observe que $\forall i, 0 \leqslant \varphi(i) \leqslant n - i$. Il y a donc $n!$ fonctions φ possibles. Lemme : il y a une bijection entre $\mathfrak{S}_n$ et les fonctions φ. Algorithme :

1. Choisir φ aléatoire : temps $O(n)$ (on sait donner un entier aléatoire en temps constant) ;
2. Construire la permutation $\sigma \in \mathfrak{S}_n$ correspondant à φ.

Complexité du problème de tri (CLRS 8.1)

Peut-on faire mieux que $n \log n$ en complexité dans le pire des cas ? Et en moyenne ?

Théorème. Résultat sur le problème du tri. Pour tout algorithme A de tri qui compare les éléments entre eux, le nombre de comparaisons fait par A dans le pire des cas est toujours $\geqslant \frac{1}{100} n \log n (n \to \infty)$.

Preuve. Soit A un algorithme de tri. Soit $C(n)$ la complexité au pire cas de l'algorithme A.

$n = 2$: $C(2) \geqslant 1$ évident.

$n = 3$: $C(3) \geqslant 2$.

De manière générale, $C(n) \geqslant n/2$ car chaque élément devra être comparé au moins une fois.

(Si on fait un graphe avec un somet par élément à trier et une arête par comparaison, le graphe doit être connexe, donc $C(n) \geqslant n - 1$.)

En dessinant l'arbre des comparaisons pour le cas $n = 3$, on voit que $C(3) \geqslant 3$ (on s'intéresse bien à la branche la plus longue !). Cf dessin (3).

Ceci suggère une manière de représenter les algorithmes, sous forme d'un *arbre de décision* : chaque noeud correspond à une comparaisons, chaque branche à ce que l'on fait dans le cas inférieur ou supérieur (correspondant aux sous-arbres droit et gauche). Chaque feuille correpsond à un résultat de l'algorithme, un ordre trié, une permutation des éléments. Une exécution de A sur une donnée correspond à un chemin de la racine vers une feuille. Le nombre de comparaisons faites correspond à la longueur du chemin. La complexité du pire cas équivaut à la hauteur de l'arbre.

Deux chemins différents correspondent à des ordres différents.

On a $n!$ permutations possibles, c'est-à-dire $n!$ feuilles dans l'arbre.

Lemme. Un arbre de décision de hauteur h comporte au plus 2^h feuilles. (on pourrait facilement aller jusqu'à dire que c'est trivial).

Fait. L'arbre de décision d'un algorithme de tri de n éléments contient (au moins) $n!$ feuilles.

La complexité au pire cas de A est donc h tel que $2^h \geqslant \#\text{feuilles} \geqslant n!$.

On a donc $n! \leqslant 2^h$, $h \geqslant \log_2 n! \sim \Theta(n \log n) > \frac{1}{100} n \log n$ pour n assez grand, grâce à la formule de Stirling, ou avec une minoration bourrin : $\log_2 n! = \log_2 (1 \cdot 2 \cdot \cdots n) \geqslant \log_2 (n/2 \cdot (n+1)/2 \cdots n) \geqslant \log_Z \left((n/2)^{n/2} \right) \geqslant n/2 \log_2 (n/2)$.

Remarque. Cette borne inférieure est également valable en moyenne.

Hachage linéaire, hachage coucou (CLRS 11.4)

Hachage

Un tableau trié sert à trouver un élément en temps $O(\log n)$ par recherche binaire. Comment faire mieux que $\log n$?

On peut utiliser une fonction de hachage.

Si on a un ensemble S de n éléments tirés d'un univers U à ranger dans un tableau de taille $2\,n$, on va utiliser une fonction de hachage $h : U \to \{1, ..., n\}$.

Avec une telle fonction on range les éléments de S dans les cases $h(x)$. Pour décider si un élément $x \in U$ est dans S, il suffit de regarder la case $h(x)$. Mais que faire si deux éléments x, y de S ont la même valeur : $h(x) = h(y)$? C'est ce qu'on appelle une *collision*.

Si on veut des cases pour toutes les valeurs possibles de U, on aura un temps constant, mais il y aura trop de mémoire.

Il faut gérer les collisions, ce qui permet d'avoir un tableau de taille raisonnable et de répondre en temps globalement constant aux requêtes faites sur les données, du type « $x \in S$? ».

Solution 1. *Hachage linéaire.*

Création de la table :

Pour tout $x \in S$:

- calculer $h(x)$;
- si la case $T(h(x))$ est libre, y mettre x ;
- sinon, si la case $T(h(x)+1)$ est libre, y mettre x ;
- et ainsi de suite, regarder successivement les cases $h(x)$ + 2, ..., $h(x)$ + i(mod n) jusqu'à trouver une case libre où mettre x.

Exemple : $n = 8$. (voir dessin (4)).

La recherche n'est pas du tout optimisée : il faut parcourir le tableau à partir de $h(x)$ jusqu'à trouver une case vide... Pour avoir suffisament de cases vides et rendre cette recherche plus rapide, on peut par exemple prendre $m = 2\,n$.

Théorème. Si $m = 2\,n$ et si $\forall x \in U$, $h(x)$ est uniforme entre 0 et $2\,n - 1$, alors pour le hachage liéaire, la complexité moyenne d'une opération de recherche est $O(1)$.

Preuve.

Soit $x \notin S$. Avec probabilité $1/2$, $h(x)$ est une case libre, dans ce cas on a fini en temps 1.

Avec proba $1/2$, $h(x)$ est occupé, on regarde donc $h(x)+1$ qui de même a une proba $1/2$ d'être libre. On a fini en temps égal au nombre de cases occupées jusqu'à la prochaine case libre (on note ce temps $C(x)$).

$$\begin{aligned} C &= \frac{1}{2}\left(\sum_k \sum_{\text{groupes taille } k} \underbrace{\mathbb{P}(\text{tomber ds ce groupe})}_{=k/n} \underbrace{\mathbb{E}(\text{coût sachant on tombe ds ce groupe})}_{k/2}\right) \\ &= \Theta\left(\sum_k \sum_{\text{groupes de taille } k} \frac{k^2}{n}\right) \end{aligned}$$

Soit I un intervalle de T de longueur k : $I = T[i_0, i_0+1, ..., i_0+k-1 (\mathrm{mod}\, n)]$

$$\begin{aligned} \mathbb{P}(I \text{ est un groupe}) &\leqslant \mathbb{P}(k \text{ éléments de } S \text{ sont tels que } h(x) \in I)^{n-k} \\ &= \underbrace{\binom{n}{k}}_{\text{choix de } S} \underbrace{\left(\frac{k}{2\,n}\right)^k}_{\substack{\text{les } k \text{ éléments sont} \\ \text{tels que } h(x)\in I}} \underbrace{\left(1-\frac{k}{2n}\right)}_{\text{les } n-k \text{ autres de } S \text{ sont tq } h(x)\notin I} \\ &\approx \frac{n^n e^{-n}}{k^k\, e^{-k}\, (n-k)^{n-k}\, e^{-(n-k)}} \cdot \frac{k^k}{2^k\, n^k} \cdot \left(1-\frac{k}{2\,n}\right)^{n-k} \\ &= \frac{1}{2^k} \cdot \frac{(1-k/2n)^{n-k}}{(1-k/n)^{n-k}} \\ &\leqslant \frac{1}{2^k} \cdot \left(1+\frac{k}{2\,n}\right)^{n-k} \\ &\leqslant \left(\frac{\sqrt{e}}{2}\right)^k \to 0 \end{aligned}$$

2013-10-07

Arbres

Déf. Une *structure de données abstraite* est une structure définie par les opérations que l'on peut faire dessus, indépendamment de comment on les fait. (c'est une spécification)

Déf. Un *dictionnaire* est une structure de donnée permettant d'enregistrer un ensemble S d'éléments tirés d'un univers totalement ordonné. Les opérations disponnibles sont :

- Rechercher si $x \in S$
- Insérer x dans S
- Supprimer x de S

Structure de données concrètes	Arbre binaire de recherche	Arbres rouge-noir	« treaps » « tarbres » (randomisé)	« splay trees » arbres éclatés (amorti)
Recherche	$O(\text{hauteur}) = O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Insertion	$O(n)$	(pire cas)	en moyenne	amortie
Suppresion	$O(n)$	pour chaque op.	puor chaque op.	pour chaque op.

Tableau 2.

Arbre rouge-noir

C'est un arbre de recherche avec des marques : chaque noeud de l'arbre a 0 ou 1 marque. S'il en a 0, on dit qu'il est rouge. S'il en a 1, on dit qu'il est noir. Les propriétés suivantes doivent être respectées :

- tous les chemins de la racine à une feuille doivent avoir le même nombre de marques (noires) ;
- un fils et son père ne peuvent être tous les deux rouges.

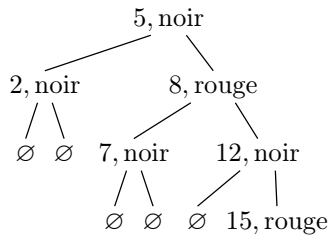


Figure 1. Exemple d'arbre rouge-noir.

Cette structure garantit que tous les chemins racine-feuille ont la même longueur à un facteur deux près. Si n est le nombre de noeuds, la hauteur de l'arbre est donc $\Theta(\log n)$.

Recherche. Identique à un ABR.

Insertion.

1. Comme dans un ABR. On marque le noeud inséré en rouge.
2. Corriger la structure si le noeud et son père sont tous deux marqués rouges. Mais comment ? On effectue une correction locale (nombre constant d'opérations) qui soit corrige le problème immédiatement, soit fait remonter le problème au niveau d'au-dessus ; on recommence alors jusqu'à ce que le problème soit corrigé (au pire des cas on remonte jusqu'à la racine). Il est dores et déjà clair qu'avec cette approche, la complexité sera de $O(\log n)$.

   Inventaire des corrections possibles.

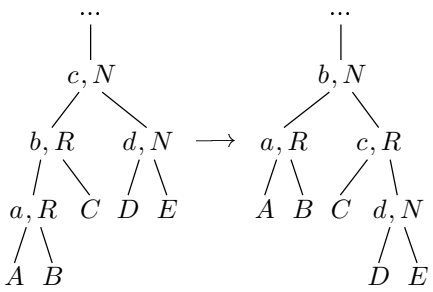


Figure 2. Les noeuds rouges/noirs sont respectivement annotés R et N.

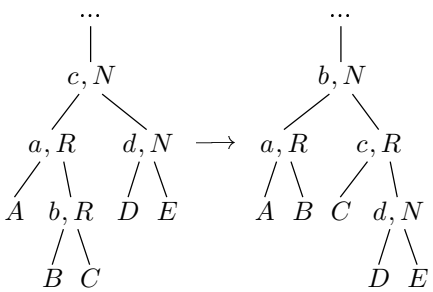


Figure 3. Autre correction possible.

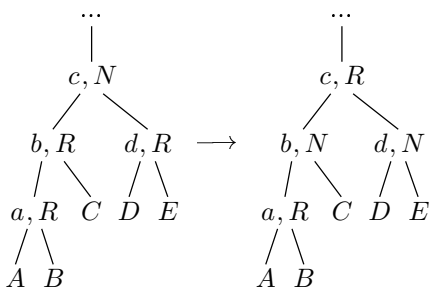


Figure 4. Ici on risque de faire remonter le problème au niveau d'au-dessus.

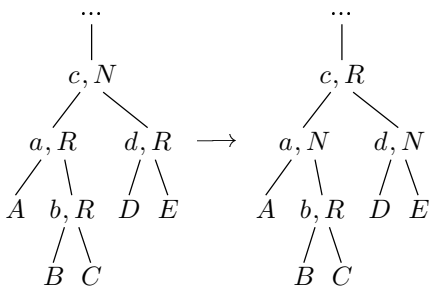


Figure 5. De même ici.

Si on arrive à la racine avec un problème, on peut simplement colorier la racine en noir, ce qui suffit généralement.

Suppression.

1. Suppression comme dans un ABR : on recherche le plus petit élément du sous-arbre droit, dont on prend la valeur pour replacer au niveau du noeud à supprimer. Ce plus petit élément est ensuite supprimé ; s'il était noir, on rajoute un marqueur noir sur son seul fils possible, le fils droit, qui contient au maximum un noeud nécessairement rouge. Le seul problème se fait si le noeud final supprimé était noir et n'avait pas de fils du tout. Dans ce cas, on s'arrange sur le sous-arbre concerné qui a une taille très limitée.

2. On se retrouve parfois avec un double marqueur noir à faire remonter.

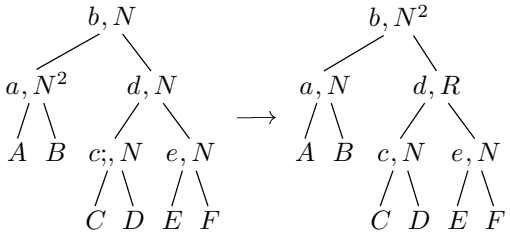


Figure 6.

« Treaps », ou « tarbres »

Chaque noeud a deux valeurs : (x_i, t_i). Du point de vue des x_i, c'est un ABR. C'est meme l'unique ABR qui aurait été obtenu à partir de l'arbre vide en insérant les x_i un par un dans l'ordre des t_i croissants.

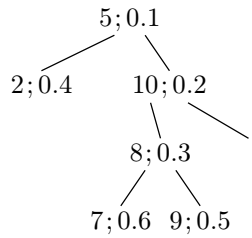


Figure 7. Exemple de tarbre, obtenu en insérant $5, 10, 8, 2, 7, 9$ dans cet ordre

Pour un tarbre randomisé, on utilise des t_i aléatoires.

Recherche. Comme dans un ABR.

Insertion. On choisit t_i uniformément au hasard dans, par exemple, l'intervalle $[0, 1]$. On l'insère ensuite comme dans un ABR, puis on corrige les incohérence de temps : les fils ont toujours un temps plus grand que leur père. Cette correction se fait en faisant remonter dans l'arbre le noeud concerné.

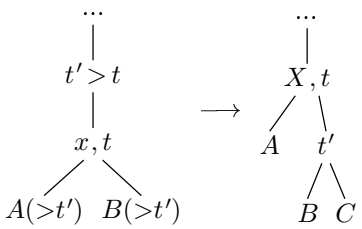


Figure 8.

Suppression. On supprime comme dans un ABR, en faisant remonter l'ettiquette de temps avec la valeur. Il faut ensuite éventuellement corriger cette ettiquette qui est trop tardive, en effectuant des rotations : on fait rotationner l'élément perturbant avec celui de ses fils qui a l'étiquette temporelle la plus précoce, afin de ne pas poser de problème avec l'autre fils (vu qu'on devient alors un de ses parents). On réitère ensuite au niveau de dessous (ie toujours en prenant le noeud perturbateur comme racine).

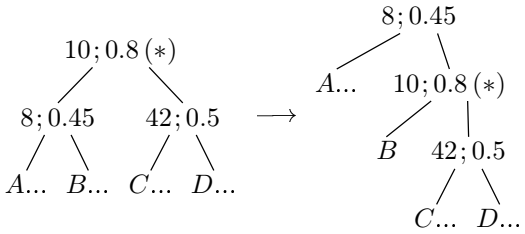


Figure 9. L'anomalie $(*)$ descend d'un cran ; la nouvelle racine partielleest le 8 vu son étiquette.

Complexité. Quelle est le coût moyen d'une recherche dans un ABR obtenu par insertion dans un ordre aléatoire ? Vu que les racines partielles sont choisies au hasard, les deux sous-arbres sont en moyenne équilibré à chaque fois. Le calcul est le meme que dans le cas du quick-sort randomisé. On trouve une hauteur en $O(\log n)$.

Arbres éclatés

On se base sur la structure d'un ABR.

Sleator & Torjan. « éclatement » d'un chemin : la racine devient le noeud x recherché.

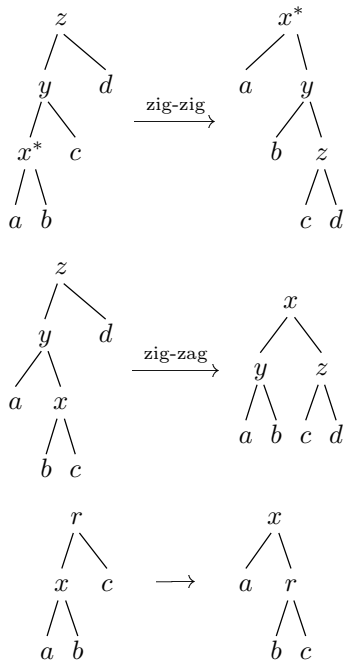


Figure 10.

Théorème. Si on part d'un arbre vide et qu'on fait une suite de m opérations de dictonnaire, le cot total est $O(m \log n)$.

2013-10-14, Jacques STERN (transparents à trouver sur Internet)

Algorithmes sur les graphes

Un graphe est une relation binaire sur un ensemble fini. On le représente habituellement avec des sommets numérotés, et la relation binaire est généralement représentée par des flèches. Une relation binaire symmétrique induit un graphe non orienté, et on représente alors des traits lorsque eux sommets sont liés.

Le nombre de sommets est toujours noté n, le nombre d'arêtes m.

On peut représenter un graphe de plusieurs manières, par exemple par une matrice d'adjacence ou par une liste d'adjacence (pour chaque sommet on a une liste chaînée des sommets accessibles depuis ce sommet ci).

Algorithme de Dijkstra

On munit le graphe d'une valuation entre chaque paire de noeuds notée $w(v, k)$, qui vaut conventionnellement ∞ si les noeuds ne sont pas reliés. Dans un premier temps les valuations seront toujours positives.

On s'intéresse au problème du plus court chemin, ie construire un chemin $\gamma = u_0, u_1, ..., u_k$ (où (u_i, u_{i+1}) est une arête pour chaque i) pour $u_0 = s$ et $u_k = t$ fixés, muni d'un poids $w(\gamma) = \sum_{i=0}^{k-1} w(u_i, u_{i+1})$ minimal.

Probème : calculer ce minimum, expliciter un chemin le réalisant.

L'algorithme de Dijkstra est basé sur une fonction de relaxation, qui maintient un tableau contenant des majorants de la distance entre un sommet source s et chaque sommet du graphe. Cette distance est initialisée à 0 pour le sommet s, et à ∞ pour les autres sommets.

Algorithme 1

DIJKSTRA(G, w, s)

    INITIALIZE-SINGLE-SOURCE(G, s)
    $S \leftarrow \varnothing$
    $Q \leftarrow V[G]$
    while $Q \neq \varnothing$
        do $u \leftarrow$ EXTRACT-MIN(Q)
            $S \leftarrow S \cup \{u\}$
            for each vertex $v \in \mathrm{Adj}[u]$
                do RELAX(u, v, w)

INITIALIZE-SIGNLE-SOURCE(G, s)

    for each vertex $v \in V[G]$
        do $d[v] \leftarrow \infty$
            $\pi[v] \leftarrow$ NIL
    $d[s] \leftarrow 0$

RELAX(u, v, q)

    if $d[v] > d[u] + w(u, v)$
        $d[v] \leftarrow d[u] + w(u, v)$
        $\pi[v] \leftarrow u$

Il faut prouver que quand cet algorithme se termine, il a bien calculé la fonction distance.

On pose $\delta(d, v) = \min w(\gamma)$ pour les chemins γ reliant d à v.

On montre ensuite $\delta(s, v) \leqslant d(v)$ conservé pendant tout l'algorithme.

Lemme. Si $\delta(s, u) = d(u)$ et si un plus court chemin de s à v a pour avant dernier point u, alors RELAX(u, v, w) rend $\delta(s, v) = d(v)$.

Il reste à démontrer qu'à la fin de l'algorithme, toutes les valeurs de $d(s)$ se sont stabilisé à $\delta(s, v)$.

Preuve. Soit un premier u placé dans S avec $\delta(s, u) \neq d(u)$, soit y le premier sommet hors de S sur le plus court chemin de s à u, soit x le sommet avant y. Par minimalité, $\delta(s, x) = d(x)$, par le lemme, $\delta(s, y) = d(y)$, par positivité, $d(y) < d(u)$. □

Le tableau π, lorsque la valeur de $d(u)$ se stabilise, contient en $\pi(u)$ un avant dernier point sur un plus court chemin de s jusqu'à u. Une fois que l'algorithme est terminé, toutes les cases de π sont ainsi remplies. On est donc capable de régénerer un chemin de s à n'importe quel noeud v du graphe, en itérant la fonction π au point v, puis $\pi(v)$, etc.

Complexité. Tout dépend de la structure de donnée qui gère l'ensemble Q des sommets restants à étudier. Cette structure doit supporter deux types d'opérations : extraction du minimum pour la valeur de d, et mise à jour lorsque les valeurs de d diminuent.

Si on ne gère pas cet ensemble de manière évoluée, par exemple avec un tableau, la contribuition de EXTRACT-MIN induit une complexité en $O(n^2)$.

On peut utiliser un tas binaire, qui permet de faire les deux opérations (recherche du minimum et décrémentation d'une clef) en temps $O(\log n)$, ce qui donne une complexité totale en $O(n \log n + m \log n)$.

Il existe une structure qui permet d'obtenir $O(n \log n + m)$: le tas de Fibonacci. Le coût de l'extraction reste en $\log n$, mais la décrémentation d'une clef se fait en temps constant.

Cas d'une valuation positive ou négative

Il y a un problème d'existence du minimum.

Déf. On appelle *cycle* un chemin $\gamma = u_0 ... u_k$ tel que $u_0 = u_k$.

Déf. On appelle *chemin sans boucle* un chemin $\gamma = u_0 \cdots u_k$ un chemin tel que $u_i \neq u_j$ si $i \neq j$.

Le problème est posé par les cycles de poids négatif : on peut faire le tour du cycle autant de fois qu'on veut, le poids total du chemin est donc non borné négativement.

Dans un graphe où tous les cycles sont de poids positifs : tout chemin a un poids minoré par celui d'un chemin sans boucle de mêmes extrémités, le problème est alors bien posé.

Algorithme de Bellman-Ford. On effectue $n-1$ fois une boucle où l'on relaxe toutes les arêtes. À l'issue de cet algorithme, tous les chemins sans boucles sont entièrement relaxés (à l'issue du premier tour, tous les chemins de longueur 1 sont relaxés, ensutie de longueur 2, etC.)

S'il exste une boucle de chemin négatif, on trouvera une arête (u, v) tel que $d[v] > d[u] + w(u, v)$ est toujours vrai après cet algo, le problème est mal posé. Si on ne trouve pas de telle arête, on est bon.

Algorithme 2

BELMAN-FORD(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
for $i \leftarrow 1$ **to** $|V[G]| - 1$
  do for each edge $(u, v) \in E[G]$
    do RELAX(u, v, q)
for each edge $(u, v) \in E[G]$
  do if $d[v] > d[u] + w(u, v)$
    then return FALSE
return TRUE

Problème avec sources multiples

Par programmation dynamique on calcule la matrice $\delta^m(u, v)$ des distances sur les chemins d'au plus m arêtes. La récurrence suivante permet de passer des chemins de longueur $m-1$ aux chemins de longueur m : $\delta^m(u, v) = \min_k \{\delta^{m-1}(u, k) + w(k, v)\}$ (utiliser $w(v, v) = 0$).

C'est analogue à la multiplication matricielle en remplaçant $+$ par min et $\times$ par $+$. A-t-on la distributivité du $+$ sur le min ? Il faut vérifier que $\min(a + c, b + c) = \min(a, b) + c$, ce qui est à peu près clair. Cela signifie que l'on peut utiliser par exemple la méthode des carrés itérés, ce qui fait une complexité $O(m^3 \log m)$.

Autre méthode :

Algorithme de Floyd-Warshall. Par programmation dynamique on calcule la matrice $\delta^k(u, v)$ des distances sur des chemins ne passant que par des points intermédiaires d'indices $\leqslant k$.

La formule de mise à jour est maintenant : $\delta^k(u, v) = \min_k \{\delta^{k-1}(u, v), \delta^{k-1}(u, k) + \delta^{k-1}(k, v)\}$ (on rajoute l'arête k, que l'on parcours zéro ou une fois).

Cette méthode s'applique aussi à la cloture transitive d'un graphe (remplacer le $+$ par un inf (un et logique) et le min par un sup (un ou logique)) : on veut calculer la matrice qui donne 1 à (u, v) s'il existe un chemin de u à v, 0 sinon.

La complexité de cet algorithme est $O(n^3)$.

Cours avancé : tas binomiaux, tas de Fibonacci

On cherche des structures adaptées à la réunion de plusieurs ensembles, tout en conservant les propriétés suivantes efficaces : insertion, suppression, extraction du minimum.

On perd la possibilité de recherche, mais on conserve la propriété des tas : tout élément a une clé inférieure à celles de ses fils.

Tas binomiaux

Un arbre binomial est soit un noeud (B_0), soit deux tas binomiaux de même indice liés : $B_k = (B_{k-1}, B_{k-1})$.

Un tas binomial est une liste d'arbres binomiaux, où chaque arbre binomial aparaît au plus qu'une seule fois.

On utilise une structure de cellule qui enregistre : parent, clef, degré, enfant, frère suivant.

Réunion de deux tas binomiaux. C'est comme une addition binaire ! si on a deux fois le même arbre B_k, on le transforme en B_{k+1} (en choisissant celui avec la racine la plus petite comme racine pour conserver la propriété de tas), ce qui fait une retenue à propager.

Cette fusion se fait par étapes ; on peut se retrouver avec au pire trois arbres identiques à une certaine étape donnée, dans ce cas là on garde le premier des trois.

Extraire le minimum. Il faut trouver l'arbre binomial, de forme B_k, qui a la racine la plus petite. Après avoir supprimé cette racine, on se retrouve avec une liste d'arbres binomiaux $B_0, ..., B_{k-1}$, que l'on peut fusionner avec ce qui reste du tas originel par l'algorithme précédent.

Décroître une clef. Il suffit d'intervertir le noeud avec ses parents si sa clef devient plus petite. Pour cela, il faut déjà disposer d'un pointeur vers ce noeud.

Supprimer une clef. On la fait remonter en lui assignant la valeur $-\infty$, puis on extrait un minimum.

Procédure	Complexité
Créer un tas	$\Theta(1)$
Insérer un élément	$O(\log n)$
Minimum	$O(\log n)$
Extraire minimum	$\Theta(\log n)$
Union	$O(\log n)$
Diminuer une clef	$\Theta(\log n)$
Supprimer	$O(\log n)$

Tableau 3. Complexité des opérations habituelles sur un tas binomial

Justification. On sait que chaque degré apparaît au plus une fois dans les racines du tas. Donc si on a t racines, le nombre d'éléments est au moins $1 + \cdots + 2^t = 2^{t+1}$, donc réciproquement avec n éléments dans le tas on a au plus $O(\log n)$ racines d'arbres binomiaux.

Tas de Fibonacci

Structure similaire mais plus flexible. C'est une structure paresseuse, avec un temps d'exécution amorti. Il n'y a pas d'indication précise sur la forme des arbres, mais on va régulièrement vérifier qu'il n'y a pas deux arbres de même degré (degré = nombre de fils).

Chaque noeud x a deux champs :

1. degre x : nombre de fils
2. marque x : indique si x a perdu un fils depuis la dernière fois où il est devenu fils

On définit le potentiel suivant : $\psi = \#\text{arbres} + 2(\#\text{marques}) = a + 2m$

Union et insertion. L'union de deux tas est la concaténation. On met à jour le minimum que l'on enregistre quelque part.

- Cout réel : $O(1)$
- Cout amorti : $O(1)$
  - insertion : $a = a + 1$, $m = m$
  - fusion : $a(T) = a(T_1) + a(T_2)$, $m(T) = m(T_1) + m(T_2)$

Extraction du minimum. On enlève le minimum, puis on insère ses fils. Ensuite, on consolide :

- Objectif : réduire le nombre d'arbres de sorte que les racines aient des degrés distincts
- Principe : trouver les racines de même degré et les relier dans un même arbre
- Pour cela, on utilise un tableau $A[0, ..., D(n)]$ tel que $A[i] = x$ si x est une racine et $\text{degre}\, x = i$ ($D(n)$ est le degré maximal d'un noeud).

On initialise le tableau A à vide, puis on insère les éléments du tas en fusionnant à chaque fois comme il faut pour qu'on ait à chaque étape au plus un tas de degré i pour tout i. On prend ensuite tous les éléments des cases non vides du tableau, ce qui donne les éléments du tas consolidé.

Analyse en complexité de la consolidation.

- Cout réel : $O(D(n) + a(T))$
- Degré $D(n)$ (degré maximal sur tous les noeuds) en $O(\log n)$, cf plus loin
- Potentiel avant : $a(T) + 2m$
- Potentiel après : $D(n) + 1 + 2\,m$
- Cout amorti = cout réel + différence de potentiel $\Delta\psi = O(D(n)) = O(\log n)$.

Faire décroitre une clef.

1. On replace le noeud à la racine (et on lui enlève sa marque)
2. On replace son père à la racine (et on lui enlève sa marque) s'il est marqué, et ainsi récursivement.
3. On marque le premier père non marqué rencontré.

- Cout réel : $O(c)$, c = nombre d'appels récursifs
- Cout amorti : $O(1)$, en effet :

  $a \to a + (c - 1) + 1 = a + c$; $m \to m - c + 2$

$\Delta\psi = 4 - c$

Borne sur le degré.

Théorème. Tout tas de Fibonacci à n noeuds obtenu par une suite d'opérations ci-dessus a un degré en $O(\log n)$.

Lemme. En numérotant les fils de x dans l'ordre où ils sont apparus, soit y_1, ..., y_k, on a $\deg(y_1) \geqslant 0$ et $\deg(y_i) \geqslant i - 2$ pour $i \geqslant 2$.

Preuve. Au moment où y_i est lié à x, on a $\deg x = \deg y_i = i - 1$; ensuite y_i perd au plus un fils.

Lemme. Pour tout noeud x de degré k d'un tas de Fibonacci, le sous-arbre issu de x a au moins $\mathrm{Fib}(k+2)$ éléments.

Preuve. Soit $\#(x)$ la taille du sous-arbre issu de x de degré k.

$$\begin{aligned} \#(x) &= 2 + \sum_{i=2}^{k} \#(y_i) \\ &\geqslant 2 + \sum_{i=2}^{k} \mathrm{Fib}(i) \\ &= \mathrm{Fib}(k+2) \end{aligned}$$

(il faut faire se télescoper les termes correctement)

Lemme. $\mathrm{Fib}(k) \geqslant \Phi^k$, où $\Phi = \left(1 + \sqrt{5}\right)/2$.

La complexité est donc démontrée.

Procédure	Complexité amortie
Créer un tas	$\Theta(1)$
Insérer un élément	$\Theta(1)$
Minimum	$\Theta(1)$
Extraire le minimum	$O(\log n)$
Union	$\Theta(1)$
Diminuer clé	$\Theta(1)$
Supprimer	$O(\log n)$

Tableau 4. Complexité des opérations habituelles sur un tas de Fibonacci

2013-10-21

Couplage

Entrée. Graphe biparti $G = (A, B, E)$, $V = A \cup B$, $E \subset A \times B$.

Sortie. Couplage M (« matching »), $M \subseteq E$ de cardinal maximum tel que $\forall u \in V,\ u$ appartienne à au plus une arête de M.

Définitions.

- *Chemin alternant* : étant donné un graphe biparti G et un couplage M, c'est un chemin qui alterne entre arête de M et arête de $E \setminus M$.
- *Sommet exposé* : sommet n'appartenant à aucune arête du couplage M.
- *Chemin augmentant* : chemin alternant dont le premier et le dernier sommet sont exposés.

Théorème. Soit G un graphe biparti et M un couplage de G. Alors M est de cardinal maximum $\Leftrightarrow$ il n'existe pas de chemin augmentant.

Preuve.

$\Rightarrow$: S'il y a un chemin augmentant, c'est facile de créer un couplage de cardinal plus grand : on intervertit sur ce chemin les arêtes appartenant au couplage et celles n'y appartenant pas.

$\Leftarrow$: Si M n'est pas maximum, montrons qu'on peut trouver un chemin augmentant : soit M^* un couplage maximum, on a donc $|M^*| > |M|$. Considérons le graphe obtenu en superposant les arêtes de M et de M^*. Chaque composante connexe est un chemin simple ou un cycle. On regarde les types de composantes connexes possibles : il y a des cycles avec autant d'arêtes dans chacun des deux couplages, des chemins avec autant d'arêtes dans chaque couplage, des couples de noeuds reliés par une arête dans M et M^*, et des chemins avec une arête de plus dans M^* que dans M. Il y a au moins un chemin de ce dernier type, ce qui donne un chemin augmentant pour M. □

Algorithme 3

Entrée : G
$M \leftarrow \varnothing$
Tant qu'il existe un chemin augmentant pour M
    S'en servir pour augmenter M
Sortie : M
Complexité : $O(|M^*|) \times$complexité de la recherche du chemin augmentant

Entrée : graphe G, couplage M
Sortie : un chemin augmentant pour M, s'il existe
Soit A_1 l'ensemble des sommets exposés de A, B_1 l'ensemble des sommets exposés de B. On cherche un chemin alternant qui relie A_1 à B_1 (tout chemin augmentant est de longueur impaire, et relie donc un sommet de A_1 à un sommet de B_1).
On construit (recherche de chemin) une forêt $F \subseteq G$ avec un arbre pour chaque sommet de A_1.
$V_F \leftarrow A_1$, $E_F \leftarrow \varnothing$
répéter.
    $X \leftarrow \{$sommet b de B tel que $\exists a \in V_F$ et une arête $(a, b \in E)$ et $b \notin V_F\}$
    Si X contient un sommet de B_1, c'est fini, on a un chemin augmentant
    Si $X = \varnothing$, c'est fini : pas de chemin augmentant.
    Autrement : **pour chaque** sommet b de X,
        choisir une arête (a, b), $b \notin V_F$
        ajouter (a, b) à E_F et b à V_F
        Soit $M(b)$ le sommet de A voisin de b par M
        ajouter $(b, M(b))$ à E_F, $M(b)$ à V_F.

Preuve de correctitude. Si on arrive sur une étape où $X = \varnothing$, est-il vrai que M est maximal ? Preuve en exo. □

Complexité. Calcul de X en temps $O(\sum_{\text{sommets du « front »}} \text{degrés})$. Le calcul d'un chemin augmentant se fait donc en $O(\sum_{u \text{ sommet du front}} \text{degre}\,(u))$. Complexité totale $\leqslant \sum_{u \in A} \text{degre}(u) = \#E$ (par convention le nombre d'arêtes est noté m). On trouve donc un couplage maximum en temps $O(m\,n)$.

Extensions.

- couplage dans un graphe général (non biparti) : algorithme de Edmonds
- couplage de poids maximal dans un graphe biparti où les arêtes ont un poids
- couplage de cardinal maximal dans un graphe biparti dont les sommets (de A) arrivent au fil du temps

- couplage de poids maximal dans un graphe général
- mariage stable

Couplage biparti en ligne (online bipartite matching)

Les sommets de B sont fixés au début du problème. Les sommets de A arrivent au cours du temps, un par étape avec leur liste entière de connections aux sommets de B. À chaque étape, on doit décider si le sommet qui arrive doit être couplé à un sommet de B et si oui, à qui. On regarde ensuite à la fin si le couplage que l'on a construit est minimal ou non.

$G = (A \cup B, E)$. Initialement, on connait B. À chaque étape, un sommet de A arrive, avec toutes ses arêtes $((\{a\} \times B) \cap E)$. On décide ou non de le coupler avec un noeud de B non encore couplé. Ce choix est effectué par un algorithme. On met alors M à jour. La sortie est le couplage M obtenu à la fin de l'algo.

Thm. Il existe un algo probabiliste (construisant un couplage aléatoire) tel que le M obtenu vérifie $\mathbb{E}(|M|) \geqslant \left(1 - \frac{1}{e}\right) |M^*|$ (environ 57% du couplage maximal). Cette algorithme maximise l'espérance (il est optimal).

Avec un algorithme glouton, on obtient facilement $|M| \geqslant 0.5 \times |M^*|$. Seulement avec un algorithme probabiliste peut-on obtenir une meilleure espérance.

Algorithme 4

On choisit un ordre de priorité aléatoire pour les sommets de B (choisi uniforme parmi les $n!$ permutations possibles, avec $n = |B|$).
À l'arrivée d'un sommet $a \in A$, si a possède des voisins exposés, on le couple avec le voisin exposé qui a la plus forte priorité.

Preuve. On note G notre graphe, π 'ordre d'arrivée des sommets de gauche, σ l'ordre de priorité des sommets de droite. On note priorite(G, π, σ) l'application qui à une exécution de l'algorithme donne un couplage.

Soit $x \in V$, $H = G \backslash \{x\}$. On a alors π_H, σ_H.

Alors priorite$(G,\ \pi,\ \sigma)$ et priorite$(H,\ \pi_H,\ \sigma_H)$ sont soit égaux, soit différents d'un chemin alternant commençant en x. On note U et V les parties gauches et droites.

On suppose $|U| = |V|$, et $|M^*| = n$. On note $m^*(u)$ le sommet couplé à u dans M^*.

Lemme. Soit $u \in U, v = m^*(u)$. Si v n'est pas couplé par priorite(G, π, σ), alors u est couplé à un sommet v' tel que $\sigma(v') < \sigma(v)$.

Lemme. Soit X_t la probabilité que le sommet de V de rang t ($\sigma(v) = t$) soit couplé. Alors $1 - X_t \leqslant \frac{1}{n} \sum_{s=1}^{t-1} X_s$.

Pourquoi ce lemme implique le théorème : $|M^*| = n$, $\mathbb{E}(|\text{sortie}|) = \sum_{s=1}^{n} X_s$. Soit $S_t = \sum_{s=1}^{t} X_s$. Le lemme dit : $1 - (S_t - S_{t-1}) \leqslant \frac{1}{n} S_{t-1}$, $S_t \geqslant 1 + \left(1 - \frac{1}{n}\right) S_{t-1}$.

> **Démonstration du lemme.** soit v le sommet de V de rang t (dans σ). $\mathbb{P}(v \text{ non couplé}) = 1 - X_t$. Soit u tel que $v = m^*(u)$. Soit $R_{t-1} = \{\text{sommets de } U \text{ couplés par l'algorithme à des sommets de } V \text{ de rang } \sigma \leqslant t-1\}$. Taille moyenne de R_{t-1} : $\mathbb{E}(|R_{t-1}|) = \sum_{s=1}^{t-1} X_s$. v non couplé $\Rightarrow$ u est couplé à un sommet de priorité $\leqslant t-1 \Rightarrow u \in R_{t-1}$. Donc $\mathbb{P}(v \text{ non couplé}) \leqslant \mathbb{P}(u \in R_{t-1})$, ie $1 - X_t \leqslant \frac{\sum_{t=1}^{n} X_s}{n}$. □ (en fait il y a une erreur ici, mais cette première preuve buggée a été gardée pendant un certain temps, avant qu'on ne s'intéresse vraiment à la question.)

Lemme correct. Soit X_t la probabilité que le sommet de V de rang t ($\sigma(v) = t$) soit couplé. Alors $1 - X_t \leqslant \frac{1}{n} \sum_{s=1}^{t} X_s$.

Couplage dans un graphe quelconque

Algorithme existant, existe sur Wikipédia. (algorithme de Edmonds)

Composantes fortement connexes dans un graphe

Entrée. G un graphe orienté.

Sortie. Liste des composantes fortement connexes de G.

Déf. Composante fortement connexe de x : $\{y \in G : \exists x \to \cdots \to y \wedge \exists y \to \cdots \to x\}$.

Algorithme à regarder ! (il en existe un : algoritme de Tarjan, basé sur un parcours en profondeur)

2013-10-28

Flots et coupes dans les graphes

Définition du problème

On s'intéresse à un problème de maximisation de flots, qui s'est développé premièrement en recherche opérationnelle dans les années 40.

Deux algorithmes : Ford-Fulkerson et Edmonds-Karp.

Entrée.

- Un graphe orienté $G = (V, E)$
- Des capacités sur les arcs : $\forall e \in E, c_e \geqslant 0$
- Deux sommets particuliers : la source $s \in V$, le puits $t \in V$

Hypothèse (mineure). On suppose qu'il n'existe pas d'arc $\cdots \to s$, et qu'il n'existe pas d'arc $t \to \cdots$. On suppose également que toutes les capacités c_e sont entières.

Sortie. Flot $f_e \geqslant 0$ satisfaisant les contraintes :

- Contrainte de capacité : $\forall e, f_e \leqslant c_e$
- Contrainte de préservation de flot : $\forall u \neq s, t$, $\sum_{\cdots \xrightarrow{e} u} f_e = \sum_{u \xrightarrow{e} \cdots} f_e$ (la somme des flots entrant sur un sommet est la somme des flots en sortant).

Objectif. Maximiser la somme $v(f) = \sum_{s \xrightarrow{e} \cdots} f_e$ des flots partant de la source.

Exemple de problème. Cf graphe (1) en annexe.

Déf. On appelle *coupe* une écriture de V comme union disjointe $V = A \sqcup B$, avec $s \in A$ et $t \in B$.

La *capacité* de la coupe est notée $c(A, B) = \sum_{a \in A \xrightarrow{e} b \in B} c_e$.

Lemme. Soit f un flot et (A, B) une coupe, alors $v(f) = f^{\text{sortant}}(A) - f^{\text{entrant}}(A)$, où

$$f^{\text{sortant}}(S) = \sum_{a \in S \xrightarrow{e} b \in V \setminus S} f_e$$

$$f^{\text{entrant}}(S) = \sum_{a \in V \setminus S \xrightarrow{e} b \in S} f_e$$

Applications.

1. Avec la coupe $A = \{s\}, B = V \setminus \{S\}$, on obtient

$$v(f) = f^{\text{sortant}}(\{s\}) - f^{\text{entrant}}(\{s\}) = f^{\text{sortant}}(\{s\})$$

2. Avec la coupe $A = V \setminus \{t\}, B = \{t\}$, on obtient

$$v(f) = f^{\text{sortant}}(V \setminus \{t\}) - f^{\text{entrant}}(V \setminus \{t\}) = f^{\text{sortant}}(V \setminus \{t\}) = f^{\text{entrant}}(\{t\})$$

Preuve du lemme. $v(f) = f^{\text{out}}(\{s\}) - f^{\text{in}}(\{s\})$.

$\forall u \in A \setminus \{s\}$, $0 = f^{\text{out}}(\{u\}) - f^{\text{in}}(\{u\})$

On somme la première ligne et toutes les occurrences de la seconde :

$$v(f) = \sum_{u \in A} f^{\text{out}}(\{u\}) - \sum_{u \in A} f^{\text{in}}(\{u\})$$

Les quatre possibilités d'arêtes comptées ici sont de type $A \to A$ (s'annule), $A \to B$ (compté positivement), $B \to A$ (compté négativement), $B \to B$ (n'apparait pas). On a donc bien

$$v(f) = f^{\text{out}}(A) - f^{\text{in}}(A).$$

Preuve alternative : « ça se voit ».

Lemme. Soit f un flot et (A, B) une coupe, alors $v(f) \leqslant c(A, B)$

Preuve. Par le lemme d'avant, $v(f) = f^{\text{out}}(A) - f^{\text{in}}(A) \leqslant f^{\text{out}}(A) \leqslant c(A, B)$

Théorème (max-flow-min-cut).

$$\max_{f \text{ flot de } s,t} v(f) = \min_{(A,B) \text{ coupe de } s,t} c(A, B)$$

Recherche d'un algorithme

On cherche un algorithme polynomial pour trouver le flot f réalisant ce maximum.

Algorithme 5

$f \leftarrow (\forall e, f_e = 0)$
tant que possible
  améliorer f :
résultat : f

Pour améliorer f, idée :

- trouver un chemin $s \to \cdots \to t$ et faire passer sur ce chemin le flot maximal possible (ie le minimum des capacités des aretes utilisées)
- supprimer la capacité utilisée du graphe (ce faisant, on supprime au moins une arête)
- recommencer.

Pour trouver le flot maximal, idée : choisir à chaque étape d'amélioration le chemin comportant le moins d'arêtes (ie le plus court). Cela est-il suffisant comme condition ? En fait ça ne marche pas. Autre idée qui ne marche pas : prendre le chemin qui permet la plus grosse amélioration.

En fait, l'amélioration n'est pas réalisable par un algorithme glouton : on peut avoir à faire des retours en arrière, ie enlever du flot là où on en a mis avant (l'idée ci-dessus n'est pas suffisante).

Idée : enlever du flot, c'est faire passer du flot en sens inverse, ce que l'on va se permettre de faire. En fait, on rajoute dans le graphe où l'on fait notre recherche pour chaque arc qui est utilisé pour faire passer du flot l'arête dans le sens opposé et avec pour capacité la quantité de flux utilisé sur l'arête dans le bon sens. D'où la définition :

Déf. *Graphe résiduel.* Étant donné $G, s, t, (c_e)$ un problème de flot, et un flot f, on définit le graphe $G_f = (V_f, E_f)$ dit graphe résiduel, avec :

- $V_f = V$
- Pour tout arc $e \in E$ tel que $f_e < c_e$, on met e dans E_f avec pour capacité $c_e^{(f)} = c_e - f_e$
- Pour tout arc $e = (u, v) \in E$ tel que $f_e > 0$, on met l'arc inverse (v, u) dans E_f avec pour capacité $c_{(v,u)}^{(f)} = f_e$.

S'il y a un chemin de s à t dans le graphe résiduel, on peut rajouter du flot. Le parcours d'une arête normale correspond à y rajouter du flot, le parcours d'une arête résiduelle correspond à enlever du flot que l'on a mis à cet endroit là. Il reste à montrer l'implication inverse, que le flot est maximal lorsqu'il ne reste pas de chemin de s à t dans le graphe résiduel.

Algorithme 6

AMÉLIORER(f) (algorithme de Ford-Fulkerson)

1. Construire le graphe résiduel pour le flot f
2. Trouver un chemin simple $P : s \to \cdots \to t$ dans ce graphe
3. L'utiliser pour augmenter le flot par $\min_{e \in P} c_e^{(f)}$

La sortie de cet algorithme est toujours un flot (facile à montrer) ; reste à montrer qu'il est maximal.

Idée : lorsqu'il n'y a plus de chemin dans le graphe résiduel, exhibons une coupe telle que le flot trouvé soit la capacité de cette coupe. Le flot sera alors maximum.

Preuve de corrictitude.

- Le résultat f^* est un flot. (laissé en exo)
- Le flot final est maximal : on sait que $\forall f$ un flot, $\forall (A, B)$ coupe, $v(f) \leqslant c(A, B)$.

  Étant donné f^*, on va exhiber une coupe (A^*, B^*) telle que $v(f^*) = c(A^*, B^*)$; on en déduira que f^* est maximal. En corrolaire, on en déduit le théorème max-flow-min-cut car on a un algorithme permettant de trouver le flux f^*.

  Construction de cette coupe : Dans le graphe résiduel, posons $A^* = \{u : s \to \cdots \to u\}$, et $B^* = V \setminus A^*$. Montrons que la capacité de cette coupe est égale à la valeur du flux :

  Soit $e \in A^* \times B^*$ une arête de G. Pourquoi e n'est-elle pas dans le graphe résiduel ? Car elle était saturée par le flot : $f_e^* = c_e$.

  Donc $f^{*\text{out}}_G(A^*) = \sum_{e \in A^* \times B^*} c_e = c(A^*, B^*)$.

  Soit $e \in B^* \times A^*$ une arête de G, pourquoi e^{renv} n'est-elle pas dans le graphe résiduel G_{f^*} ? Car $f_e^* = 0$. Donc $f^{*\text{in}}_G(A^*) = 0$

  On en déduit : $v(f^*) = f^{*\text{out}}(A^*) - f^{*\text{in}}(A^*) = c(A^*, B^*) - 0 = c(A^*, B^*)$

Analyse de complexité.

Complexité d'une étape d'amélioration. $O(m+n)$ (représentation des graphes avec des listes d'adjacence).

Total. $O(m+n) \times$ #iterations. Au pire des cas, le nombre d'itérations sera la somme des poids sortant de la source ou arrivant au puits, car à chaque itération le flot augmente d'au moins 1 (toutes les capacités sont entières).

$$C = O\left((m+n) \times \sum_{s \xrightarrow{e} \dots} c_e\right)$$

Exemple où Ford-Fulkerson est très lent. Voir dessin (2) : on peut avoir jusqu'à 200 itérations ! Il faut prendre les chemins $s\,u\,v\,t(1), s\,v\,u\,t(1), s\,u\,v\,t(1), s\,v\,u\,t(1), \dots$. Si on s'arrange pour trouver le chemin avec la plus grande capacité à chaque itération (ce qui n'est pas forcément plus coûteux), l'algorithme est meilleur sur cet exemple précis.

Algorithme d'Edmonds-Karp. L'algorithme est globalement le même, mais la recherche du chemin utilisé pour l'augmentation du chemin est faite par un parcours BFS, où l'on choisit le chemin *avec le plus court nombre d'arêtes* dans le graphe résiduel à chaque étape.

Correct. L'algorithme d'Edmonds-Karp est un cas particulier de Ford-Fuklerson, donc il est correct.

Complexité. C'est un $O(m\,n\,(m+n))$:

Déf. Soit $\delta_f(u,v) =$ distance de u à v dans le graphe résiduel G_f

Lemme 1. $\forall v, \delta_f(s,v)$ ne peut pas diminuer lors de l'exécution de l'algorithme (en particulier, on finit sur $\delta_f(s,t) = \infty$).

Déf. Soit $e = (u,v) \in E$. Lors d'une itération $f \to f'$, e est *critique* si e disparait du graphe résiduel (ie $e \in G_f$ mais $e \notin G_{f'}$).

Lemme 2. $\forall e$, e est critique au plus $n/2$ fois lors de l'exécution de l'algorithme.

Preuve du lemme 1. Par l'absurde : soit une itération $f \to f'$ et un sommet v tel que $\delta_f(s,v) > \delta_{f'}(s,v)$ (on choisit v tel que $\delta_{f'}(s,v)$ soit minimum). Soit p un plus court chemin de s à v dans $G_{f'}$. Soit u le prédécesseur de v sur ce chemin : $p : s \to \dots \to u \to v$. $\delta_{f'}(s,v) = \delta_{f'}(s,u) + 1$ puisque p est un plus court chemin.

Pour u la distance n'a pas diminué : $\delta_f(s,v) > \delta_{f'}(s,v) = \delta_{f'}(s,u) + 1 \geqslant \delta_f(s,u) + 1$ (par choix de v). Donc dans G_f, l'arête $u \to v$ n'est pas présente, mais elle est présente dans $G_{f'}$. L'apparition de cette arête ne peut s'expliquer que par le fait que l'on a fait passer du flot dans le sens $v \to u$ lors de l'itération $f \to f'$, c'est-à-dire que cet arc est utilisé dans le chemin q choisi pour l'itération $f \to f'$. Or q est un plus court chemin dans G_f (car on est en train d'appliquer Edmonds-Karp), donc $\delta_f(s,u) = \delta_f(s,v) + 1$. C'est totalement n'importe quoi, donc le lemme est démontré.

Preuve du lemme 2. Soit $e = (u,v) \in E$ un arc du graphe de départ. On regarde une itération où e est critique : $f, G_f, p : e \in G_f$ disparaît. Si cette arête disparaît, c'est qu'on l'a utilisée dans le sens $v \to u$ dans p qui est un plus court chemin, donc $\delta_f(s,v) = \delta_f(s,u) + 1$.

Puis, l'itération où e réaparaît : $f', G_{f'}, p' : e \notin G_{f'}$ réaparaît. e réaparaît car p' utilise l'arc inverse $v \to u$. Or p' est un plus court chemin, donc $\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1$.

En utilisant le lemme 1, qui indique que $\delta_f(s,v)$ ne peut pas diminuer, on a :

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1 \leqslant \delta_f(s,v) + 1 \leqslant \delta_f(s,u) + 2$$

$\delta_f(s,u)$ a augmenté d'au moins 2, or trivialement $\delta_f(s,u) \in \{1, \dots, n, \infty\}$, donc le nombre de cycles de criticité possibles est majoré par $n/2$.

Application des flots au problème du couplage biparti

Le problème du couplage biparti peut s'exprimer comme un problème d'optimisation de flot.

Voir dessin (3) : on oriente tous les arcs de gauche à droite et on leur donne une capacité 1 ; on crée une source s qui est reliée par un arc de capacité 1 à chaque noeud de gauche, et symmétriquement à droite. On calcule le flot maximal. Les arêtes du milieu utilisées (flot utile 1) sont les arêtes du couplages, celles non utilisées (flot utile 0) sont celles qui ne font pas partie du couplage. La capacité 1 sur toutes les arêtes $s \to g$ et $d \to t$ assure que chaque noeud est couplé à au plus un autre noeud (dans ce cas, un flot à valeurs entières est équivalent à un couplage, et $v(f)$ donne le cardinal de ce couplage).

On dit qu'on a réduit le problème de couplage au problème de maximisation du flot.

La borne de Ford-Fulkerson garantit qu'on a une plutôt bonne complexité.

Complément : NP-complétude

Déf. On se restreint aux problèmes de décision, ie aux problèmes tels que sortie $\in$ {vrai, faux}. P = {problèmes solubles en temps polynomial en la taille de l'entrée}. NP = non-déterministe polynomial = {problèmes tels que, si i est une donnée d'entrée telle que la réponse soit vrai, alors on peut vérifier avec de l'aide en temps polynomial que la réponse est vrai}.

On a évidemment $P \subset NP$.

Exemple de problème NP-complet. Graphe hamiltonien (exste-t-il un chemin qui visite tous les sommets d'un graphe une et une seule fois ?). Il n'y a pas d'algo polynomial pour trouver un circuit hamiltonien ou vérifier si un graphe est hamiltonien, mais on sait vérifier polynomialement si un circuit est hamiltonien (on peut donc affirmer que le graphe est hamiltonien). L'oracle est donc un chemin hamiltonien.

Remarque. Le complémentaire n'est pas forcément dans NP ! En effet, quel oracle pourrait nous permettre de prouver qu'un graphe *n'est pas* hamiltonien ?

Problème SAT.

Entrée. Une formule logique de la forme, par exemple :

$$(x_1 \vee \bar{x_3} \vee x_4) \wedge (x_2 \vee x_6 \vee \bar{x_4}) \wedge \cdots$$

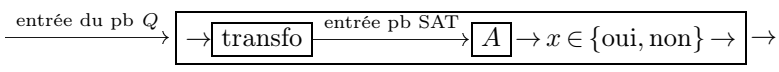
n variables, m clauses.

Sortie. Existe-t-il une valuation des x_i qui rende la formule vrai ?

Ce problème est dans NP : en effet, l'oracle est une valuation qui rend la formule vraie.

Théorème de Cook-Levin (1971). SAT est « le plus difficile » (maximalement difficile) de tous les problèmes de NP. C'est-à-dire : si SAT $\in P$, alors tous les problèmes de NP sont dans P, ie $P = NP$.

Comment le prouver. Par réduction : soit Q un problème de NP. Supposons que SAT $\in P$, alors on a un algorithme A pour SAT. On peut réduire toute entrée de Q à une entrée de SAT :



Déf. Un problème est NP-complet s'il est dans NP et si tous les problèmes de NP se réduisent à lui. On sait donc que SAT est NP-complet.

Question ouverte. $P = NP$?