Algorithmique 3
Notes de cours

IR2 – Université Paris-Est Marne-la-Vallée

Cyril Nicaud

25 novembre 2012

Avant-propos : Ce présent document correspond à des notes sur le cours d'algorithmique 3 en IR2. Il en est dans sa toute première version et contient forcément quelques erreurs (certaines déjà signalées ont été corrigées depuis la version chapitre par chapitre). En cas de doute, c'est le cours la référence, et vous pouvez toujours me demander par email.
Les notes sont succintes et ne contiennent notamment pas tous les développements et les exemples détaillés présentés en cours. Ce document est donc là pour aider, mais ne remplace en rien le cours en lui-même.

CHAPITRE I

Rappels

I.1 Estimation asymptotique

I.1.1 La fonction log

► La fonction logarithme népérien, notée ln ou log, est une fonction continue strictement croissante de $\mathbb{R}_+^*$ dans $\mathbb{R}$. On a $\log(1) = 0$ et $\log(e) = 1$. On a de plus :

$$\lim_{x \to 0^+} \log(x) = -\infty$$
$$\lim_{x \to +\infty} \log(x) = +\infty$$

► La dérivée de $x \mapsto \log x$ est $x \mapsto \frac{1}{x}$. Son inverse est $x \mapsto e^x$.

► Pour tout x, y dans $\mathbb{R}_+^*$, on a $\log(xy) = \log(x) + \log(y)$. Pour tout $x \in \mathbb{R}_*^+$ et pour tout $y \in \mathbb{R}$, on a $\log(x^y) = y \log(x)$; en particulier, $-\log(x) = \log \frac{1}{x}$.

Définition ($\log_b$) Soit $b \in \mathbb{R}_+^*$, le *logarithme base b* est l'application de $\mathbb{R}_+^*$ dans $\mathbb{R}$ définie par

$$\log_b(x) = \frac{\log x}{\log b}.$$

Définition (Base b) Soit $b \geq 2$ un entier. Tout entier $n \geq 1$ s'écrit d'une unique façon sous la forme

$$n = \sum_{i=0}^{k} a_i \, b^i,$$

où $k \in \mathbb{N}$ et les a_i sont dans $\{0, \ldots, b-1\}$ avec $a_k \neq 0$. Les a_i s'appellent l'*écriture de n en base b*.

► On utilise habituellement la base 10. Dans les ordinateurs classiques, les nombres entiers sont représentés en base 2.

Théorème I.1 (Nombre de chiffres) *Soit $n \geq 1$ et $b \geq 2$ deux entiers. Le nombre de chiffres dans l'écriture en base b de n est $\lceil \log_b(n+1) \rceil$, où $\lceil x \rceil$ est la* partie entière supérieure *de x.*

Théorème I.2 (Comparaison de croissance) *Soit $\alpha, \beta > 0$, on a*

$$\lim_{n\infty} \frac{(\log n)^\alpha}{n^\beta} = 0,$$

on dit que "les puissances de n l'emportent sur le logarithme".

I.1.2 Vitesse de croissance

Définition (Ordre de croissance) Soit $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ deux suites réelles strictement positives. On dit que $(f_n)_{n\in\mathbb{N}}$ croît plus lentement que $(g_n)_{n\in\mathbb{N}}$, noté $f_n \ll g_n$ quand

$$\lim_{n\to\infty} \frac{f_n}{g_n} = 0.$$

Théorème I.3 (Transistivité) *La relation $\ll$ sur l'ensemble des suites réelles strictement positives est transitive : si $(f_n)_{n\in\mathbb{N}}$, $(g_n)_{n\in\mathbb{N}}$, $(h_n)_{n\in\mathbb{N}}$ sont telles que $(f_n)_{n\in\mathbb{N}} \ll (g_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}} \ll (h_n)_{n\in\mathbb{N}}$ alors $(f_n)_{n\in\mathbb{N}} \ll (h_n)_{n\in\mathbb{N}}$.*

- On note $\mathbf{1}$ la suite constante égale à 1 : pour tout n, $\mathbf{1}_n = 1$.
- On rapelle que $n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times n$.

Théorème I.4 (Echelle) *Pour tous réels ϵ et c avec $0 < \epsilon < 1 < c$, on a*

$$\mathbf{1} \ll \log n \ll (\log n)^c \ll n^\epsilon \ll n \ll n^c \ll c^n \ll n! \ll n^n \ll c^{c^n}.$$

Théorème I.5 (Inverse) *Soient $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ strictement positives, les suites $(\frac{1}{f_n})_{n\in\mathbb{N}}$ et $(\frac{1}{g_n})_{n\in\mathbb{N}}$ vérifient*

$$f_n \ll g_n \Leftrightarrow \frac{1}{g_n} \ll \frac{1}{f_n}.$$

Théorème I.6 (Constante multiplicative) *Soient $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ telles que $f_n \ll g_n$. Pour toute constante strictement positive c on a $f_n \ll c \cdot g_n$.*

Théorème I.7 (Cas fréquent) *Soient $f_n = n^\alpha (\log n)^\beta$ et $g_n = n^\gamma (\log n)^\delta$, définies pour $n \geq 1$ et $\alpha, \beta, \gamma, \delta \geq 0$. On a*

$$f_n \ll g_n \Leftrightarrow \begin{cases} \alpha < \gamma \\ ou \\ \alpha = \gamma \text{ et } \beta < \delta. \end{cases}$$

I.2 Notation $\mathcal{O}(\ )$

Définition ($\mathcal{O}$) Soient $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ deux suites réelles strictement positives. On dit que $f_n \in \mathcal{O}(g_n)$ quand il existe une constante $C \in \mathbb{R}_+^*$ et $n_0 \in \mathbb{N}$ tels que pour tout $n \geq n_0$ on ait

$$|f_n| \leq C \cdot |g_n|.$$

▶ Comme on est dans le cas de suites strictements positives on pourrait se passer du n_0 dans la définition. On le laisse pour plusieurs raisons, notamment : la définition ainsi formulée s'étend aux suite à valeurs dans $\mathbb{R}$ (avec des valeurs absolues), elle est souvent plus facile à établir, si on a besoin de préciser C (le plus petit possible) il vaut mieux utiliser cette définition, ...

▶ $\mathcal{O}(g_n)$ est donc un ensemble de suites.

Définition (Définition équivalente) Soient $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ deux suites réelles strictement positives. On a $(f_n)_{n\in\mathbb{N}} \in \mathcal{O}((g_n)_{n\in\mathbb{N}})$ si et seulement si la suite $(\frac{f_n}{g_n})_{n\geq 0}$ est bornée.

Théorème I.8 (Convergente $\subset \mathcal{O}(1)$) *Toute suite convergente est dans $\mathcal{O}(1)$. La réciproque est fausse.*

Définition (Ω) On note $f_n \in \Omega(g_n)$ quand il existe un réel $C > 0$ et $n_0 \in \mathbb{N}$ tels que pour tout $n \geq n_0$ on ait $|f_n| \geq C \cdot |g_n|$.

▶ On a donc $f_n \in \mathcal{O}(g_n) \Leftrightarrow g_n \in \Omega(f_n)$.

Définition (Θ) On note $f_n \in \Theta(g_n)$ quand on a à la fois $f_n \in \mathcal{O}(g_n)$ et $f_n \in \Omega(g_n)$.

Définition (Opérations sur des ensembles de suites) Soit $(u_n)_{n\in\mathbb{N}}$ une suite réelle et F et G deux ensembles de suites. On utilise les définitions suivantes :

$$\begin{aligned}
u_n + F &= \{(u_n + v_n)_{n\in\mathbb{N}} \mid (v_n)_{n\in\mathbb{N}} \in F\} \\
u_n \times F &= \{(u_n \times v_n)_{n\in\mathbb{N}} \mid (v_n)_{n\in\mathbb{N}} \in F\} \\
F + G &= \{(u_n + v_n)_{n\in\mathbb{N}} \mid (u_n)_{n\in\mathbb{N}} \in F,\ (v_n)_{n\in\mathbb{N}} \in G\} \\
F \times G &= \{(u_n \times v_n)_{n\in\mathbb{N}} \mid (u_n)_{n\in\mathbb{N}} \in F,\ (v_n)_{n\in\mathbb{N}} \in G\}
\end{aligned}$$

Théorème I.9 (Règles de calcul) *On a les règles de calculs suivantes, pour $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ strictement positives :*

- $f_n \in \mathcal{O}(f_n)$.
- $c \times O(f_n) \subset \mathcal{O}(f_n)$, *pour* $c \mathbb{R}^+$.
- $\mathcal{O}(f_n) \times \mathcal{O}(g_n) \subset \mathcal{O}(f_n \times g_n)$.
- $\mathcal{O}(f_n) + \mathcal{O}(g_n) \subset \mathcal{O}(\max(f_n, g_n))$.
- *Si* $f_n \in \mathcal{O}(g_n)$ *alors* $\mathcal{O}(f_n) + \mathcal{O}(g_n) \subset \mathcal{O}(g_n)$.
- *Si* $f_n \ll g_n$ *alors* $f_n \in \mathcal{O}(g_n)$.

I.3 Complexité d'un algorithme

Important : Cette partie est beaucoup plus de l'informatique que des mathématiques et se prête mal à des notes succinctes comme le reste du cours. En conséquence, il y aura très peu d'informations dans le poly : l'essentiel des considérations du cours **ne sont pas présentes ci-dessous**.

▶ Le but de ce chapitre est de donner des outils pour comparer différentes solutions algorithmiques à un problème donné.

▶ Pour quantifier les performances d'un algorithme on doit se munir d'une notion de *taille* sur les entrées.

▶ Les principales ressources mesurées sont le *temps* (nombre d'instructions utilisées) et l'*espace* (quantité d'espace mémoire nécessaire).

▶ On distingue plusieurs types d'analyses de complexité : l'analyse dans le *meilleur des cas*, le *pire des cas* et *en moyenne*. Pour ce cours on étudie exclusivement le pire des cas. Donc si $T(\mathcal{A})$ est le nombre d'instructions nécessaires pour que l'algorithme fasse ses calculs sur l'entrée $\mathcal{A}$, on s'intéresse à la suite $(t_n)_{n\in\mathbb{N}}$ définie par

$$t_n = \max_{|\mathcal{A}|=n} T(\mathcal{A}),$$

où $|\mathcal{A}|$ est la taille de l'entrée $\mathcal{A}$.

▶ Il n'est souvent pas pertinent d'essayer de quantifier trop précisément t_n, vu qu'on raisonne au niveau de l'algorithme et non d'une implatation. On se contente donc d'estimer t_n avec un ordre de grandeur en Θ ou $\mathcal{O}$. Un résultat typique : la complexité de l'algorithme de tri par insertion est en $\mathcal{O}(n^2)$.

▶ **(ligne la plus effectuée)** La façon la plus simple dévaluer la complexité d'un algorithme est la suivante : un programme est constitué d'un nombre fini de lignes. Appelons-les ℓ_1 à ℓ_k. Soit n_1 ... n_k le nombre de fois qu'elles sont effectuées. La complexité de l'algorithme est $\sum \ell_i n_i$. Soit ℓ_j l'une des lignes qui est effectuée le plus souvent. Si toutes les lignes s'effectuent en temps $O(1)$, la complexité de l'algorithme est majorée par $kn_jO(1)$ soit $O(n_j)$.

▶ Attention aux instructions qui appellent d'autres fonctions et qui ne s'éxécutent pas en temps constant.

▶ On a parfois des boucles imbriquées dont les paramètres dépendent les uns des autres. Par exemple, combien d'étoiles sont affichées, en fonction de n, quand on exécute les instructions suivantes :

Algorithm 1

```
1: for i de 1 à n do
2:     for j de 1 à i do
3:         Print(⋆)
4:     end for
5: end for
```

Pour $i = 1$ il y en a une, pour $i = 2$ il y en a 2, ... En tout il y a donc

$$1 + 2 + \cdots + n = \sum_{i=1}^{n} i$$

étoiles affichées. Ici on peut calculer que le résultat est $\frac{n(n+1)}{2}$, mais on ne peut pas toujours obtenir une formule simple ; on peut néanmoins utiliser le théorème ci-dessous.

Théorème I.10 *Soient $\alpha, \beta \geq 0$, on a*

$$\sum_{i=1}^{n} i^{\alpha} (\log i)^{\beta} = \Theta(n^{\alpha+1} (\log n)^{\beta}).$$

► Le théorème ci-dessus dit que pour ce genre de somme, la majoration par n fois le plus grand terme donne un résultat assez précis (un Θ).

► Attention à ne pas appliquer le théorème à d'autres formes de fonctions. Par exemple

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \neq \Theta(1).$$

On a en fait $H_n = \Theta(\log n)$.

► Quand l'algorithme est récursif c'est plus compliqué. Une bonne façon de commencer l'analyse est de transformer le programme en formule. Pour celà on dit que $T(n)$ est le temps d'exécution dans le pire des cas pour des données de taille n, et on essaye d'exprimer $T(n)$ en fonction de $T(i)$ pour des $i < n$. Par exemple pour le programme suivant :

Algorithm 2

```
1: procedure FIBONACCI(n)
2:     if n ≤ 1 then
3:         return n
4:     else
5:         return Fibonacci(n − 1) + Fibonacci(n − 2)
6:     end if
7: end procedure
```

On obtient l'équation $T(n) = T(n-1)+T(n-2)+\mathcal{O}(1)$ avec certaines conditions initiales en $T(0)$ et en $T(1)$. Le terme $\mathcal{O}(1)$ correspond au temps pour effectuer les instructions qui mènent à faire les deux appels récursifs, et pour effectuer la somme des résultats.

I.4 Structures de données classiques

Reportez-vous à votre cours de l'an dernier. On retiendra que sans hachage (qui pose certains problèmes, notamment le choix de la fonction de hachage), gérer un ensemble de taille n où on veut faire des insertions, des suppressions et des tests d'appartenance, peut se faire avec des structures où les requêtes ont une complexité $\mathcal{O}(\log n)$.

CHAPITRE II

Diviser pour régner

II.1 Présentation de la méthode

La méthode de diviser pour régner est une méthode qui permet, parfois de trouver des solutions efficaces à des problèmes algorithmiques. L'idée est de découper le problème initial, de taille n, en plusieurs sous-problèmes de taille sensiblement inférieure, puis de recombiner les solutions partielles.

L'exemple typique est l'algorithme de *tri fusion* : pour trier un tableau de taille n, on le découpe en deux tableaux taille $\frac{n}{2}$ et l'étape de *fusion* permet de recombiner les deux solutions en $n-1$ opérations. On peut l'écrire ainsi :

Algorithm 3 Tri fusion

```
1: procedure TriFusion(T)
2:     if n ≤ 1 then
3:         return T
4:     else
5:         n = |T|
6:         T_1 = TriFusion(T[0 ... n/2])
7:         T_2 = TriFusion(T[n/2 + 1 ... n − 1])
8:         return Fusion(T_1,T_2)
9:     end if
10: end procedure
```

On va estimer la complexité en comptant le nombre $T(n)$ de comparaisons effectuées par l'algorithme. On a vu qu'on obtient directement que

$$\begin{cases} T(0) = 0, \\ T(1) = 0, \\ T(n) \approx 2\,T(n/2) + n - 1. \end{cases}$$

le $\approx$ est là car il y a des parties entières à considérer pour être rigoureux.

II.2 Forme générale et théorème maître

La forme générale considérée dans ce cours va être :
- **Diviser :** on découpe le problème en a sous-problèmes de tailles $\frac{n}{b}$, qui sont de même nature, avec $a \geq 1$ et $b > 1$.
- **Régner :** les sous-problèmes sont résolus récursivement.
- **Recombiner :** on utilise les solutions aux sous-problèmes pour reconstruire la solution au problème initial en temps $\mathcal{O}(n^d)$, avec $d \geq 0$.

L'équation qu'on aura à résoudre quand on traduit le programme en équation sur la complexité est :

$$\begin{cases} T(1) = \text{constante}, \\ T(n) \approx a\,T\left(\frac{n}{b}\right) + O(n^d). \end{cases}$$

Le théorème maître permet de résoudre ce type d'équations.

Théorème II.1 (Théorème Maître) *On considère l'équation $T(n) = a\,T\left(\frac{n}{b}\right) + \mathcal{O}(n^d)$. Soit $\lambda = \log_b a$. On a les trois cas suivants :*

1. *si $\lambda > d$, alors $T(n) = O(n^\lambda)$;*
2. *si $\lambda = d$, alors $T(n) = O(n^d \log n)$;*
3. *si $\lambda < d$, alors $T(n) = O(n^d)$.*

► Par exemple, pour le tri fusion, on a $a = 2$, $b = 2$, $\lambda = d = 1$ et donc une complexité de $O(n \log n)$.

► En pratique, seuls les cas 1. et 2. peuvent mener à des solutions algorithmiques intéressantes. Dans le cas 3., tout le coût est concentré dans la phase "recombiner", ce qui signifie souvent qu'il y a des solutions plus efficaces.

II.3 Exemples

II.3.1 Dichotomie

Si T est un tableau trié de taille n, on s'intéresse à l'algorithme qui recherche si $x \in T$ au moyen d'une dichotomie. Pour l'algorithme récursif, on spécifie un indice de début d et de fin f, et on recherche si x est dans T entre les positions d et f. L'appel initial se fait avec $d = 0$ et $f = n - 1$. Voir l'algorithme 4 pour la description.

On identifie les paramètres : $a = 1$ car on appelle soit à gauche, soit à droite (ou on a fini, mais on se place dans le pire des cas), $b = 2$ car les sous-problèmes sont de taille $n/2$ et $d = 0$ car on se contente de renvoyer la solution, donc en temps constant. La complexité de la dichotomie est donc $\mathcal{O}(\log n)$.

II.3.2 Exponentiation rapide

Il s'agit de calculer x^n pour x et n donnés, en calculant la complexité par rapport à n. La méthode naïve (multiplier n fois 1 par x) donne une complexité linéaire. On peut faire mieux

Algorithm 4 Dichotomie

```
1: procedure RECHERCHE(T,x,d,f)
2:     if f < d then
3:         return Faux
4:     else
5:         m = ⌊(b+a)/2⌋
6:         if T[m] = x then
7:             return Vrai
8:         else if T[m] < x then
9:             return Recherche(T,x,m + 1,f)
10:        else
11:            return Recherche(T,x,d,m − 1)
12:        end if
13:    end if
14: end procedure
```

en utilisant le fait que

$$\begin{cases} x^0 = 1, & \\ x^n = (x^2)^{\frac{n}{2}} & \text{si n est pair et strictement positif,} \\ x^n = x(x^2)^{\frac{n-1}{2}} & \text{si n est impair.} \end{cases}$$

On peut directement traduire cette constatation en algorithme. Et on retrouve les mêmes pa-

Algorithm 5 Exponentiation rapide

```
1: procedure PUISSANCE(x,n)
2:     if n = 0 then
3:         return 1
4:     else
5:         if n est pair then
6:             return Puissance(x ∗ x, n/2)
7:         else
8:             return x∗Puissance(x ∗ x, (n−1)/2)
9:         end if
10:    end if
11: end procedure
```

ramètres pour le théorème maître que dans le cas de la dichotomie. La complexité de l'exponentiation rapide est donc en $\mathcal{O}(\log n)$.

► L'exponentiation rapide peut être utilisée pour des "multiplications" plus compliquées, comme la multiplication de matrices, la composition de fonctions, ... Dans ces cas, il ne faut pas oublier de compter le coût de la multiplication dans les calculs, qui n'est pas toujours constante.

II.3.3 Algorithme de Karatsuba

► On rappelle qu'un polynôme P est de la forme

$$P(X) = a_0 + a_1X + a_2X^2 + a_3X^3 + \ldots + a_nX^n = \sum_{i=0}^{n} a_iX^i,$$

où les a_i sont appelés les coefficients de P. Attention, il y a $n+1$ coefficients. On peut naturellement représenter P en machine par un tableau de taille $n+1$ et avec $P[0] = a_0$, $P[1] = a_1$,...

► Les polynômes sont abondamment utilisés en informatique. Ils sont par exemple de bons outils pour approximer des fonctions plus complexes.

► Si $P = \sum_{i=0}^{n} a_iX^i$ et $Q = \sum_{i=0}^{n} b_iX^i$, calculer le polynôme $R = P+Q$ est facile, car l'addition des polynômes revient à l'addition deux à deux des coefficients de même rang. On a ainsi

$$P + Q = (a_0 + b_0) + (a_1 + b_1)X + (a_2 + b_2)X^2 + \ldots (a_n + b_n)X^n.$$

On peut donc le calculer en temps $\mathcal{O}(n)$ en faisant une simple boucle.

► La multiplication des polynômes est plus compliquée, si on développe les premiers termes, on a

$$PQ = a_0b_0 + (a_0b_1 + a_1b_0)X + (a_0b_2 + a_1b_1 + a_2b_0)X^2 + \ldots + a_nb_nX^{2n}.$$

La formule général pour le k-ème coefficient c_k de PQ c'est

$$c_k = \sum_{i+j=k} a_ibj.$$

Si on implémente cette règle en algorithme, on obtient une multiplication de polynômes de complexité $\mathcal{O}(n^2)$.

► L'objectif est d'obtenir une multiplication plus rapide. Pour cela on commence à décomposer P et Q en deux polynômes. On écrit[1]

$$P = R \cdot X^{n/2} + S, \qquad Q = T \cdot X^{n/2} + U,$$

où R, S, T et U sont des polynômes de taille $\frac{n}{2}$.

On peut multiplier les deux expressions et on obtient

$$PQ = RT \cdot X^n + (RU + ST) \cdot X^{n/2} + SU.$$

On peut effectuer les 4 produits RT, RU, ST et SU récursivement, puis recombiner en temps linéaire (on a juste à faire des sommes et des décalage (multiplier par X^i c'est décaler de i cases les coefficients). On est dans un cas typique de diviser pour régner, avec les paramètres $a = 4$, $b = 2$ et $d = 1$. Le théorème maître nous donne une complexité de $\mathcal{O}(n^2)$: on n'a rien gagné.

► Pour améliorer la complexité il faut introduire une nouvelle idée. C'est ce qu'a fait Karatsuba en remarquant qu'on peut aussi écrire le produit comme :

$$PQ = RT \cdot X^n + ((R + S)(T + U) - (RT + SU)) \cdot X^{n/2} + SU.$$

Cela semble plus compliqué, mais on remarque qu'on a plus que trois produits plus petits à effectuer (RT, SU et $(R+S)(T+U)$). Le reste se fait en temps linéaire, on a donc les paramètres $a = 3$, $b = 2$ et $d = 1$. Le théorème maître donne une complexité de $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.585})$: on a gagné significativement en efficacité.

[1] Pour cet algorithme on ne s'occupe pas de bien faire les $\frac{n}{2}$ selon la parité de n : on s'autorise à écrire $\frac{n}{2}$ partout. Le traitement rigoureux avec les parties entières ne change pas le résultat.

II.4 Théorème d'Akra-Bazzi (1998)

▶ Dans tous les exemples de ce chapitre on a approximé les formules pour ne pas faire apparaître les parties entières et les légers décalages. Par exemple, si on écrit la formule exacte pour le tri fusion, on obtient que

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n).$$

Les autres algorithmes peuvent donner lieu à des formules encore plus compliquées, avec des -1 en plus dans les appels récursifs.

▶ Heureusement, Akra et Bazzi ont montré une extension du théorème maître qui permet de travailler avec les approximations :

Théorème II.2 *Si on a*

$$T(n) = a\,T\left(\frac{n}{b} + h(n)\right) + O(n^d), \qquad h(n) = O\left(\frac{n}{(\log n)^2}\right)$$

alors le résultat du théorème maître est encore valable.

▶ En particulier, on peut remplacer des quantités comme $\lfloor \frac{n+1}{b} \rfloor - 3$ par $\frac{n}{b}$ et appliquer le théorème maître avec le bon résultat, comme on l'a fait jusqu'ici. Le théorème d'Akra-Bazzi permet de valider mathématiquement les approximations que l'on faisait.

▶ Le vrai théorème d'Akra-Bazzi est encore plus général, puisqu'il permet de résoudre des récurrences avec des a et des b différents, comme par exemple

$$T(n) = 3T\left(\frac{n}{2}\right) + 2T\left(\frac{n}{3}\right) + T\left(\frac{n}{5}\right) + \mathcal{O}(n).$$

Cela dépasse largement le cadre de ce cours, les curieux trouveront plus d'information sur wikipedia.

CHAPITRE III

Programmation Dynamique

III.1 Exemple introductif : la suite de Fibonacci

La suite de Fibonacci est la suite d'entier $(u_n)_{n\geq 0}$ définie récursivement par :

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \quad \forall n \geq 2 \end{cases}$$

On peut traduire directement cette définition en un algorithme récursif :

Algorithm 6 Fibonacci

```
1: procedure FIBONACCI(n)
2:     if n ≤ 1 then
3:         return n
4:     else
5:         return Fibonacci(n − 1) + Fibonacci(n − 2)
6:     end if
7: end procedure
```

Pour analyser la complexité de cet algorithme, on remarque que chaque appel à Fibonacci() se fait en temps constant (si on ne tient pas compte des appels récursifs lancés). Il suffit donc de compter le nombre d'appels, pour n en entrée, que l'on va noter A_n. La suite A_n vérifie :

$$\begin{cases} a_0 = 1 \\ a_1 = 1 \\ a_n = 1 + a_{n-1} + a_{n-2} \quad \forall n \geq 2 \end{cases}$$

Cette suite ressemble à la suite de Fibonacci. On peut l'étudier mathématiquement et on trouve que $A_n \in \Theta(\phi^n)$, où $\phi = \frac{1+\sqrt{5}}{2} \approx 1,6$ est le nombre d'or. La complexité de l'algorithme est donc exponentielle.

D'où vient une telle complexité ? Si on développe le calcul de u_6, on a :

$$u_6 = u_5 + u_4 = u_4 + u_3 + u_3 + u_2 = u_3 + u_2 + u_2 + u_1 + u_2 + u_1 + u_1 + u_0 = \cdots$$

On remarque que les mêmes calculs sont effectués un nombre important de fois.

Pour améliorer la performance de l'algorithme, on peut stocker les résultats intermédiaires obtenus et les réutiliser directement quand on en a besoin au lieu de les recalculer. C'est l'idée de la programmation dynamique. Cela donne :

Algorithm 7 Fibonacci

```
1: procedure FIBONACCI(n,T[ ])
2:     if n ≤ 1 then
3:         return n
4:     end if
5:     if T[n] n'est pas défini then
6:         T[n] = Fibonacci(n − 1) + Fibonacci(n − 2)
7:     end if
8:     return T[n]
9: end procedure
```

La complexité devient alors linéaire : on ne remplit la case $T[n]$ qu'une fois, donc le nombre d'appels à Fibonacci(i), pour chaque valeur $i \leq n$ est d'au plus 3, une fois pour le calculer, une fois quand on calcule Fibonacci($i + 1$) et une fois quand on calcule Fibonacci($i + 2$).

On peut dé-récursiver l'algorithme. Il s'agit de remplir le tableau T directement case par case. On s'aperçoit qu'il faut faire le contraire de ce que fait l'algorithme récursif : commencer par les petites valeurs.

Algorithm 8 Fibonacci

```
1: procedure FIBONACCI(n)
2:     Allouer T avec n + 1 cases
3:     T[0] = 0
4:     T[1] = 1
5:     for i de 2 à n − 1 do
6:         T[i] = T[i − 1] + T[i − 2]
7:     end for
8:     return T[n]
9: end procedure
```

Il est immédiat que l'algorithme est bien linéaire. Il prend aussi une quantité de mémoire linéaire.

La dernière étape consiste à essayer de gagner en mémoire en ne mémorisant que le nécessaire. Pour notre exemple, il suffit de se rappeler des deux derniers termes pour calculer le nouveau. On peut donc n'utiliser qu'un espace mémoire constant (Voir l'algo 9).

III.2 Principe général

On part d'un problème dont on peut trouver une solution optimale à partir des solutions optimales de sous-problèmes du même type. Quand ses sous-problèmes se recouvrent, c'est-à-dire que leurs résolutions ont des calculs en commun, on stocke en mémoire les calculs déjà effectués afin de ne pas avoir à les refaire dans le traitement d'un autre sous-problème.

Pour résoudre un problème en utilisant la programmation dynamique il faut :

Algorithm 9 Fibonacci

```
1: procedure FIBONACCI(n)
2:     Allouer T avec n + 1 cases
3:     a = 0
4:     b = 1
5:     for i de 2 à n − 1 do
6:         c = b
7:         b = a + b
8:         a = b
9:     end for
10:    return b
11: end procedure
```

1. Trouver un découpage en sous-problèmes plus petits, de sorte que les sous-problèmes se recouvrent.
2. Mémoriser les calculs déjà effectués pour ne pas lancer plusieurs fois le même appel.
3. Essayer de dé-récursiver l'algorithme, pour obtenir un algorithme itératif. Il faut en général commencer par les plus petits objets pour reconstruire des objets de plus en plus grands.
4. Essayer de gagner de l'espace mémoire en "jetant" ce qui ne sert plus, ou plutôt en réutilisant l'espace mémoire qui ne sert plus.

Dans le cadre du cours on vous demande surtout de faire 1. et 2., parfois 3.

III.3 Exemple : le sac à dos avec répétitions

On dispose de n types d'objets différents. Le i-ème objet x_i a un poids $\texttt{poids}[i]$ et un prix $\texttt{prix}[i]$. L'endroit contient plein d'exemplaires de chaque type d'objet. On dispose également d'un sac de capacité C, le poids maximal qui peut être dedans.

Le problème est d'optimiser la valeur totale du sac : on met autant d'exemplaire de chaque objet que l'on veut, tant qu'on ne dépasse pas la capacité, et on veut que la somme des prix soit maximale. On considère que toutes les valeurs sont des entiers.

La solution s'obtient en faisant le découpage suivant : tout objet x_i de poids inférieur ou égale à C est placé dans le sac, il reste donc à traiter le sous-problème avec $C - \texttt{poids}[i]$, ce qu'on fait récursivement, en ajoutant le prix de x_i pour avoir la valeur optimale d'un sac contenant x_i. Une fois qu'on a calculé la valeur optimale d'un sac contenant x_1, la valeur optimale d'un sac contenant x_2, ... on prend la plus grande de toute ces valeurs. Mathématiquement, cela donne, en notant $\texttt{Sac}(C)$ la valeur optimale pour une capacité C :

$$Sac(C) = \begin{cases} 0 & \text{si tous les poids sont supérieurs àC,} \\ \max_{i|\texttt{poids}[i]\leq C} \left(Sac(C - \texttt{poids}[i]) + \texttt{prix}[i]\right) & \text{sinon.} \end{cases}$$

On peut traduire directement l'expression mathématique en algorithme récursif, et utiliser la mémorisation sur les valeurs de la capacité pour faire de la programmation dynamique. La complexité du résultat est en $\mathcal{O}(Cn)$.

Quand on dérécursive on obtient l'algorithme ci-dessous.

Algorithm 10 Sac à dos

```
1: procedure SAC(C)
2:     Allouer S avec C + 1 cases
3:     S[0] = 0
4:     for p de 1 à C do
5:         m = 0
6:         for i de 0 à n − 1 do
7:             if poids[i] ≤ p then
8:                 if S[p − poids[i]] + prix[i] ≥ m then
9:                     m = S[p − poids[i]] + prix[i]
10:                end if
11:            end if
12:        end for
13:        T[p] = m
14:    end for
15:    return T[C]
16: end procedure
```

III.4 Exemple : Distance d'édition

Nous développons à présent un exemple classique d'utilisation de la programmation dynamique : le calcul de la *distance d'édition*.

Soient u et v deux mots sur un alphabet A. La distance d'édition entre u et v est le nombre minimum d'opérations à effectuer pour transformer u en v. Les opérations autorisées sont :

- Insertion : on ajoute une lettre de A dans u, où on veut.
- Suppression : on enlève une lettre de u.
- Substitution : on change une lettre de u en une autre lettre (éventuellement la même, ce qui ne change alors pas le mot).

Exemple : $u = verites$ et $v = eviter$. On peut faire la suite minimale de transformations suivante :

$$verites \mapsto erites \mapsto evites \mapsto eviter$$

Comme il n'y a pas de transformation en moins d'étapes (c'est affirmé, non prouvé, mais on peut tester toutes les possibilités), la distance d'édition entre u et v est 3.

Aligments : on représente les opérations effectuées pour passer d'un mot à l'autre par un *alignement*. Voilà un alignement entre *chien* et *niche* :

$$w = \begin{pmatrix} c & h & i & e & n & - & - \\ - & n & i & - & c & h & e \end{pmatrix}$$

Si on regarde colonne par colonne un alignement, on voit qu'on a une suite de transformations qui permettent de passer du mot en haut au mot en bas. Une colonne de la forme $\binom{\alpha}{-}$ correspond à une suppression de α. Une colonne de la forme $\binom{-}{\beta}$ correspond à une insertion de β. Une colonne de la forme $\binom{\alpha}{\beta}$ correspond à une substitution de α en β. Le coût pour passer du premier mot au deuxième en utilisant ces transformations est exactement le nombre de colonnes où les deux lettres (en haut et en bas sont différentes). Sur l'exemple, il y a 7 colonnes, dont une $\binom{i}{i}$ avec deux fois la même lettre, pour un coût total de 6.

Un *alignment optimal* entre u et v est un alignement qui réalise la distance d'édition, c'est-à-dire qui minimise le coût parmi tous les alignements et qui a donc pour coût $d(u, v)$.

Remarque importante : Soit $w = \binom{x\alpha}{y\beta}$ un alignement optimal de dernière lettre $\binom{\alpha}{\beta}$, alors $\binom{x}{y}$ est également un alignement optimal. Cela donne une piste pour la programmation dynamique, puisqu'on peut se ramener à des alignements de mots plus petits.

On distingue trois cas pour un alignement de ua avec vb :

- Si la dernière colonne est $\binom{\alpha}{-}$, cela signifie que la dernière opération effectuée est une suppression. Donc nécessairement on a supprimé la dernière lettre de ua, c'est-à-dire a : on a donc $\alpha = a$. Ce qui reste après, si l'alignement était optimal, c'est un alignement optimal entre u et vb, qui est donc de poids $d(u, vb)$. Le coût total est donc $d(u, vb) + 1$.
- Si la dernière colonne est $\binom{-}{\beta}$, cela signifie que la dernière opération effectuée est une insertion. Donc nécessairement on a inséré la dernière lettre de vb, c'est-à-dire b : on a donc $\beta = b$. Ce qui reste après, si l'alignement était optimal, c'est un alignement optimal entre ua et v, qui est donc de poids $d(ua, v)$. Le coût total est donc $d(ua, v) + 1$.
- Si la dernière colonne est $\binom{\alpha}{\beta}$, avec α et β qui ne sont pas des $-$, cela signifie que la dernière opération effectuée est soit une substitution, soit rien. Le "rien" n'arrive que quand $\alpha = \beta$. Ce qui reste après, si l'alignement était optimal, c'est un alignement optimal entre u et v, qui est donc de poids $d(u, v)$. Le coût total est donc $d(u, v) + \delta_{a,b}$, où $\delta_{a,b}$ vaut 1 si $\alpha \neq \beta$ et 0 sinon.

On en déduit la formule récursive pour $d(ua, vb)$:

$$d(ua, vb) = \min\{d(ua, v) + 1, d(u, vb) + 1, d(u, v) + \delta_{a,b}\}.$$

Il nous manque les conditions aux bords, mais on remarque que $d(\varepsilon, v) = |v|$ (en insérant les lettres de v une par une) et $d(u, \varepsilon) = |u|$ (en supprimant les lettres de u une par une).

Pour les algorithmes ci-dessous, les indices des lettres d'un mot u de taille n vont de 0 à $n-1$ et $u[0, i]$ est le préfixe $u_0u_1 \cdots u_i$ de u de longueur $i+1$ (ou le mot vide si i est négatif). On utilise la notation $u[i]$ pour désigner la lettre u_i. Le mot u est de taille n et le mot v de taille m.

Algorithm 11 Distance d'édition

```
1: procedure D(u,v)
2:     if u = ε then
3:         return m
4:     end if
5:     if v = ε then
6:         return n
7:     end if
8:     if u[n − 1] == v[m − 1] then
9:         δ = 0
10:    else
11:        δ = 1
12:    end if
13:    return min(d(u, v[0, m − 2]) + 1, d(u[0, n − 2], v) + 1, d(u[0, n − 2], v[0, m − 2]) + p)
14: end procedure
```

On retombe donc sur un algorithme récursif avec des sous-problèmes qui se recouvrent, on va stocker dans un tableau les calculs intermédiaires, c'est à dire les distance d'édition de tous les préfixes de u et v. On note $T[][]$ ce tableau ; on stocke dans $T[i][j]$ la distance d'édition entre le préfixe de longueur i de u et celui de longueur j de v. Plutôt que de faire des appels récursifs, on va remplir T dans l'ordre de la taille des préfixes.

Algorithm 12 Distance d'édition itératif

```
1: procedure D(u,v)
2:     for i de 0 à n do
3:         T[i][0] = i
4:     end for
5:     for j de 0 à m do
6:         T[0][j] = j
7:     end for
8:     for i de 1 à n do
9:         for j de 1 à m do
10:            if u[i − 1] == v[j − 1] then
11:                δ = 0
12:            else
13:                δ = 1
14:            end if
15:            T[i][j] = min(T[i][j − 1] + 1, T[i − 1][j] + 1, T[i − 1][j − 1] + δ)
16:        end for
17:    end for
18:    return T[n][m]
19: end procedure
```

La complexité est en $\mathcal{O}(n \times m)$ en temps et en espace. On remarque qu'on peut faire le calcul en ne gardant en mémoire que deux lignes ou deux colonnes (puisqu'on ne regarde que dans la colonne d'avant et la ligne d'avant), ce qui permet de ne stocker que $\mathcal{O}(n)$ valeurs.

Exemple : On prend $u = aaba$ et $v = bab$:

	i	0	1	2	3	4
j			a	a	b	a
0		0	1	2	3	4
1	b	1	1	2	2	3
2	a	2	1	1	2	2
3	b	3	2	2	1	2

CHAPITRE IV

Introduction à la Théorie des Graphes

IV.1 Terminologie

Informellement un graphe est constitué de ronds (les sommets) et de flêches qui relient deux ronds (les arcs). Ils sont omniprésents en informatique aussi bien théorique que pratique. Le but de ce cours est d'étudier des algorithmes pour résoudre des problèmes classiques qui peuvent se poser sur ce type de structures.

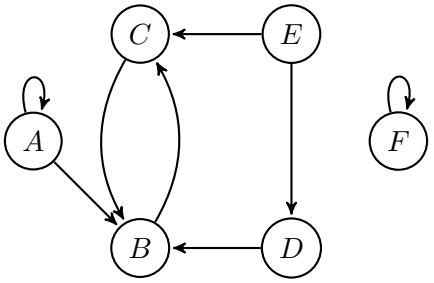


FIG. IV.1 – Un graphe avec 6 sommets et 8 arcs (dont 2 boucles).

Un couple $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ est un *graphe orienté* quand $\mathcal{S}$ un ensemble fini et $\mathcal{A}$ un sous-ensemble de $\mathcal{S} \times \mathcal{S}$. $\mathcal{S}$ est l'ensemble des *sommets* et $\mathcal{A}$ est l'ensemble des *arcs*. Si $a = (s, t) \in A$ on dit que l'arc à pour début s et pour fin t, ce qu'on note respectivement $d(a)$ et $f(a)$. Un arc tel que $d(a) = f(a)$ est appelé une *boucle*. On représente graphiquement le graphe par des ronds pour les sommets et des flêches reliant les ronds pour les arcs. Soit s un sommet. Tout sommet t tel que $(s, t) \in \mathcal{A}$ est un *successeur* de s.

Un chemin C dans un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ est une suite $(a_1, \cdots, a_n)$ telle que $f(a_i) = d(a_{i+1})$ pour $i \in \{1, \cdots, n-1\}$. On dit que le chemin va de $d(a_1)$ à $f(a_n)$. Si $d(a_1) = f(a_n)$ on dit que le chemin est un *circuit* ou un *cycle*. Un somme t est dit *accessible depuis* s quand il existe un chemin de s à t. Un graphe est dit *fortement connexe* quand tout sommet est accessible depuis tout autre.

Si pour tout $(s, t) \in \mathcal{A}$ alors (t, s) est aussi dans $\mathcal{A}$ on dit que le graphe est *non-orienté*. Graphiquement on fait juste un trait entre s et t plutôt que les deux flêches. Les doubles-arcs s'appellent des *arêtes*.

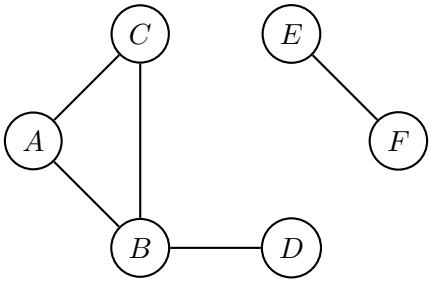


FIG. IV.2 – Un graphe non-orienté avec 6 sommets et 5 arêtes (sans boucle).

Un graphe est *valué* quand on attache un poids (souvent à valuers dans $\mathbb{R}$) à chaque arc. Dans le cas non-orienté, il faut que le poids soit le même dans les deux sens.

IV.2 Exemples de modélisation

Les graphes sont utilisés pour modéliser de multiples situations aussi bien pratiques que théoriques. Voilà quelques exemples :

1. **Métro :** les sommets sont les stations et les arcs les tronçons de ligne entre deux stations.
2. **Plan :** les sommets sont, par exemple, les villes et les arcs les autoroutes. Il peut être valué par la distance ou par le prix du payage.
3. **Web :** les sommets sont les pages html et les arcs sont les hyperliens.
4. **Graphe d'un programe :** les sommets sont les fonctions, et on met un arc entre f et g si la fonction f appelle la fonction g.
5. **Automates finis :** les sommets sont les états et les arcs sont les transitions, valuées par les lettres de l'alphabet.

IV.3 Représentation en machine

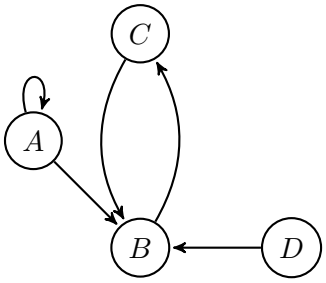
Pour tout ce cours on considère que l'ensemble des sommets ne change pas. Cette spécificité influe sur le choix des structures de données pour réprésenter le graphe. Informellement, un graphe est *peu dense* quand le nombre d'arcs A est très petit devant S^2, où S est le nombre de sommets. La plupart des graphes rencontrés en pratique sont peu dense, on va donc exprimer les complexités avec deux paramètres, le nombre d'arcs A et le nombre de sommets S, pour en tenir compte.

IV.3.1 Matrice d'adjacence

On numérote les sommets de 0 à $S-1$ et on fait un tableau M de taille $S \times S$ avec en ligne i colonne j la valeur M_{ij} avec

$$M_{ij} = \begin{cases} 1 & \text{si } (i,j) \in \mathcal{A} \\ 0 & \text{sinon} \end{cases}$$

Par exemple :



La matrice associée est :

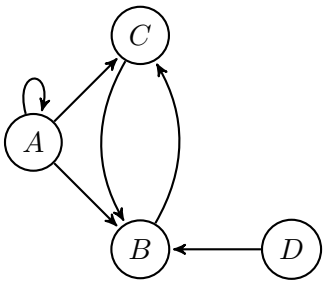
$$M = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

La taille de la représentation est facile à calculer, il faut exactement S^2 bits, donc la complexité en espace est $\mathcal{O}(S^2)$.

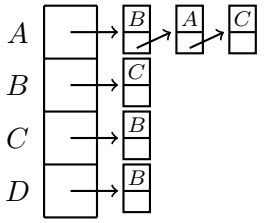
IV.3.2 Listes d'adjacence

L'autre représentation classique d'un graphe en machine est d'utiliser un tableau de listes. Chaque case du tableau est indicée par un des sommets s du graphe, et contient la liste (non nécessairement ordonnée) des successeurs du sommet s.

Par exemple :



Le tableau de listes associé est :



En mémoire il faut exactement un maillon de chaîne par case, soit A maillons en tout, plus la place pour allouer le tableau de taille S. **On considère dans ce cours** qu'on a toujours un nombre d'arcs non négligeable devant le nombre de sommets, ce qui est réaliste pour les applications, les graphes avec très peu d'arcs ayant un intérêt algorithmique limité. On peut donc dire que la taille de la représentation par listes d'adjacence est $\mathcal{O}(A)$.

En remarque, la représentation par listes d'adjacence semble adaptée aux graphes peu dense, consommant moins de mémoire que la représentation par matrice.

IV.3.3 Quelques complexités

Voilà quelques opérations de base avec leurs complexités respectives. Il se trouve que pour presque tous les algorithmes considérés dans ce cours, on aura principalement besoin de la dernière opération, ce qui rend la représentation par listes la plus adaptée dans la majorité des cas.

Opération	**Matrice**	**Listes**
Tester s'il y a un arc $x \to y$	$\mathcal{O}(1)$	$\mathcal{O}(S)$
Tester si x a au moins un successeur	$\mathcal{O}(S)$	$\mathcal{O}(1)$
Tester si x a au moins un prédecesseur	$\mathcal{O}(S)$	$\mathcal{O}(A)$
Lister tous les arcs	$\mathcal{O}(S^2)$	$\mathcal{O}(\mathcal{A})$

CHAPITRE V

Parcours de graphes

V.1 Rappel sur les arbres

Un parcours en profondeur d'un arbre peut s'effectuer simplement de façon récursive, en appelant la fonction suivante depuis la racine de l'arbre.

Algorithm 13 Parcours d'arbre

```
1: procedure PP(𝒩)
2:     (traitement préfixe de 𝒩)
3:     for ℳ fils de 𝒩 do
4:         PP(ℳ)
5:     end for
6:     (traitement postfixe de 𝒩)
7: end procedure
```

Si l'arbre est binaire et qu'on effectue un traitement entre le fils droit et le fils gauche, on parle de traitement infixe.

La complexité est $O(n)$, où n est le nombre de nœuds, puisqu'on fait exactement un appel par nœud et que chaque appel à une complexité le nombre de fils. Mais la somme sur tous les nœuds du nombre de fils c'est en $O(n)$ (cela vaut tous les nœuds sauf la racine.

Algorithm 14 Parcours d'arbre avec pile ou file

```
1: procedure PP(𝒩)
2:     todo = {𝒜}
3:     while todo ≠ ∅ do
4:         𝒩 = Extraire de todo
5:         for ℳ fils de 𝒩 do
6:             Ajouter ℳ dans todo
7:         end for
8:     end while
9: end procedure
```

On peut éviter la récursivité en utilisant une pile, si on change la pile en une file sans rien changer d'autre, on obtient un parcours en largeur de l'arbre : si on définit la distance d'un noeud à la racine comme étant le nombre minimum d'arêtes à traverser pour aller de la racine à ce noeud, on traite les sommets dans l'ordre de la distance à la racine. Ces versions sont également linéaires.

V.2 Parcours en profondeur d'un graphe

Si on fait la même chose que sur les arbres, il y a un problème car on peut avoir une boucle infinie à cause des cycles. On va donc marquer les sommets déjà visité pour éviter ça.

Par définition, le parcours d'un graphe consiste à visiter tous les sommets. On a donc deux fonctions : une qui fait un parcours récursif depuis un sommet donné, et une qui l'appelle pour tous les sommets non déjà visités. Le tableau `Visité` permet de marquer les sommets qu'on a déjà visités pour éviter de les traiter à nouveau.

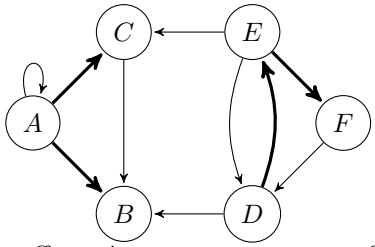
Algorithm 15 Parcours en profondeur

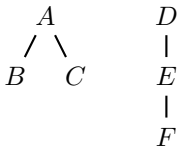
```
1: procedure PP(𝒢)
2:     for s ∈ 𝒮 do
3:         Visité[s] = Faux
4:     end for
5:     for s ∈ 𝒮 do
6:         if Visité[s] == Faux then
7:             PPrec(𝒢,s,Visité)
8:         end if
9:     end for
10: end procedure

11: procedure PPrec(𝒢,s,Visité)
12:     Visité[s] = Vrai
13:     for t successeur de s do
14:         if Visité[t] == Faux then
15:             PPrec(𝒢,t,Visité)
16:         end if
17:     end for
18: end procedure
```

Pour analyser la complexité, on remarque que PPrec est appelée exactement une fois par sommet, et que son coût est le nombre de successeurs du sommet d'appel. La complexité en listes d'adjacence est donc proportionnelle à la somme sur tous les sommets du nombre de successeurs. Mais cette quantité vaut exactement A. Pour les matrices, pour trouver les successeurs il faut parcourir toute la ligne, ce qui coûte S à chaque fois. En conclusion la complexité du parcours en profondeur est $\mathcal{O}(A)$ pour les listes et $\mathcal{O}(S^2)$ pour les matrices.

V.3 Arborescence et classification des arcs

Lorsqu'on appelle `PP(G,s,visité)` depuis un sommet s dans la boucle de `PP(G)` on construit un arbre, appelé l'arbre de parcours en profondeur depuis s, qu'on définit récursivement par :





On a effectué un parcours en profondeur sur le graphe à gauche en utilisant l'ordre alphabétique à chaque fois qu'il y avait un choix à faire. En gras sont indiquées les arcs qui ont servi à découvrir de nouveaux sommets. La forêt du parcours est indiquée à droite.

FIG. V.1 – Exemple de parcours en profondeur

s est la racine, si t_1,...,t_k sont les successeurs non visités d'un sommet t au moment où ils sont considérés par PPrec, alors ce sont les fils de t dans l'arbre.

L'arbre retrace les appels récursifs effectués. Quand la boucle principale rappelle `PPrec` avec un autre sommet, on obient un autre arbre. A la fin on a une forêt, appelée la forêt du parcours en profondeur.

Remarque importante : il n'y a pas unicité du parcours, ça dépend dans quel ordre on considère les sommets. Il n'y a donc pas non plus unicité de la forêt du parcours, le nombre d'arbres peut même changer d'un parcours en profondeur à l'autre.

Pour un parcours en profondeur donné, on considère 4 types d'arcs dans le graphe :

- les arcs d'arbres : ce sont ceux qui relient un sommet à un de ses fils dans l'arbre.
- les arcs arrières : qui relient un sommet avec lui-même ou un de ses ancètres dans l'arbre.
- les arcs avant : qui relient un sommet avec un descendant (qui n'est pas un fils) dans l'arbre.
- les arcs transverses : tous les autres.

Pour ce qui va nous concerner, les notions importantes sont celles d'arc d'arbre et d'arc arrière.

V.4 Parcours en profondeur avec couleurs

On s'intéresse à détecter les arcs arrières. Si t est un descendant de s dans la forêt, alors t est traité entre le début et la fin du traitement de s, d'après le code récursif. Par conséquent, s'il y a un arc arrière $t \to s$ il relie un sommet t avec un sommet en cours de traitement s. On va donc modifier l'algorithme pour savoir quand un sommet est en cours de traitement.

On utilise un code de couleur pour raffiner le tableau Visité. On utilise la convention :

- Couleur[s] = Vert : quand s n'est pas encore visité ;
- Couleur[s] = Bleu : quand s est en cours de visite ;
- Couleur[s] = Rouge : quand le traitement de s est terminé.

On détecte donc un arc arrière quand on voit un successeur de couleur Bleu.

Algorithm 16 Parcours en profondeur coloré

```
1: procedure PP(G)
2:     for s ∈ S do
3:         Couleur[s] = Vert
4:     end for
5:     for s ∈ S do
6:         if Couleur[s] == Vert then
7:             PPrec(G,s,Couleur)
8:         end if
9:     end for
10: end procedure

11: procedure PPrec(G,s,Couleur)
12:     Couleur[s] = Bleu
13:     for t successeur de s do
14:         if Couleur[t] == Bleu then
15:             Signaler arc arrière s → t
16:         end if
17:         if Couleur[t] == Vert then
18:             PPrec(G,t,Couleur)
19:         end if
20:     end for
21:     Couleur[s] = Rouge
22: end procedure
```

V.5 Parcours en profondeur avec compteurs

Quand on va vers un sommet rouge dans l'algorithme précédant, cela peut signaler aussi bien un arc avant qu'un arc transverse. Un moyen de différencier les deux et de marquer la date de début et de fin de traitement avec l'algorithme suivant. On utilise un tableau Début[] pour noter la date de début de traitement d'un sommet et Fin[] pour noter sa date de fin de traitement. L'avancée du temps est simulée par un compteur (on a une variable globale Compteur pour ça).

La complexité ne change pas, c'est toujours $\mathcal{O}(A)$ pour la représentation par listes et $\mathcal{O}(S^2)$ pour celle par matrice.

Remarque : on peut dessiner le diagramme temporel du parcours sur lequel on retrouve la forêt. De plus on ne peut jamais avoir

```
Début[s] < Début[t] et Fin[s] < Fin[t]  (impossible)
```

à cause de la structure des appels récursifs.

Maintenant on peut completement caractériser les types d'arcs $s \to t$ au moment où on les rencontre :

- Si Début$[t] = \emptyset$ c'est un arc d'arbre.
- Sinon si Fin$[t] = \emptyset$ c'est un arc arrière.
- Sinon si Début$[t] >$ Début$[s]$ c'est un arc avant.
- Sinon (cas Début$[t] <$ Début$[s]$) c'est un arc transverse.

Et on peut modifier l'algorithme pour signaler les types d'arcs.

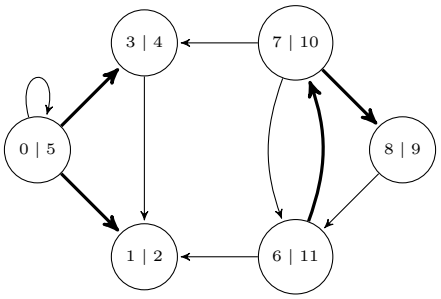


FIG. V.2 – Parcours en profondeur avec dates de début / fin. Dans chaque sommet est indiqué la date début de traitement (à gauche) et la date de fin de traitement (à droite)

Algorithm 17 Parcours en profondeur temporisé

```
1: procedure PP(G)
2:     Compteur = 0
3:     for s ∈ S do
4:         Début[s] = Fin[s] = ∅
5:     end for
6:     for s ∈ S do
7:         if Début[s] == ∅ then
8:             PPrec(G,s,Début,Fin)
9:         end if
10:    end for
11: end procedure

12: procedure PPrec(G,s,Début,Fin)
13:    Début[s] = Compteur
14:    Compteur ++
15:    for t successeur de s do
16:        if Début[s] == ∅ then
17:            PPrec(G,t,Début,Fin)
18:        end if
19:    end for
20:    Fin[s] = Compteur
21:    Compteur ++
22: end procedure
```

V.6 Théorème du chemin vert

Ce résultat est pratique pour raisonner sur ce qui se passe lors d'un parcours en profondeur. On rappelle qu'on ne maîtrise *a priori* pas grand chose sur l'ordre dans lequel vont être traités les successeurs d'un sommet.

Théorème du chemin vert : On considère un graphe qui est parcouru en profondeur et tel qu'à l'instant $d[s]$ on commence le traitement du sommet s (il passe donc bleu) et qu'il existe un chemin composé uniquement de sommets verts allant de s à t (t est vert aussi). Alors, t sera traité avant la fin du traitement de s. Autrement dit, c'est un descendant de s dans l'arborescence de parcours en profondeur.

Esquisse de preuve : par récurrence sur la longueur du chemin. Si le chemin est de longueur 1 ; t est successeur de s, au moment où on le selectionne dans la liste des successeurs, soit il a déjà été traité et est descendant d'un des fils de s, soit on commence (et termine) le traitement : dans les deux cas c'est un descendant de s.

Si la propriété est vraie pour des chemins de longueur au plus n, pour un chemin de longueur $n+1$ on regarde le premier sommet vert x qui est traité parmi ceux du chemin. Cela forme un chemin vert au temps $d[x]$ qui est plus court et qui va aussi en t cela permet de conclure par récurrence.

CHAPITRE VI

Applications du parcours en profondeur

VI.1 Graphe de contraintes d'ordre

Soit E un ensemble fini et $R \subset E \times E$ un ensemble de couples d'éléments qu'on voit comme des contraintes d'ordre : si $(e, f) \in R$ cela veut dire que la tâche e doit nécessairement être effectuée avant la tâche f.

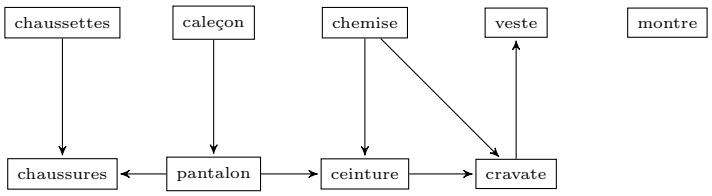
Le *graphe de contraintes d'ordre* de R est le graphe d'ensemble de sommets E et avec un arc entre s et t si et seulement si $(s, t) \in R$.

Exemple :

avant	chaussettes	caleçon	pantalon	pantalon	chemise	chemise	ceinture	cravate
après	chaussures	pantalon	ceinture	chaussures	ceinture	cravate	veste	veste

Il y a aussi une montre, sans contrainte d'ordre dessus.

Le graphe associé est



On distingue deux types de problèmes algorithmiques :
– Est-ce qu'il y a un ordre possible sur les sommets qui respecte les contraintes ?
– Comment trouver efficacement un tel ordre quand il existe ?
Comme application on a notamment la répartition de tâches sur un système avec plusieurs processeurs.

VI.2 Détection de circuits

Pour répondre à la question "existe-t-il un ordre qui respecte les contraintes" ? on remarque que c'est le cas si et seulement si le graphe est sans circuit : il n'existe pas de sommet s tel qu'il

y ait un chemin non vide de s à s. Un graphe sans circuit est dit *acyclique*, et un graphe qui possède au moins un circuit est dit *cyclique*.

Le problème revient donc à tester l'existance d'un circuit dans un graphe. On le résoud grâce au théorème suivant.

Théorème VI.1 *Un graphe possède un circuit si et seulement si tout parcours en profondeur du graphe contient un arc arrière.*

Démonstration :

• S'il y a un circuit

$$s = s_0 \to s_1 \to \cdots \to s_{n-1} \to s_n = s$$

soit t le premier sommet traité lors du parcours en profondeur. A l'instant $d[t]$ il y a un un chemin vert de t à son prédecesseur t' dans le cycle. Donc t' est un descendant de t et l'arc $t' \to t$ est un arc arrière.

• S'il y a un arc arrière $s \to t$ alors on voit le circuit directement sur l'arborescence de parcours : par définition t est un ancêtre de s donc il y a un chemin de t à s ; on referme le circuit avec l'arc arrière. □

La complexité est la même que pour un parcours en profondeur : $O(A)$ listes et $O(S^2)$ matrice.

VI.3 Tri toplogique

On cherche maintenant à ordonner les sommets de façon à respecter les contraintes. Pour cela on réalise un parcours en profondeur en notant les dates de fin de traitement. Une solution est de trier les sommets par fin de traitement décroissant.

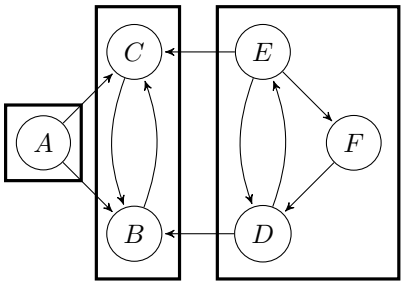
En effet, au moment où un arc $s \to t$ est exploré, le sommet t ne peut pas être bleu, sinon c'est un arc arrière. S'il est vert, t est un descendant de s et donc $f[t] < f[s]$. S'il est rouge, le traitement est terminé et on a aussi $f[t] < f[s]$. L'inégalité est vraie dans les deux cas.

Pour ne pas avoir à effectuer un tri à la fin, il suffit de les insérer en tête de liste en fin de traitement. Cela donne l'algorithme suivant, de même complexité qu'un parcours en profondeur.

VI.4 Composantes fortement connexes (CFC)

Définition : on dit que s et t sont dans la même composante fortement connexe quand il existe un chemin de s à t et de t à s. On s'autorise les chemins de longueur nulle : s est toujours dans la même CFC que s.

Une CFC est donc un ensemble maximal de sommets tel qu'on peut toujours aller de l'un d'eux à un autre.



Un graphe orienté, dont les composantes fortement connexes sont encadrées.

Algorithm 18 Test d'acyclicité

```
1: procedure testAcyclic(𝒢)
2:     for s ∈ 𝒮 do
3:         Couleur[s] = Vert
4:     end for
5:     for s ∈ 𝒮 do
6:         if Couleur[s] == Vert then
7:             if arcArriere(𝒢,s,Couleur) == Vrai then
8:                 return Faux
9:             end if
10:        end if
11:    end for
12:    return Vrai
13: end procedure

14: procedure arcArriere(𝒢,s,Couleur)
15:    Couleur[s] = Bleu
16:    for t successeur de s do
17:        if Couleur[t] == Bleu then
18:            return Vrai
19:        end if
20:        if Couleur[t] == Vert then
21:            if arcArriere(𝒢,t,Couleur) == Vrai then
22:                return Vrai
23:            end if
24:        end if
25:    end for
26:    Couleur[s] = Rouge
27:    return Faux
28: end procedure
```

Algorithm 19 Tri topologique

```
1: procedure TriTopo(G)
2:     L = Liste vide
3:     for s ∈ S do
4:         Visité[s] = Faux
5:     end for
6:     for s ∈ S do
7:         if Visité[s] == Faux then
8:             TriTopoRec(G,s,Visité,L)
9:         end if
10:    end for
11:    return L
12: end procedure

13: procedure PPrec(G,s,Visité,L)
14:    Visité[s] = Vrai
15:    for t successeur de s do
16:        if Visité[t] == Faux then
17:            TriTopoRec(G,t,Visité,L)
18:        end if
19:    end for
20:    Ajouter s en début de L
21: end procedure
```

Il est souvent utile de découper un graphe en CFC et de raisonner sur le "graphe des CFC", où on a fusionné tous les sommets d'une même composante en un seul sommet. Cela donne une vision stratégique des déplacements dans le graphe.

L'algorithme "magique" pour calculer les CFC est le suivant :

```
CFC(G)
* Faire un PP en notant les dates de fin
* Transposer le graphe
* Faire un PP en utilisant l'ordre décroissant des dates de fin
```

La propriété qui fait que ça fonctionne est la suivante : si s est le dernier sommet terminé dans le premier parcours et que t est dans l'arbre de s pour le deuxième, alors il existe un chemin de s à t dans G^t, et donc un chemin de t à s dans S. De plus $f[t] < f[s]$ car $f[s]$ maximal. Supposons par l'absurde que le premier sommet x de ce chemin, dans le premier parcours, n'est pas s. Alors il existe un chemin vert de x à s, impossible car $f[x] < f[s]$. Donc le premier rencontré est s et $d[s] < d[t] < f[t] < f[s]$ par suite t est un descendant de s donc accessible de puis s. Donc t et s sont dans la même CFC. On calcule ainsi toute la CFC de s. On l'enlève et recommence pour finir la preuve.

CHAPITRE VII

Plus courts chemins

VII.1 Présentation du problème

VII.1.1 Graphes pondérés

On s'intéresse à des graphes qui sont pondérés et a priori orientés. Il y a donc un *poids*, un réel, sur chaque arc qui peut représenter, par exemple le coût pour l'emprunter. On note $p(\alpha)$ le poids de l'arc α.

Le poids d'un chemin $C = \alpha_1, \dots \alpha_n$, est défini comme étant la somme des poids de ses arcs :

$$p(C) = p(\alpha_1) + \ldots + p(\alpha_n) = \sum_{i=1}^{n} p(\alpha_i).$$

Pour la représentation par matrice on adopte la convention suivante : s'il y a un arc entre i et j de poids p, on met p dans la matrice en ligne i / colonne j. S'il n'y a pas d'arc on met le symbole ∞. Pour la représentations par listes, il faut maintenant dans chaque maillon de chaîne indiquer le poids en plus de la destination.

VII.1.2 Plus courts chemins

Si s et t sont deux sommets du graphe, un *plus court chemin* entre s et t est un chemin allant de s à t de poids minimal.

Attention : il n'y a pas nécessairement unicité du plus court chemin entre deux sommets.

Attention : la notion de plus court chemin n'est pas toujours bien définie. Il faut qu'il n'y ait pas de circuit de poids négatif dans le graphe. Un tel circuit peut être emprunté un nombre arbitraire de fois, faisant baisser le poids à chaque nouveau tour.

Le théorème suivant est simple mais fondamental ; il est utilisé dans tous les algorithmes de calcul plus courts chemins. Si α est un arc, on note $d(\alpha)$ le sommet de départ de α et $f(\alpha)$ le sommet d'arrivée (de fin).

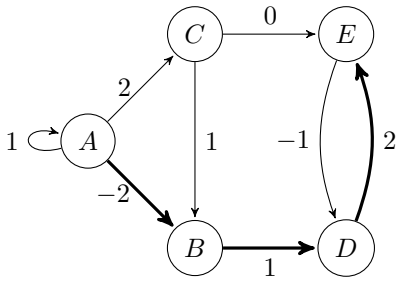


FIG. VII.1 – Un exemple de graphe pondéré. Le plus court chemin pour aller de A à E est de poids $1 = -2 + 1 + 2$ et est indiqué en gras.

Théorème VII.1 (Découpage) *Soit $\mathcal{G}$ un graphe pondéré sans circuit de poids négatif. Si $C = \alpha_1, \ldots, \alpha_n$ est un plus court chemin de $d(\alpha_1)$ à $f(\alpha_n)$, alors pour tout $i \in \{1, \ldots, n-1\}$, $\alpha_1, \ldots, \alpha_i$ est un plus court chemin de $d(\alpha_1)$ à $f(\alpha_i)$ et $\alpha_{i+1}, \ldots, \alpha_n$ est un plus court chemin de $d(\alpha_{i+1})$ à $f(\alpha_n)$.* "Si on coupe un plus court chemin en deux, on obtient deux plus courts chemins".

Démonstration : Par l'absurde, si $\alpha_1, \ldots, \alpha_i$ n'est pas un plus court chemin, alors il existe un chemin strictement plus court C' pour aller de $d(\alpha_1)$ à $f(\alpha_i)$. Mais alors C' suivi de $\alpha_{i+1}, \ldots, \alpha_n$ serait un chemin plus court que C avec même début et fin : c'est impossible. On procède de même pour $\alpha_{i+1}, \ldots, \alpha_n$. □

On se sert souvent du théorème de découpage avec comme partie droite juste un arc ($i = n$) : cela permet de dire que si on enlève le dernier arc dans un plus court chemin, on a encore un plus court chemin, et de faire des formules récursives (et de la programmation dynamique). Cela donne également un procédé simple pour stocker les chemins, comme on le verra dans la suite.

L'autre théorème important de cette partie permet de limiter la longueur des chemins considérés dans les algorithmes :

Théorème VII.2 (Longueur des plus courts chemins) *Soit $\mathcal{G}$ un graphe pondéré sans circuit de poids négatif, et soient s et t deux sommets de $\mathcal{G}$ tel que t est accessible depuis s. Parmi les plus courts chemins entre s et t, il y en a au moins un de longueur (nombre d'arcs) inférieure ou égale à $S - 1$.*

Démonstration : Soit C un plus court chemin de s à t de longueur minimale parmi les plus courts chemins de s à t. S'il est de longueur inférieure ou égale à $S - 1$, c'est gagné. Supposons par l'absurde qu'il est de longueur $\geq S$. Alors le chemin C possède au moins S arcs, et passe donc par au moins $S + 1$ sommets. Par suite, il existe un sommet x par lequel il passe deux fois : il fait donc un circuit sur x. Si on enlève ce circuit du chemin, on obtient un chemin strictement plus court. Mais c'est encore un plus court chemin entre s et t car le circuit est de poids positif. On a donc trouvé un chemin de poids minimal plus court entre s et t, ce qui contredit les hypothèses sur C. □

VII.1.3 Problèmes de plus courts chemins

L'objectif de ce chapitre c'est de calculer des plus courts chemins dans un graphe. On distingue trois types de problèmes :

1. pour s et t fixés, calculer un plus court chemin entre s et t ;
2. pour s fixé, calculer un plus court chemin entre s et t pour tous les sommets t ;
3. calculer un plus court chemin entre s et t pour tous les couples de sommets (s, t).

On verra que d'un point de vue complexité, on ne sait pas résoudre 1 plus efficacement que 2 : les algorithmes pour les problèmes de type 1 sont en réalité des algorithmes de type 2 (où éventuellement on s'arrête avant d'avoir tout calculé, mais ça ne fait pas gagner en complexité dans le pire des cas).

VII.2 Algorithme de Bellman-Ford (début des années 60)

On va proposer une première solution pour les problèmes de type 2.

D'abord, d'après le théorème sur la longueur, les plus courts chemins sont de longueur au plus $S-1$. On note $d(s_0, t, \ell)$ la longueur d'un plus court chemin entre s_0 et t qui utilise au plus ℓ arcs. La solution cherchée est $d(s_0, t, S-1)$.

On utilise le théorème de découpage : pour trouver $d(s_0, t, \ell)$, avec $\ell \geq 1$, il suffit de regarder les chemins les plus courts de longueurs au plus $\ell - 1$ et d'y ajouter éventuellement un arc menant à t. Formellement :

$$d(s_0, t, \ell) = \min \Big\{ d(s_0, t, \ell - 1),\ \min_{x \to t} \big(d(s_0, x, \ell - 1) + p(x \to t) \big) \Big\}$$

C'est donc un problème de programmation dynamique : on s'est ramené à un problème du même type avec des longueurs plus petites. L'initialisation est simple : $d(s_0, s_0, 0) = 0$ (on reste sur place à coût 0) et $d(s_0, t, 0) = \infty$ pour $t \neq s$.

L'algorithme de Bellman-Ford c'est la version dérécursivée et optimisée en mémoire du problème de programmation dynamique : on construit les $d(s_0, t, \ell)$ en faisant croître ℓ de 0 à $S-1$, en les stockant dans un tableau $d_\ell[t]$ (s_0 est fixé, c'est un problème de type 2). On se rend compte ensuite qu'il suffit de se rappeler de $d_{\ell-1}[\,]$.

Algorithm 20 Bellman-Ford version 0

```
1:  procedure BellmanFord_v0(G, s_0)
2:      for t ∈ S do
3:          if t ≠ s_0 then
4:              d_ℓ[t] = ∞
5:          else
6:              d_ℓ[t] = 0
7:          end if
8:      end for
9:      for ℓ de 1 à S − 1 do
10:         Recopier d_ℓ[ ] dans d_{ℓ−1}[ ]
11:         for x → y ∈ A do
12:             if d_{ℓ−1}[x] + p(x → y) < d_ℓ[y] then
13:                 d_ℓ[y] = d_{ℓ−1}[x] + p(x → y)
14:             end if
15:         end for
16:     end for
17: end procedure
```

On remarque d'abord que l'algorithme est toujours valide si on utilise un seul tableau $d[\,]$ à la place de $d_\ell[\,]$ et $d_{\ell-1}[\,]$: on ne suit pas exactement la formule de récurrence, mais on ne peut que trouver des plus courts chemins plus vite.

En deuxième remarque, on peut à moindre coût stocker les plus courts chemins, et non seulement leur longueur. Il suffit, pour chaque sommet t de se rappeler de son prédecesseur $\pi[t]$ dans un plus court chemin. On reconstitue alors le plus court chemin de s_0 à t en sens inverse, en remontant les prédecesseurs successifs de t jusqu'à trouver s_0 :

$$s_0 \to \cdots \to \pi[\pi[t]] \to \pi[t] \to t.$$

Il suffit pour cela d'initialiser $\pi[t]$ à $\emptyset$ et de le mettre à jour après la ligne 13 quand on améliore un chemin.

Enfin, l'algorithme permet de détecter les circuits de poids négatif : il suffit de faire une itération de plus et de voir si on améliore les valeurs obtenues. Il y a un circuit de poids négatif si et seulement si il y a amélioration.

```
 1: procedure BellmanFord(G,s_0)
 2:     for t ∈ S do
 3:         π[t] = ∅
 4:         if t ≠ s_0 then
 5:             d[t] = ∞
 6:         else
 7:             d[t] = 0
 8:         end if
 9:     end for

10:     for ℓ de 1 à S − 1 do
11:         for x → y ∈ A do
12:             if d[x] + p(x → y) < d[y] then
13:                 d[y] = d[x] + p(x → y)
14:                 π[y] = x
15:             end if
16:         end for
17:     end for

18:     for x → y ∈ A do
19:         if d[x] + p(x → y) < d[y] then
20:             Signaler circuit de poids négatif
21:         end if
22:     end for
23: end procedure
```

La complexité de l'algorithme est $\mathcal{O}(AS)$ pour les listes et $\mathcal{O}(S^3)$ pour les matrices. L'algorithme est assez coûteux, mais fonctionne avec des poids négatifs.

Théorème VII.3 (Validité du test de circuit négatif) *L'algorithme de Bellman-Ford détecte bien les circuits de poids négatifs.*

Démonstration : D'abord, si l'algorithme indique un circuit de poids négatif, c'est qu'un chemin de s à t de longueur au plus S est strictement meilleur qu'un chemin de longueur au plus $S-1$. Cela indique la présence d'un circuit de poids négatif (cf la preuve du théorème sur la longueur).

Réciproquement, supposons qu'il y ait un circuit $x_1, \dots x_n$ de poids négatif. On pose $x_{n+1} = x_1$ pour simplifier l'écriture. On a

$$\sum_{i=1}^{n} p(x_i \to x_{i+1}) < 0,$$

donc

$$\sum_{i=1}^{n} (d[x_i] + p(x_i \to x_{i+1})) < \sum_{i=1}^{n} d[x_i] = \sum_{i=1}^{n} d[x_{i+1}]$$

Pour que cela soit possible, il faut qu'il y ait au moins une valeur de i telle que

$$d[x_i] + p(x_i \to x_{i+1}) < d[x_{i+1}].$$

□

VII.3 Algorithme de Dijkstra (fin des années 50)

On cherche toujours une solution à un problème de type 2, mais avec la contrainte qu'il n'y ait pas de poids négatif. Cette simplification conduit à un algorithme beaucoup plus efficace, puisqu'elle rend possible une stratégie gloutonne. On note s_0 la source à partir de laquelle on calcule les plus court chemins.

L'idée est la suivante : a tout moment lors de l'exécution de l'algorithme, on a séparé l'ensemble $\mathcal{S}$ des sommets en deux ensembles disjoints $\mathcal{E}$ et $\mathcal{F}$. L'ensemble $\mathcal{E}$ contient les sommets dont on connaît la distance à s_0 et $\mathcal{F}$ ceux qui n'ont pas encore été déterminés. A chaque étape on enlève un élément t de $\mathcal{F}$ bien choisi, et on l'ajoute à E. Le choix de t est le *choix glouton*, toute la difficulté revient à le choisir de façon à bien calculer les plus courts chemins, et de pouvoir le faire efficacement.

Le choix de t s'effectue de la façon suivante : on regarde tous les chemins composés d'un chemin $s_0 \rightsquigarrow x$ qui reste dans $\mathcal{E}$ (ceux dont on connaît la distance à s_0) suivi d'un arc $x \to y$, où $y \in F$. Parmi tous ces chemins, on sélectionne celui de poids minimal, autrement dit celui qui minimise la quantité $d(s_0, x) + p(x \to y)$. C'est le chemin de moindre poids qui permet de sortir de E, et on choisit $t := y$. Il est facile de voir qu'on a bien $d(s_0, t) = d(s_0, x) + p(x \to t)$, puisque pour atteindre t il faut nécessairement sortir de $\mathcal{E}$ et que cela "coûte" déjà au moins $d(s_0, x) + p(x \to t)$. Comme il n'y a pas de poids négatif, tout chemin qui atteint un élément de $\mathcal{F}$ a un poids supérieur ou égal à celui-là, il n'y a donc pas de meilleur chemin pour atteindre t.

En pratique on ne regarde évidemment pas tous les chemins à chaque étape. Il suffit de mettre à jour, pour chaque sommet $y \in \mathcal{F}$ la meilleure quantité $d(s_0, x) + p(x \to y)$ vue jusqu'ici. La mise à jour se fait quand on extrait un sommet t de $\mathcal{F}$ pour le mettre dans $\mathcal{E}$: on connaît alors sa distance et on considère tous les arcs d'origine t et à destination dans $\mathcal{F}$ pour chercher s'il y a une amélioration.

Pour stocker les plus courts chemins, et non pas seulement le poids des plus courts chemins, on utilise la même technique pour que l'algorithme de Bellman-Ford, il suffit pour tout sommet y de se rappeler de son prédecesseur $\pi[y]$ dans un plus court chemin. On utilise un tableau $d[\,]$ avec $d[s] = d(s_0, s)$ si $s \in \mathcal{E}$, et sinon $d[s]$ est la meilleur estimation de distance vue jusqu'ici, c'est-à-dire le meilleur des $d(s_0, x) + p(x \to s)$ pour $x \in \mathcal{E}$.

La complexité de l'algorithme dépend à la fois de la représentation choisie pour les graphes et de la structures de données utilisée pour gérer l'extraction de minimum. Dans le cas de la

```
1: procedure DIJKSTRA(G,s_0)
2:     F = S
3:     for t ∈ S do
4:         π[t] = ∅
5:         if t == s_0 then
6:             d[t] = 0
7:         else
8:             d[t] = ∞
9:         end if
10:    end for

11:    while F ≠ ∅ do
12:        t = Extraire de F l'élément qui minimise d[ ]
13:        for s successeur de t do
14:            if d[t] + p(t → s) < d[s] then
15:                d[s] = d[t] + p(t → s)
16:                π[s] = t
17:            end if
18:        end for
19:    end while
20: end procedure
```

représentation par matrice, le plus simple est d'utiliser un tableau pour $d[\,]$, ce qui conduit à une complexité de $O(S^2)$ pour les extractions de minimum et de $O(S^2)$ aussi pour la mise à jour de $d[\,]$, soit une complexité totale en $O(S^2)$. En représentation par listes d'adjacence, l'utilisation d'un tableau pour $d[\,]$ donne aussi une complexité en $O(S^2)$. On peut cependant utiliser à la place une file de priorité (codée par un tas), qui supporte les opérations d'ajout, extraction de minimum et mise à jour d'une valeur en temps $O(\log S)$. Ce qui donne une complexité globale de l'algorithme en $O(A \log S)$ (c'est l'étape de mise à jour qui domine). Le choix entre tableau et file de priorité dépend donc de si $A \log S \ll S^2$ ou non.

VII.4 Algorithme de Floyd-Warshall (fin des années 50)

Il semblerait que c'est en fait Bernard Roy qui l'ait décrit en premier, l'algorithme devrait porter son nom.

Il s'agit d'un algorithme de type 3 :on veut connaître les plus courts chemins de s à t pour tout s et pour tout t. L'algorithme de Floyd-Warshall est un algorithme de programmation dynamique, mais où on ne découpe pas les chemins selon leur longueur comme dans l'algorithme de Bellman-Ford. Le découpage se fait selon les sommets intermédiaires.

Soit $C = x_0 \to x_1 \to \cdots \to x_n$ un chemin de longueur n, l'ensemble des *sommets intermédiaires* de C est l'ensemble $\{x_i \mid i \in \{1, \ldots, n-1\}\}$. En particulier, il n'y a x_0 (ou x_n) parmi les sommets intermédiaires que s'ils apparaissent une seconde fois dans C.

On suppose maintenant que les sommets du graphe sont numérotés de 0 à $S-1$, ce que l'on peut toujours faire si besoin. On note $d_k(s,t)$ le poids d'un plus court chemin allant de s à t dont **tous les sommets intermédiaires sont strictement plus petits que** k.

En particulier, pour calculer d_0 on n'a le droit à aucun sommet intermédiaire sur les chemins

considérés, on ne peut donc choisir que des arcs (chemins de longueur 1). Pour d_S, on a le droit à tous les sommets intermédiaires, c'est-à-dire qu'on considère tous les chemins, et donc $d_S = d$.

Il s'agit par conséquent de trouver comment passer de d_k à d_{k+1}. L'initialisation (d_0) se fait en regardant tous les arcs, et on a terminé quand $k = S$. On suppose qu'il n'y a pas de circuit de poids négatif. On observe que lorsqu'on passe du calcul de d_k à celui de d_{k+1}, on rajoute le sommet k à la liste des sommets intermédiaires possibles. Un plus court chemin de s à t de sommets intermédiaires $\leq k$ passe par k ou non. S'il passe par k, en enlevant les circuits inutiles, c'est un chemin de la forme $s \rightsquigarrow k \rightsquigarrow t$, où les deux chemins ont des sommets intermédiaires $\leq k-1$. S'il ne passe pas par k, ses sommets intermédiaires sont $\leq k-1$. On en déduit donc la formule :

$$d_{k+1}(s,t) = \min\big\{d_k(s,t), d_k(s,k) + d_k(k,t)\big\}.$$

On a la formule récursive qui permet de faire de la programmation dynamique. On dérécursive et optimise pour trouver l'algorithme de Floyd-Warshall. On utilise $d[s,t]$ pour stocker le poids du plus court chemin de s à t et $\pi[s,t]$ pour stocker le prédecesseur de t dans un plus court chemin d'origine s.

```
1:  procedure FLOYD-WARSHALL(𝒢)
2:      for s ∈ 𝒮 et t ∈ 𝒮 do
3:          if s, t then
4:              d[s,t] = 0
5:              π[s,t] = ∅
6:          else
7:              if s → t then
8:                  d[s,t] = p(s → t)
9:                  π[s,t] = s
10:             else
11:                 d[s,t] = ∞
12:                 π[s,t] = ∅
13:             end if
14:         end if
15:     end for

16:     for k de 0 à S − 1 do
17:         for s de 0 à S − 1 do
18:             for t de 0 à S − 1 do
19:                 if d[s,k] + d[k,t] < d[s,t] then
20:                     d[s,t] = d[s,k] + d[k,t]
21:                     π[s,t] = π[k,t]
22:                 end if
23:             end for
24:         end for
25:     end for
26: end procedure
```

Pour une fois, la meilleure représentation est par matrice. La complexité est en $O(S^3)$.

CHAPITRE VIII

Arbres couvrants minimaux

VIII.1 Préambule : représentation des partitions

VIII.1.1 Définition

Si E est un ensemble, une *partition* P de E est un ensemble de sous-ensembles de E, $P = \{E_1, \ldots, E_k\}$, tel que :

1. les E_i sont deux à deux disjoints ;
2. les E_i ne sont pas vides ;
3. l'union de tous les E_i fait E tout entier.

Autrement dit, tout élément de E est dans un unique E_i (et aucun E_i n'est vide). Chaque E_i s'appelle une *part* ou un *bloc* de P.

Par exemple, si $E = \{a, b, c, d, e\}$, $P = \{\{a, c\}, \{b\}, \{d, e\}\}$ est une partition de E. Si on l'ensemble de tous les singletons, on a une autre partition $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}\}$.

Dans la suite on supposera que $E = \{0, \ldots, n-1\}$ et on s'intéresse à trouver une structure de données adaptée pour représenter les partitions, sachant que l'on souhaite pouvoir effectuer les opérations suivantes :

init : initialiser P comme étant la partition avec les n singletons : $P = \{\{0\}, \{1\}, \ldots, \{n-1\}\}$.

find : étant donné x et y dans E, savoir si x et y sont dans la même part ;

union : fusionner deux parts de P : cela revient à remplacer les parts E_i et E_j par la seule $E_i \cup E_j$.

On cherche à optimiser la complexité sachant que l'on va effectuer **une initialisation et m opérations de find ou union sur un ensemble à n éléments**.

VIII.1.2 Représentation par des forêts d'arbres

Si on utilise des structures de données élémentaires, comme des tableaux ou des listes chaînées, on arrive facilement à coder les partitions. Cependant les complexités ne sont pas très bonnes, il est difficile d'avoir à la fois une bonne complexité pour union et pour find. Par exemple, si on numérote les parts et qu'on met le numéro de la part i dans un tableau en

position $T[i]$, le find se fait en $\mathcal{O}(1)$, mais l'union prend un temps $\mathcal{O}(n)$ dans le pire des cas. La complexité du problème est donc de $\mathcal{O}(mn)$.

L'idée est d'utiliser des arbres pour encoder chaque part. L'opération find compare les racines des arbres de x et de y : ils sont dans la même part si et seulement si ils ont la même racine. Pour l'union, il suffit de greffer la racine de E_i sous celle de E_j (ou le contraire) pour obtenir un arbre de $E_i \cup E_j$. Cela se fait en complexité raisonable si on s'arrange pour ne pas trop augmenter la hauteur, en greffant le moins haut à la racine du plus haut.

On peut encore optimiser l'algorithme en utilisant les opérations find pour optimiser la hauteur des arbres : si on a calculé la racine r de x, autant en profiter pour mettre x directement comme fils de r, comme ça on réduit potentiellement la hauteur de l'arbre, à moindre coût. Comme pour trouver la racine de x, on remonte tous les ancêtres de x jusqu'à trouver r, on va en profiter pour mettre sous r tous les éléments de la branche de r à x.

On utilise un tableau $rang[]$, où $rang[i]$ est une estimation de la hauteur du sous-arbre de racine i. Ce n'est pas la hauteur exacte, car maintenir la hauteur précise coûte trop cher en opérations. Les algorithmes sont les suivants : On représente les arbres en utilisant un tableau de prédécesseurs π, où $\pi[i]$ est le père de i, sauf si i est une des racines, auquel cas $\pi[i] = i$ par convention.

Algorithm 21 Init

```
1: procedure INIT(n)
2:     for i de 0 à n − 1 do
3:         π[i] = i
4:         rang[i] = 0
5:     end for
6:     return (π, rang)
7: end procedure
```

La fonction find renvoit la racine de l'arbre de x. Il suffit alors de comparer les racines de x et y pour savoir s'ils sont dans la même part.

Algorithm 22 Find

```
1: procedure FIND(x,π)
2:     if π[x] = x then
3:         return x
4:     else
5:         π[x] = Find(π[x],π)
6:         return π[x]
7:     end if
8: end procedure
```

Union fusionne les parts de x et de y, en appelant find pour trouver les racines des arbres associés. Le rang sert d'estimateur pour savoir quel arbre mettre sous l'autre.

La complexité du problème initial est de $\mathcal{O}(m\alpha(n))$, où $\alpha(n)$ est une fonction qui tend vers l'infini mais extrèmement lentement : pour tous cas pratiques, on a $\alpha(n) \leq 5$. Cette méthode est donc "presque" linéaire en m, le nombre d'opérations d'union ou de find.

Algorithm 23 Union

```
1: procedure UNION(x,y,π)
2:     r_x = Find(x,π)
3:     r_y = Find(y,π)
4:     if rang[r_x] < rang[r_y] then
5:         π[r_x] = r_y
6:     else
7:         if rang[r_x] == rang[r_y] then
8:             rang[r_x] = rang[r_x] + 1
9:         end if
10:        π[r_y] = r_x
11:    end if
12: end procedure
```

VIII.2 Arbre couvrant minimal : description

On s'intéresse à des graphes **non-orientés** et valués. On suppose que le graphe est connexe : de tout sommet on peut aller à tout autre.

Un *arbre* dans notre contexte est un graphe non-orienté sans circuit. Un *arbre couvrant* d'un graphe $\mathcal{G}$ est un sous-graphe de $\mathcal{G}$, c'est-à-dire qu'on n'utilise qu'une partie des arêtes de $\mathcal{G}$, qui est un arbre et qui relie tous les sommets. Le *poids* d'un arbre couvrant est la somme des poids de ses arêtes.

On remarque que par récurrence immédiate, un arbre couvrant sur un graphe à S sommets possède exactement $S-1$ arêtes.

Un *arbre couvrant minimal* est un arbre couvrant de poids minimal. Le problème qui nous intéresse dans ce chapitre c'est de calculer efficacement un tel arbre.

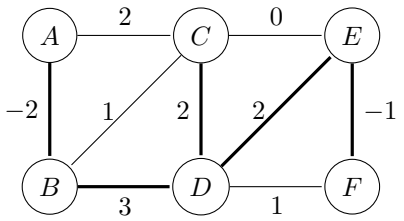


FIG. VIII.1 – Un exemple de graphe non-orienté valué. En gras est indiqué un arbre couvrant de poids 4, qui n'est pas minimal puisqu'il existe un arbre couvrant de poids -1.

VIII.3 Algorithme de Kruskal (1956)

La solution que nous allons voir est due à Kruskal. Il s'agit d'un algorithme glouton, qui ajoute une à une des arêtes dans l'arbre jusqu'à ce qu'il possède bien $S-1$ arêtes.

A tout moment lors de l'algorithme, on a une forêt, c'est-à-dire un ensemble d'arbres. Quand il n'y a plus qu'un seul arbre, l'algorithme est terminée. L'idée de la méthode est simple : on

prend les arêtes par poids croissant, et on les ajoute dans la solution quand elles ne créent pas de circuit. Tester à chaque étape la présence de circuit serait trop coûteux pour être efficace, il faut une autre idée.

La gestion des forêts se fait avec la structure de partitions de la première partie. On va tenir à jour une partition P sur l'ensemble des sommets telle qu'à tout moment, deux sommets sont dans la même part si et seulement si ils sont dans le même arbre de la forêt. On voit que c'est la seule information dont on a besoin, car au moment où on considère l'arête $x - y$:

- si x et y sont dans la même part (find), il ne faut pas ajouter l'arête, cela créerait un cycle ;
- sinon on ajoute l'arête et on met la structure à jour : maintenant tous les sommets de l'arbre de x et de celui de y sont dans le même arbre, il faut les fusionner avec la fonction union.

De plus l'initialisation est exactement ce qu'il nous faut, puisqu'au début on n'a aucun arbre, donc chaque sommet n'est relié qu'à lui-même.

Algorithm 24 Kruskal

```
1: procedure KRUSKAL(𝒢)
2:     L = Liste des arêtes triée par poids croissant
3:     𝒜 = ∅
4:     π = Init(|𝒢|)
5:     for x − y dans L dans l'ordre do
6:         if Find(x,π) ≠ Find(y,π) then
7:             Union(x,y,π)
8:             Ajouter x − y dans 𝒜
9:         end if
10:    end for
11:    return 𝒜
12: end procedure
```

Au total on fait au pire une union et deux find par itération, et il y a une itération par arête. La complexité de la boucle for est donc $\mathcal{O}(A\,\alpha(S))$ puisqu'on fait un nombre proportionnel à A d'opération sur la partition d'un ensemble de taille S. L'étape de tri est, de façon surprenante, l'étape la plus coûteuse, puisqu'elle nécessite $\mathcal{O}(A \log A) = \mathcal{O}(A \log S)$ opérations. C'est la complexité finale de l'algorithme, Kruskal s'effectue en temps $\mathcal{O}(A \log S)$ grâce à l'efficacité de la structure de donnée choisie pour les partitions.