Quelques développements pour l'agrég de maths (leçons d'informatique)

Nguyễn Lê Thành Dũng (ENS Ulm, département informatique, promo 2012)

Préparation à l'agrégation de l'ENS Paris-Saclay, 2016–2017

Voici quelques développements relativement originaux pour les leçons d'informatique de l'agrégation. Ils devraient garder leur pertinence après le changement de programme de la session 2018 qui voit le lambda-calcul et la sémantique opérationnelle remplacer la réécriture et l'unification.

Je conseille fortement d'aller puiser de l'inspiration pour les développements d'algorithmique dans les notes de cours de Jeff Erickson, disponibles sur le Oueb à l'adresse suivante : `algorithms.wtf`.

Table des matières

1 Algorithme de Hirschberg

Une superbe astuce qui consiste à utiliser un diviser-pour-régner afin de baisser la complexité spatiale d'un algorithme de programmation dynamique. On l'applique ici au problème du *plus long sous-mot commun* (PLSC), mais la technique est assez générique.

Pour une fois, le cours de Jeff Erickson n'est pas la meilleure référence : il se complique inutilement la vie avec sa relation de récurrence.

On considère le problème d'optimisation suivant :

— Entrée : $u, v \in \Sigma^*$ (Σ alphabet fini) ;

— Sortie : w sous-mot commun de u et v, tel que sa longueur $|w|$ soit maximale.

Notons $\mathrm{PLSC}(u, v)$ l'ensemble des solutions optimales, et $l(u, v)$ la longueur d'une telle solution.

Proposition. *Soient $u, v \in \Sigma^*$, $m = |u|$, $n = |v|$. le tableau $t[i] = l(u[1...i], v)$, $i = 1, ..., m$ peut être calculé en temps $O(mn)$ et espace $O(m)$.*

Démonstration. C'est l'algorithme archi-classique de programmation dynamique… Admis. □

En particulier, $l(u, v)$ peut être calculé avec cette complexité (remarquons de plus que les rôles de u et v étant symétriques, on peut avoir un espace $O(\min(m, n))$). Il s'agit maintenant de trouver une *solution* optimale $w \in \mathrm{PLSC}(u, v)$, avec les mêmes complexités temporelle et spatiale que le calcul de la *valeur* optimale. (En temps $O(mn)$ et espace $O(mn)$, ça se fait en lisant la solution sur le tableau d'états de la programmation dynamique.)

Lemme. *Posons $g = u[1...\lfloor m/2 \rfloor]$ et $d = u[\lfloor m/2 \rfloor + 1...m]$. Soit i maximisant $l(g, v[1...i]) + l(d, v[i+1...n])$. Alors, pour tout $w \in \mathrm{PLSC}(g, v[1...i])$ et $w' \in \mathrm{PLSC}(d, v[i + 1...n])$, on a $w \cdot w' \in \mathrm{PLSC}(u, v)$.*

Démonstration. Soit $s \in \mathrm{PLSC}(u, v)$. s est un sous-mot de u et $g \cdot d = u$, donc il existe s', s'' tels que s' (resp. s'') soit un sous-mot de g (resp. d). Maintenant, $s' \cdot s''$ est sous-mot de v, donc il existe j tel que s' (resp. s'') soit un sous-mot de $v[1...j]$ (resp. $v[j + 1...n]$).

Pour w, w' donnés dans l'énoncé, par deux arguments successifs de maximalité, on a

$$|w \cdot w'| = l(g, v[1...i]) + l(d, v[i + 1...n]) \geq l(g, v[1...j]) + l(d, v[j + 1...n]) \geq |s \cdot s'| = l(u, v).$$

Or $w \cdot w'$ est un sous-mot commun de u et v, donc c'est un plus long sous-mot commun. □

Ce lemme conduit à considérer l'algorithme suivant, dont il prouve la correction :

---

procédure HIRSCHBERG(u, v)
  $(m, n) \leftarrow (|u|, |v|)$
  si $m \leq 1$ **alors**
    calculer $w \in \mathrm{PLSC}(u, v)$ en temps $O(n)$
    renvoyer w
  sinon
    $(g, d) \leftarrow (u[1...\lfloor m/2 \rfloor], u[\lfloor m/2 \rfloor + 1...m])$
    $lg \leftarrow [l(g, v[1...i]) \mid i = 1, \dots, n]$ ▷ En un seul appel à la sous-routine de prog. dyn.
    $ld \leftarrow [l(d, v[i + 1...n]) \mid i = 1, \dots, n]$ ▷ Idem en renversant le tableau
    Soit i minimisant $lg[i] + ld[i]$
    renvoyer HIRSCHBERG($g, v[1...i]$) · HIRSCHBERG($d, v[i + 1...n]$)

L'analyse n'est pas très dure, mais il faut expliciter des constantes…

2 Tri des suffixes

Ce développement d'algorithmique du texte se recase aussi dans la leçon *Exemples d'algorithmes de tri*, en fournissant une application inattendue du tri par base. Référence : Crochemore-Hancart-Lecroq, *Algorithmique du texte*, section 4.4. La présentation est un peu trop « bas niveau », notamment en ce qui concerne les détails du tri par base ; par conséquent, ça fait peur au premier abord, alors qu'en fait le principe est très simple.

On cherche à trier par ordre lexicographique l'ensemble des suffixes d'un mot. L'application principale est la construction d'une *table des suffixes* [1], structure de données utile dans de nombreux problèmes d'algorithmique du texte (notamment en concours de programmation).

Définissons formellement le problème :

— Entrée : $w \in A^*$ (A alphabet fini), $|w| = n$.

— Sortie : $\sigma \in \mathfrak{S}_n$ telle que $w[\sigma(1) \dots n] < \dots < w[\sigma(n) \dots n]$.

Naïvement, on pourrait appliquer un tri par comparaison sur l'ensemble des n suffixes, mais chaque comparaison entre deux chaînes prend un temps $O(n)$ ce qui donne une complexité totale $O(n^2 \log n)$! On va voir comment faire mieux.

Principe de l'algorithme Soit $R_k[i]$ le rang de $w[i \dots i+2^k-1]$ dans $\{w[j \dots j+2^k-1] \mid j = 1, \dots, n\}$ ordonné *sans répétition*, de sorte que $R_k[i] < R_k[j] \Leftrightarrow w[i \dots i+2^k-1] < w[j \dots j+2^k-1]$. (Si $i+2^k > n$, on tronque à n.) On a donc $1 \leq R_k[i] \leq n$ pour $i \in \{1, \dots, n\}$, et on pose $R_k[i] = 0$ pour $i > n$.

Notons que :

— si $k = 0$, alors cela revient à trier les suffixes par leur première lettre, c'est-à-dire à trier les lettres de w ;

— si $k \geq \log_2 n$, alors on a le vrai ordre sur les suffixes : formellement, $\sigma^{-1}(i) = R_k[i]$.

L'algorithme va calculer par récurrence R_k pour $k = 0, \dots, \lceil \log_2 n \rceil$.

Lemme. *$R[k+1][i]$ est le rang de $(R_k[i], R_k[i+2^k])$ dans $\{(R_k[j], R_k[j+2^k]) \mid j = 1, \dots, n\}$ ordonné sans répétition par ordre lexicographique.*

Heuristique. Plaçons-nous dans le cas où $i + 2^{k+1} - 1$ et $j + 2^{k+1} - 1$ sont inférieurs à n. Alors

$$w[i \dots i + 2^{k+1} - 1] < w[j \dots i + 2^{k+1} - 1]$$

$$\Leftrightarrow w[i \dots i + 2^k - 1] \cdot w[i + 2^k \dots i + 2^{k+1} - 1] < w[j \dots j + 2^k - 1] \cdot w[j + 2^k \dots j + 2^{k+1} - 1]$$

$$\Leftrightarrow (w[i \dots i + 2^k - 1], w[i + 2^k \dots i + 2^{k+1} - 1]) < (w[j \dots j + 2^k - 1], w[j + 2^k \dots j + 2^{k+1} - 1])$$

et il ne reste plus qu'à traduire cela en termes de rangs. □

Début d'une preuve plus rigoureuse. Le souci avec l'argument précédent, c'est les dépassements d'indice. Pour les éviter en se facilitant la vie, on pose $u = w\$^\omega \in (A \cup \{\$\})^\omega$, où $\$$ est plus petit que les lettres de A, cette condition s'écrit aussi $u[\sigma(1) \dots] < \dots < u[\sigma(n) \dots]$. Puis on étudie l'ordre des suffixes (infinis !) de u sans répétition, en attribuant à $\$^\omega$ le rang 0 : cela correspond aux rangs donnés par R. □

1. En anglais, *suffix array*.

D'où l'algorithme suivant :

---

procédure TriSuffixes(w)
  $n \leftarrow |w|$
  $R_0 \leftarrow$ TriRang($[w[i] \mid i = 1, \dots, n]$)
  pour $k \leftarrow 1, \dots, \lceil \log_2 n \rceil$ **faire**
    $R_k \leftarrow$ TriRang($[(R_{k-1}[i], R_{k-1}[i+k]) \mid i = 1, \dots, n]$)
  renvoyer l'inverse de la permutation R_k

Implémentation du tri On sait que $0 \leq R_{k-1}[i] \leq n$ pour tout $i \in \mathbb{N}^*$. Trier les couples $(R_{k-1}[i], R_{k-1}[i+k])$ revient donc à trier une liste de n nombres à 2 chiffres en base $n+1$.

Rappelons que l'algorithme de *tri par base*[2] permet de trier n nombres à d chiffres en base b en temps $O(d(n+b))$. Ici, sa complexité est donc de $O(2 \times (n + (n+1)))$, soit $O(n)$.

Complexité totale On a $\lceil \log_2 n \rceil$ itérations dont chacune coûte $O(n)$: au total, l'algorithme tourne en temps $O(n \log n)$. On a donc battu l'algorithme naïf d'un facteur n !

En espace, l'implémentation proposée consomme $O(n)$. Cependant, garder en mémoire les R_k pour $k = 1, \dots, \lceil \log_2 n \rceil$, ce qui prend $O(n \log n)$, est utile par la suite dans la construction de la table des suffixes, pour calculer les plus longs préfixes communs entre deux suffixes.

2. En anglais, *radix sort*.

3 Union-find avec compression de chemin

On considère l'implémentation usuelle d'une structure *Union-Find* (qui maintient des classes d'équivalence d'un ensemble fini, de cardinal n) avec une forêt enracinée. Il existe deux optimisations classiques :

— l'*union par rang* permet d'avoir des opérations en temps $O(\log n)$ dans le pire cas ;
— la *compression de chemins* permet d'atteindre une complexité $O(\alpha(n))$ amortie, où α est la réciproque de la fonction d'Ackermann (autant dire qu'il s'agit quasiment de $O(1)$).

En pratique, c'est la compression de chemins qui apporte une accélération significative, l'union par rang ne faisant principalement qu'ajouter un overhead d'un facteur constant quand les deux optimisations sont implémentées. Souvent, dans la vraie vie, on va donc implémenter *uniquement* la compression de chemins. Obtient-on alors une complexité raisonnable en théorie ?

Théorème. *L'implémentation de la structure* Union-Find *sans union par rang et avec compression de chemins a des opérations en temps $O(\log n)$ amorti.*

La démonstration donnée ici est inspirée d'une réponse trouvée sur `cs.stackexchange` ; une autre preuve est proposée dans Melhorn, *Algorithms and Data Structures – The Basic Toolbox*.

Description de l'implémentation On représente une partition de $\{1, \dots, n\}$ par une forêt : chaque partie E correspond à un arbre étiqueté par les éléments de E, dont la racine $r \in E$ est le représentant canonique. Cette forêt est elle-même codée par un tableau : $T[i] = j$ signifie que

— si $i = j$, le nœud d'étiquette i est une racine ;
— sinon, i a pour parent j.

L'opération Union, qui fusionne deux parties, chacune correspondant à un arbre, affecte la racine d'un arbre comme fils de la racine de l'autre arbre. L'opération Find prend en entrée un élément $x \in \{1, \dots, n\}$ et renvoie la racine r correspondante ; ainsi, on peut tester si x et y sont dans la même partie par $\textsc{Find}(x) = \textsc{Find}(y)$.

La compression de chemins consiste à faire en sorte que Find « aplatisse » le chemin entre x et r en rattachant tous les nœuds sur ce chemin directement à la racine.

Implémentation de la structure Union-Find

```
procédure Init(n)
    T ← nouveau tableau de taille n
    pour i ← 1, ..., n faire
        T[i] ← i
    renvoyer T
procédure Find(T, x)
    si T[x] ≠ x alors T[x] ← Find(T, t[x])
    renvoyer T[x]
procédure Union(T, x, y)
    T[Find(x)] ← Find(y)
```

Analyse amortie On va compter le nombre d'appels récursifs à FIND comme mesure de complexité (ce qui est raisonnable, puisque FIND effectue un traitement en $O(1)$ avant de s'appeler récursivement). On utilise la *méthode du potentiel* avec la fonction potentiel suivante :

$$\Phi = \sum_{i=1}^{n} \log_2 w(i)$$

où $w(i)$ est la taille du sous-arbre enraciné en i.

Complexité de FIND Regardons ce qui se passe quand on appelle FIND(x). Soit $x = x_1 \to ... \to x_k = r$ le chemin à la racine. On a alors k appels à FIND : c'est le coût réel.

La différence de potentiel, elle, s'écrit (en notant w (resp. w') pour les coûts avant (resp. après) les mutations entraînées par FIND)

$$\Phi' - \Phi = \sum_{i=1}^{n} \log_2 \frac{w'(i)}{w(i)} = \sum_{j=1}^{k} \log_2 \frac{w'(x_j)}{w(x_j)} = \sum_{j=2}^{k-1} \log_2 \left(1 - \frac{w(x_{j-1})}{w(x_j)}\right)$$

En effet, d'une part, les nœuds en-dehors du chemin ne sont pas affectés ; d'autre part, on a $w'(x_j) = w(x_j) - w(x_{j-1})$ pour les nœuds intermédiaires du chemin puisque l'on enlève à x_j son fils x_{j-1} pour le rattacher à la racine.

Maintenant, on veut montrer $k + \Phi' - \Phi = O(\log n)$; il suffit pour cela qu'il n'y ait qu'au plus $O(\log n)$ nœuds intermédiaires tels que $1 - w(x_{j-1})/w(x_j) \geq 1/2$. On peut s'en convaincre en remarquant qu'en parcourant le chemin de x à la racine, la taille du sous-arbre enraciné au nœud actuel ne peut doubler qu'au plus $\log_2 n$ fois.

Complexité de UNION C'est aussi du $O(\log n)$ amorti (il est important d'utiliser la même fonction potentiel que pour FIND) : en exercice pour la lectrice !

4 Minimisation par double renversement

Source : Sakarovitch, *Éléments de théorie des automates*.

Notons D la déterminisation d'un automate, prenant en entrée un AFN et renvoyant un AFDC, partie accessible de l'automate des parties. Notons T la transposition d'un automate, obtenue en renversant le sens des flèches. La transposée d'un automate reconnaissant L reconnaît $\tilde{L}$, langage miroir de L.

Algorithme (Calcul d'un AFDC minimal à partir d'un AFN)**.** *Si $\mathscr{A}$ est un automate reconnaissant le langage L, $D \circ T \circ D \circ T(\mathscr{A})$ est un AFDC minimal reconnaissant L.*

Comme la dernière opération est une déterminisation, la sortie est bien un AFDC. Le langage reconnu est préservé puisque D le préserve et T prend le miroir, ce qui est involutif. Reste à prouver la minimalité.

Lemme (Brzozowski)**.** *Soit $\mathscr{B}$ un automate reconnaissant L,* co-déterministe *et* co-accessible *(autrement dit, dont la transposée est un AFD accessible). Soient u et v vérifiant $u^{-1}L = v^{-1}L$. Alors tout état atteignable en lisant u l'est en lisant v.*

Démonstration. Soit q atteignable à partir de u, q est co-accessible donc il existe w atteignant l'unique état final à partir de q. $uw \in L$ donc $vw \in L$, et le seul chemin acceptant pour vw doit passer par q après avoir lu v par co-déterminisime… Faire un dessin ! □

Appliquons maintenant le lemme à $\mathscr{B} = T \circ D \circ T(\mathscr{A})$. Si deux mots ont même résiduels, ils atteignent le même état dans l'automate des parties de $\mathscr{B}$ en vertu du lemme. Il y a autant d'états que de résiduels dans $D(\mathscr{B})$, ce dernier est donc minimal.

Proposition. *La complexité de cet algorithme est simplement exponentielle en le nombre d'états de l'automate d'entrée.*

Pour une raison simple : la première fois qu'on déterminise, on peut avoir une explosion exponentielle ; la seconde fois, c'est impossible puisque la taille de la sortie est connue pour être celle d'un AFDC minimal !

TODO : réfléchir à la complexité précise de la déterminisation (avec taille de l'alphabet en paramètre).

Pour avoir une minoration correspondante, on considère $L = X^{n-1}aX^*$ où $X = \{a, b\}$ est l'alphabet. L est reconnaissable par un AFD à $n + 1$ états, et son miroir a 2^n résiduels ! La minimisation ne fait que *rajouter* un état puits pour rendre l'automate complet.

5 Théorème de Herbrand sémantique

On suppose que le langage $\mathscr{L}$ contient des constantes, pour assurer la non-vacuité des modèles de Herbrand.

Théorème (Herbrand sémantique)**.** *Soit A une formule existentielle : $A = \exists x_1 \dots \exists x_n B$ où B est sans quantificateur. Alors A est valide si et seulement si il existe un entier k et k instances $B\sigma_1, \dots, B\sigma_k$ telles que $B\sigma_1 \vee \dots \vee B\sigma_k$ soit valide.*

On peut étendre le résultat en remplaçant « formule valide » par « conséquence d'une théorie axiomatisée par des formules purement universelles ».

La preuve procède par réduction de la logique du premier ordre à la logique propositionnelle. Soit $\mathscr{A}$ l'ensemble des atomes clos du langage $\mathscr{L}$ (les $R(t_1, \dots, t_n)$ où R est un symbole de relation n-aire ($n \in \mathbb{N}$) autre que $=$ et $t_1, \dots, t_n$ sont des termes clos).

Remarque (fondamentale !)**.** *Les formules closes sans quantificateur sur $\mathscr{L}$ correspondent aux formules propositionnelles sur l'ensemble de variables propositionnelles $\mathscr{A}$.*

On peut donc parler de satisfaisabilité *propositionnelle* pour les formules sans quantificateur : cela correspond à être satisfaisable pour toute valuation booléenne $v : \mathscr{A} \to \{\mathrm{V}, \mathrm{F}\}$.

Lemme. *Si A est valide au premier ordre, alors $\{\neg B\sigma \mid \sigma \text{ substitution}\}$ est insatisfaisable propositionnellement.*

La preuve est renvoyée à plus tard. Par compacité propositionnelle, il existe $B\sigma_1, \dots, B\sigma_k$ tels que $\neg B\sigma_1 \wedge \dots \wedge \neg B\sigma_k$ soit insatisfaisable propositionnellement.

Maintenant, si $\mathfrak{M}$ est une $\mathscr{L}$-structure, alors elle induit une valuation booléenne sur les atomes $X \in \mathscr{A}$: $v_{\mathfrak{M}}(X) = \mathrm{V} \Leftrightarrow \mathfrak{M} \vDash X$. On a besoin d'un petit lemme technique.

Lemme. *Une formule close φ est satisfaite par $v_{\mathfrak{M}}$ si et seulement si $\mathfrak{M} \vDash \varphi$.*

Démonstration. Par induction structurelle triviale. □

Comme $v_{\mathfrak{M}}$ ne satisfait pas $\neg B\sigma_1 \wedge \dots \wedge \neg B\sigma_k$, on a donc que $\mathfrak{M} \nvDash \neg B\sigma_1 \wedge \dots \wedge \neg B\sigma_k$, autrement dit $\mathfrak{M} \vDash B\sigma_1 \vee \dots \vee B\sigma_k$, et ce pour toute $\mathscr{L}$-structure.

$B\sigma_1 \vee \dots \vee B\sigma_k$ est donc valide, CQFD ; reste à montrer le lemme.

Preuve du premier lemme. On a vu que la validité propositionnelle était plus forte que la validité au premier ordre ; il s'agit d'établir une sorte de réciproque.

Supposons par l'absurde que v soit une valuation satisfaisant tous les $\neg B\sigma$. On construit un *modèle de Herbrand* $\mathfrak{M}$:

— dont les élements $\bar{t}$ sont en bijection avec les termes clos t du langage ;
— où les symboles de fonctions sont interprétés par $f(\overline{t_1}, \dots, \overline{t_n}) = \overline{f(t_1, \dots, t_n)}$;
— où $\mathfrak{M} \vDash R(\bar{t_1}, \dots, \bar{t_n})$ ssi $v(R(t_1, \dots, t_n))$ vaut vrai.

On a donc par construction $v_{\mathfrak{M}} = v$, donc (par le second lemme) $\mathfrak{M} \vDash \neg B\sigma$ pour toute substitution σ par des termes clos.

Comme A est valide, $\mathfrak{M} \vDash \exists x_1 \dots \exists x_n B$, c'est-à-dire qu'il existe $\overline{t_1}, \dots, \overline{t_n}$ dans le modèle tels que $\mathfrak{M} \vDash B[\overline{t_1}/x_1, \dots, \overline{t_n}/x_n]$. Par un lemme d'adéquation qu'on pourrait montrer par induction structurelle mais qui est évident, en interprétant les éléments du modèles comme des termes, on a en fait $\mathfrak{M} \vDash B\sigma$ où $\sigma(x_i) = t_i$. Contradiction. □

6 Idée exclue : schéma de réflexion de Kreisel

Théorème (Kreisel)**.** *Soit T un sous-système fini de l'arithmétique de Peano. Pour toute formule close F, AP prouve : « F prouvable dans $T \Rightarrow F$ ».*

Note : dans la formule ϕ dont il est affirmé que $AP \vdash \phi$, à gauche de $\Rightarrow$, on a un prédicat de prouvabilité appliqué au code de Gödel de F, et à droite, on a vraiment F.

Corollaire. *L'arithmétique de Peano prouve la cohérence de ses sous-systèmes finis, et n'est donc pas finiment axiomatisable.*

D'abord, on peut se ramener au cas où $T = \varnothing$. Ensuite, essentiellement, on veut montrer naïvement que F prouvable $\Rightarrow F$ vraie par induction sur les règles logiques, qui préservent la vérité. Comme les formules apparaissant dans une preuve sans coupures de F sont de complexité logique bornée par celle de F, on peut définir un prédicat de vérité borné qui fait marcher ça, et de sorte que « F vraie » soit équivalent à F.

Souci majeur : les détails sont impossibles à expliciter en 15 minutes, et la preuve repose sur la possibilité de raisonner de façon interne dans AP, ce pourquoi il faut déjà avoir une bonne intuition de quels principes logiques y sont autorisés (les fonctions récursives sont définissables, l'induction structurelle est valide…) car cela devient imbuvable sans un recours intensif à l'agitage de mains.

Je ne connais pas d'autre référence « livre de cours » que Le Point aveugle (Girard) pour ce théorème…