LA FONCTION D'ACKERMANN N'EST PAS PRIMITIVE RECURSIVE

Leçon 912

Référence Cori Lascar tome 2 p. 18-22.
Dehornoy (maths de l'info) p. 190.

Théorème

La fonction d'Ackermann, définie par $A(n,m) := A_n(m)$, où

$$A_0(m) = m+1$$
$$A_{n+1}(0) = A_n(1)$$
$$A_{n+1}(m+1) = A_n(A_{n+1}(m))$$

n'est pas primitive récursive.

Résumé

I - Résultats préliminaires :

- Croissances de $m \mapsto A_n(m)$
  $n \mapsto A_n(m)$
- $m+1 \leq A_n(m)$.
- $km \leq A_2^k(m)$
- $A_k(A_n(m)) \leq A_{\max(k,n)+2}(m)$.

II - Preuve que les fonctions primitives récursives sont majorées par A_n pour un certain $n \in \mathbb{N}$.

III - Preuve par l'absurde que la fonction d'Ackermann n'est pas primitive récursive.

I • Croissances :

* $A_n(m+1) \geqslant A_n(m)$

→ $n=0$: $A_0(m+1) = m+2 \geqslant m+1 = A_0(m)$.

→ n fixé $A_{n+1}(m+1) = A_n(A_{n+1}(m)) \geqslant A_{n+1}(m)$.

* $A_{n+1}(m) \geqslant A_n(m)$

→ $m=0$: $A_{n+1}(0) = A_n(1) \geqslant A_n(0)$

→ m fixé : $A_{n+1}(m+1) = A_n(A_{n+1}(m)) \geqslant A_n(m+1)$

• On voit qu'on a aussi et aboutit de $A_n(m) \geqslant m+1$.

→ $n=0$: $A_0(m) = m+1$

→ n fixé : $A_{n+1}(0) = A_n(1) \geqslant 1$

$A_{n+1}(m+1) = A_n(A_{n+1}(m)) \geqslant A_{n+1}(m)+1 \geqslant m+2$

• On a aussi envie de se débarasser de trucs du type $A_n(km)$, donc on va montrer que $km \leq A_2^k(m)$

→ $k=1$: $A_2(m) \geqslant 2m+3 \geqslant m$

→ k fixé : $A_2^{k+1}(m) = A_2^k(A_2(m))$

$= A_2^k(2m+3)$

$\geqslant 2km+3k$

$\geqslant (k+1)m$.

• En fin, pour $s = \max(k,n)$:

$A_k \circ A_n(m) \leqslant A_s \circ A_{s+1}(m)$

$= A_{s+1}(m+1)$

$\leqslant A_{s+2}(m)$

Et on a utilisé ici une dernière inégalité : $A_n(m+1) \leq A_{n+1}(m)$.

→ $m=0$ $\quad A_n(1) = A_{n+1}(0)$

→ m fixé $\quad A_n(m+2) = A_{n-1}(A_n(m+1))$

$\leq A_n(A_n(m+1))$

$\leq A_n(A_{n+1}(m))$ par hypothèse de récurrence

$= A_{n+1}(m+1)$

II] • Montrons que les fonctions récursives primitives sont bornées par un certain A_n, n donné, par induction :

• $zero(n) = 0 \leq A_0(n)$

• $succ(n) = n+1 \leq A_0(n)$

• $proj_i^k(n_1, \dots, n_k) = n_i \leq A_0(n_1 + \dots + n_k)$.

• Schéma de composition :

soit $f : \mathbb{N}^k \to \mathbb{N}$ bornée par A_n

$g_1, \dots, g_k : \mathbb{N}^p \to \mathbb{N}$ bornées chacune par $A_{n'}$.

On suppose sans perte de généralité que $n \geq n'$.

Alors pour $\vec{x} \in \mathbb{N}^p$, on a :

$f(g_1, \dots, g_k)(\vec{x}) \leq A_n(k A_{n'}(s))$ où $s = x_1 + \dots + x_k$.

$\leq A_n(A_2^k(A_{n'}(s)))$

$\leq A_{n+2(k+1)}(s)$

• Schéma de récurrence :

soit $f(0, \vec{x}) = g(\vec{x})$

$f(m+1, \vec{x}) = h(f(m, \vec{x}), m, \vec{x})$

où g et h sont bornées par A_n et $A_{n'}$.

On a alors

- $f(0,\vec{x}) = g(\vec{x}) \leq A_n(s)$

- supposons que pour m fixé, $f(m,\vec{x}) \leq A_k(m+s)$.

alors on a

$$\begin{aligned} f(m+1,\vec{x}) &= h(f(m,\vec{x}), m, \vec{x}) \\ &\leq A_{n'}(A_k(m+s)+m+s) \\ &\leq A_{n'}(A_2(A_k(m+s))) \\ &\leq A_{n'+2}(A_k(m+s)) \quad \text{en supposant } n' \geq 2 \\ &\leq A_{k-1}(A_k(m+s)) \quad \text{en supposant } n'+2 \leq k-1 \\ &= A_k(m+1+s) \end{aligned}$$

III] • Supposons maintenant la fonction d'Ackermann primitive récursive.

Alors $\varphi(m) = A(m,m+1)$ l'est aussi.

Il existe donc k tel que

$$\varphi(m) \leq A_k(m)$$

On a alors

$$\varphi(k) = A(k,k+1) = A_k(k+1) \leq A_k(k)$$

ce qui est absurde !

INSERTION DANS UN B-ARBRE.

Leçons 901, 921, 932

Référence Cormen chap. 18

Théorème

L'insertion dans un B-arbre de hauteur h se fait en $O(h)$ accès disque.
(La suppression aussi)
+ la hauteur h d'un B-arbre avec $t-1$ à $2t-1$ clés par noeud est $h = O(\log_t(n))$ où n est le nombre total de clés.

Résumé

0 - Petit mot sur B-arbre vs. arbre binaire de recherche et sur le coût des accès disques et de la recherche dans un B-arbre.
I - Etude de l'insertion dans un B-arbre.
II - Borne sur la hauteur h.
Bonus - Un mot sur la suppression.

0 · Un B-arbre c'est une arborescence T telle que :

1) Chaque noeud x contient les champs
   - $n(x)$: le nombre de clés
   - $clé_1(x) \leq \dots \leq clé_{n(x)}(x)$: les valeurs stockées dans le noeud.
   - $feuille(x)$: booléen qui dit si l'arbre est une feuille.
2) Chaque noeud x contient $c_1(x), \dots, c_{n+1}(x)$ pointeurs vers ses enfants
3) Les clés déterminent les valeurs stockées dans les noeuds enfants.
4) Toutes les feuilles ont pour profondeur h.
5) Chaque noeud a minimum $t-1$ clés (sauf la racine) et maximum $2t-1$.

- L'idée d'un B-arbre est de généraliser les ABR de façon à minimiser le nombre d'accès disque, qui sont de loin les opérations les plus coûteuses en temps, mais qui sont nécessaires pour stocker de gros volumes de données.

- Pour un B-arbre, la recherche se fait en $O(t \log_t(n))$ contre $O(\log_2(n))$ dans un ABR.
  t est effectivement une constante mais en pratique on prend $t \simeq 1000$ ce qui permet de stocker beaucoup de clés dans une petite hauteur.

Remarque : accéder à 1000 clés stockées consécutivement sur un disque dur n'est pas si long car le bras du disque bouge peu.

I] • On va avoir besoin d'une procédure qui permet de découper un noeud plein en deux noeuds.

Définissons ainsi

PARTAGER

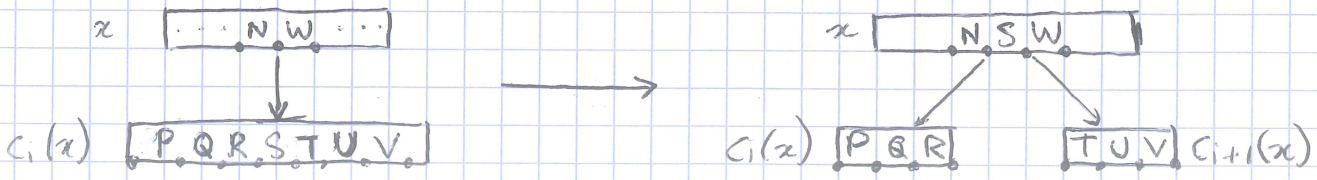
Entrées : x un noeud non plein
i un indice tel que $c_i(x)$ est plein.

1) Découper $c_i(x)$ autour de la médiane et rajouter la médiane au noeud x.

2) Ecrire sur le disque dur les mises à jour du noeud parent et des deux noeuds enfants

3 ECRITURES

Exemple avec $t = 4$.



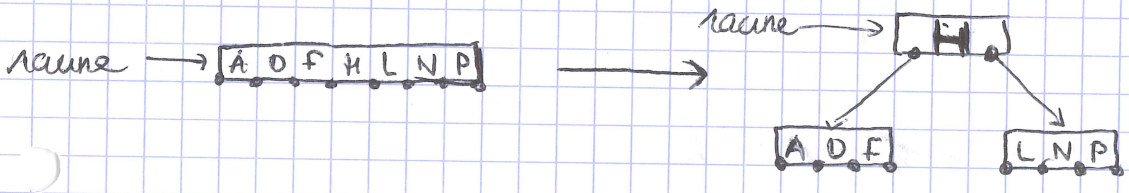
• On est maintenant armés pour insérer une nouvelle clé dans un arbre : on effectue l'insertion par le haut !

On a vu qu'on peut toujours s'en sortir si on vient d'un noeud qui n'est pas plein pour vider celui d'en dessous qui nous intéresse.

Avant de décrire notre algorithme d'insertion, on va s'assurer que la racine de T n'est pas pleine. LECTURE

→ si elle est pleine, on rajoute une racine vide au dessus, notée s, puis on appelle PARTAGER (s, 1)

Exemple avec $t=4$



• Maintenant on peut définir la procédure d'insertion.

INSERER

Entrées : x un noeud supposé non plein
k la clé à insérer.

1) Si x est une feuille, on insère simplement la clé dedans. ECRITURE

2) Sinon

• Parcourir les clés jusqu'à trouver l'enfant de x dans lequel on est susceptibles de pouvoir rajouter la clé k.

• Si ce noeud enfant est plein LECTURE

- le couper en deux en appelant PARTAGER ECRITURE
- choisir l'enfant du côté qui correspond à k.

• INSÉRER récursivement sur le noeud non plein choisi.

- Notons (ECRITURE/LECTURE) à chaque endroit où l'on a accédé au disque dur. Il y en a $O(1)$ pour chaque appel de INSERER.
→ on en effectue donc en tout $O(h)$.

II] - Tout l'intérêt des B-arbres c'est qu'on va pouvoir avoir une complexité agréable sur l'insertion : estimons h !

- Comptons le nombre minimal de [illegible] dans un B-arbre de hauteur h :

Profondeur	0	1	2	...	h
Nb min de clés	1	2	$2t$		$2t^{h-1}$

D'où la relation sur le nombre de clés :

$$n \geq 1 + \underbrace{(2 + 2t + \dots + 2t^{h-1})}_{\text{nombre minimum de noeuds}}\underbrace{(t-1)}_{\text{nombre minimum de clés par noeud.}} = 1 + 2(t-1)\frac{t^h - 1}{t-1} = 2t^h - 1$$

racine

On en déduit donc $h \leq \log_t\left(\frac{n+1}{2}\right)$.

À titre d'exemple, avec $t = 1000$ et $n = 10^9$, on a $h \leq 3$.
ce qui explique que SGL s'en serve pour faire des index :
- on accède très peu de fois au disque dur
- on peut rajouter facilement des clés

THÉORÈME DE COMPACITÉ

Leçons 916, 203

Références

c'est un ensemble de variables.

Théorème

- Soit $\mathcal{F}_0(\mathcal{P})$ l'ensemble des formules du calcul prépositionnel sur $\mathcal{P}$ et $S \subseteq \mathcal{F}_0(\mathcal{P})$ un ensemble de formules du calcul prépositionnel.
- S est insatisfiable si et seulement si il contient un sous ensemble fini insatisfiable.

+ application :

$\mathbb{Z}^2$ peut être pavé si et seulement si tous les carrés $[\![-n,n]\!]^2$ peuvent l'être.

Résumé

I - Un modèle est une interprétation (assignation de valeurs de vérités aux variables) qui rend une formule vraie.

L'ensemble des modèles d'une formule φ est :

$$I(\varphi) = \{ I \in \{0,1\}^{\mathcal{P}} \mid I \models \varphi \}$$

Par induction structurelle sur $\mathcal{F}_0(\mathcal{P})$, $I(\varphi)$ est un ouvert fermé de $\{0,1\}^{\mathcal{P}}$.

La propriété de Borel-Lebesgue donne ensuite le résultat.

II - Définition d'un ensemble de formules correspondant à la propriété :

"le plan est pavé par les tuiles d'un ensemble T"

Puis preuve par théorème de compacité

III · On va utiliser les théorèmes suivants

Théorème (Tykhonov)

Un produit de compacts est compact au sens de la topologie produit.

Propriété (Borel-Lebesgue) Pour (X,d) compact : (c'est la définition d'un compact !)

Tout recouvrement ouvert de X contient un sous recouvrement fini.

- $\{0,1\}$ est compact pour sa topologie discrète $2^{\{0,1\}}$.
  donc le théorème de Tykhonov donne la compacité de $\{0,1\}^{\mathcal{P}}$ pour la topologie produit, $\mathcal{T}$.
- Soit $\varphi \in \mathcal{F}_0(\mathcal{P})$, $I(\varphi)$ est un ouvert fermé de $\{0,1\}^{\mathcal{P}}$, on le prouve par induction structurelle sur $\mathcal{F}_0(\mathcal{P})$

  → $I(\bot) = \emptyset \in \mathcal{T}$
  $I(\bot)^c = \{0,1\}^{\mathcal{P}} \in \mathcal{T}$.

  → Pour tout $p \in \mathcal{P}$:
  $I(p) = \{1\} \times \prod_{q \neq p} \{0,1\} \in \mathcal{T}$

  $I(p)^c = \{0\} \times \prod_{q \neq p} \{0,1\} \in \mathcal{T}$

  → si $\varphi, \psi \in \mathcal{F}_0(\mathcal{P})$, supposons que $I(\varphi)$ et $I(\psi)$ sont des ouverts fermés de $\{0,1\}^{\mathcal{P}}$.
  - $I(\neg\varphi) = I(\varphi)^c$
  - $I(\varphi \vee \psi) = I(\varphi) \cup I(\psi)$

  sont des ouverts fermés de $\{0,1\}^{\mathcal{P}}$.

- Raisonnons par contraposée.
  Si S est insatisfiable, alors $\bigcap_{\varphi \in S} I(\varphi) = \emptyset$.
  La propriété de Borel-Lebesgue donne alors un sous ensemble fini S' de S tel que $\bigcap_{\varphi \in S'} I(\varphi) = \emptyset$.

  En résumé : on a extrait de S un ensemble S' fini insatisfiable !

- Enfin, la réciproque est naturelle : il suffit de compléter un ensemble fini insatisfiable, il reste insatisfiable !

Remarque : le théorème peut alors aussi s'exprimer comme :
S est satisfiable si tous ses sous ensembles finis sont satisfiables.

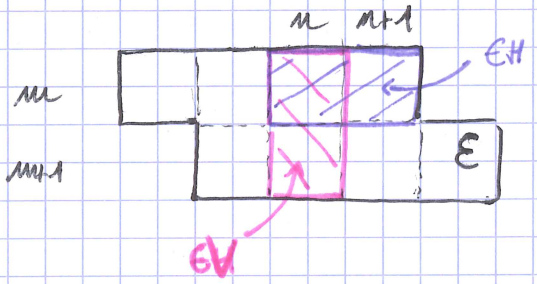
II] • Soit T un ensemble fini de tuiles.

• On définit $H, V \subseteq T^2$ les compatibilités horizontales et verticales.

• Une partie $\mathcal{E}$ de $\mathbb{Z}^2$ peut alors être pavée s'il existe $f : \mathcal{E} \to T$ telle que, pour tout $(m,n) \in \mathcal{E}$:

→ $(m+1, n) \in \mathcal{E} \Rightarrow (f(m,n), f(m+1,n)) \in H$

→ $(m, n+1) \in \mathcal{E} \Rightarrow (f(m,n), f(m,n+1)) \in V$



• Exprimons cela avec une formule du calcul propositionnel, en définissant un ensemble $S_\mathcal{E}$ de formules sur $\mathcal{P} = \mathbb{Z}^2 \times T$.

On notera $h(i,j) = (i, j+1)$ pour $i,j \in \mathbb{Z}^2$

$d(i,j) = (i+1, j)$

les voisins horizontal et vertical de (i,j).

• Dès lors, posons :

$$S_\mathcal{E} = \left\{ \left(\bigvee_{t \in T} (c,t) \right) \wedge \left(\neg \bigvee_{t \neq t' \in T} ((c,t) \wedge (c,t')) \right) \,\middle|\, c \in \mathcal{E} \right\}$$ il n'y a qu'une tuile par case

$$\cup \left\{ \bigvee_{(t,t') \in H} (c,t) \wedge (c',t') \,\middle|\, c \in \mathcal{E},\ c' = d(\mathcal{E}) \right\}$$ compatibilité horizontale

$$\cup \left\{ \bigvee_{(t,t') \in V} (c,t) \wedge (c',t') \,\middle|\, c \in \mathcal{E},\ c' = h(\mathcal{E}) \right\}$$ compatibilité verticale.

• $\mathcal{E}$ ne peut être pavé que si $S_\mathcal{E}$ est satisfiable, donc :

$S_{\mathbb{Z}^2}$ est satisfiable si et seulement si tout sous ensemble fini l'est.

Or pour $\mathcal{E} \subseteq \mathbb{Z}^2$ fini, $S_\mathcal{E} \subseteq S_{[\![-n;n]\!]^2}$ pour un certain $n \in \mathbb{N}$.

Donc si les carrés $[\![-n;n]\!]^2$, $n \in \mathbb{N}$ sont pavables, $\mathbb{Z}^2$ est pavable.

THEORÈME DE COOK

Leçons 913, 915, 928

Références Introduction à la calculabilité, Wolper p. 185

LFCC, Carton p. 203

Théorème

SAT est NP-complet

Résumé

I - SAT ∈ NP

II - Comment décrire une machine de Turing ?

III - Définition d'un ensemble de propositions pour représenter chacun des éléments de II en utilisant SAT.

I | • On peut définir un algorithme non déterministe qui :

i - génère une fonction d'interprétation de façon non déterministe

ii - vérifie que cette fonction d'interprétation rend la formule vraie.

• Donc SAT ∈ NP.

II | • Soit une machine de Turing

$$M = (Q, \Gamma, \Sigma, \Delta, s, B, F).$$

• Une exécution de Γ peut être décrite comme :

i - un tableau R de taille $(p(n)+1, p(n)+1)$ (*) qui, pour chaque configuration, donne le contenu du ruban.

tous de taille $p(n)+1$:

ii - un vecteur Q donnant, pour chaque configuration, l'état.

iii - un vecteur P donnant, pour chaque configuration, la position de la tête de lecture.

iv - un vecteur C donnant le choix non déterministe effectué par Γ à chaque étape. C'est un nombre plus petit que c = nb max de choix.

où $p(n)$ est un polynôme bornant la complexité de Π et $w = w_1 \cdots w_n$ un mot de Σ^*.

- La machine M accepte w alors si et seulement si il existe une exécution de M sur w de longueur au plus $p(n)$ qui mène à un état accepteur.

(*) notons que, la tête de lecture ne se déplaçant que d'une case à chaque étape, le nombre de symboles non blancs sur le ruban est aussi borné par $p(n)+1$.

III | • Notre objectif est de montrer qu'il existe une transformation qui, à tout mot w et à tout langage $L \in NP$, associe une instance de L_{SAT} (le langage de l'encodage des instances positives de SAT) qui est positive si et seulement si $w \in L$.

→ on veut donc associer à Π et w une instance de SAT qui est positive si et seulement si M accepte w.

• On cherche donc à produire une formule qui n'est satisfaite que par un contenu de R, Q, P et C définissant une exécution de Π qui accepte w. Introduisons alors une variable pour chaque case de ces tableaux et pour chaque valeurs qu'elles peuvent prendre :

i -	$r_{ij\alpha}$	pour $0 \leq i,j \leq p(n)$ et $\alpha \in \Gamma$
ii -	q_{ik}	pour $0 \leq i \leq p(n)$ et $k \in Q$.
iii -	p_{ij}	pour $0 \leq i,j \leq p(n)$
iv -	c_{ik}	pour $0 \leq i \leq p(n)$ et $1 \leq k \leq r$.

• Il y a ici $O(p(n)^2)$ propositions : c'est bien polynomial !

• La convention utilisée sera donc que $r_{ij\alpha} = 1$ ssi $R[i,j] = \alpha$ et $q_{ik} = 1$ ssi $Q[i] = k$.

→ il faut donc s'assurer qu'il n'y a qu'un symbole écrit par case du tableau.

• Pour cela on définit

* (1a) $\bigwedge_{0 \leq i,j \leq p(n)} \left(\left(\bigvee_{\alpha \in \Gamma} r_{ij\alpha} \right) \wedge \left(\bigwedge_{\alpha \neq \alpha' \in \Gamma} (\neg r_{ij\alpha} \vee \neg r_{ij\alpha'}) \right) \right)$

il y a un symbole en (i,j)

s'il y a un symbole, il ne peut pas y en avoir un autre.

+ on peut écrire des formules (1b) et (1c) similaires pour q et c.

Ces formules sont en FNC, de longueur $O(p(n)^2)$.

* (2) $\bigwedge_{0 \leq i \leq p(n)} \left(\left(\bigvee_{0 \leq j \leq p(n)} p_{ij} \right) \wedge \left(\bigwedge_{0 \leq j \neq j' \leq p(n)} (\neg p_{ij} \vee \neg p_{ij'}) \right) \right)$

la tête de lecture est quelque part

elle ne peut pas être à deux endroits différents

Ces formules sont en FNC, de longueur $O(p(n)^3)$.

➡ il faut s'assurer que, de plus :

- la première configuration doit être la configuration initiale :

(3) $\left(\bigwedge_{0 \leq j \leq n-1} r_{0j w_{j+1}} \wedge \bigwedge_{n \leq j \leq p(n)} r_{0jB} \right) \wedge q_{0s} \wedge p_{00}$

il est écrit w sur le ruban, suivi uniquement de blancs

état initial

tête de lecture au début.

- chaque configuration est obtenue à partir de la précédente suivant la relation de transition de la machine de Turing, c'est-à-dire

• les cases n'étant pas sous la tête de lecture ne sont pas modifiées :

(4a) $\bigwedge_{\substack{0 \leq i < p(n) \\ 0 \leq j \leq p(n) \\ \alpha \in \Gamma}} \left((r_{ij\alpha} \wedge \neg p_{ij}) \supset r_{(i+1)j\alpha} \right)$ soit $\bigwedge_{\substack{0 \leq i < p(n) \\ 0 \leq j \leq p(n) \\ \alpha \in \Gamma}} (\neg r_{ij\alpha} \vee p_{ij} \vee r_{(i+1)j\alpha})$

soit il est écrit autre chose qu'α, soit il y a la tête de lecture, soit il y a α après.

• la case sous la tête de lecture, l'état et la position de la tête de lecture sont modifiés en adéquation avec Δ, la relation de transition.
On remarque que grâce à c_{ik}, la relation s'exprime bien :

dans la configuration i
l'état k
la position de la tête de lecture j
le symbole lu α
le choix non déterministe

Δ précise alors de façon unique
l'état suivant k'
le symbole à écrire α'
le déplacement de la tête $d \in \{-1, 1\}$.

Cela donne :

(4b) $\bigwedge_{\substack{0 \le i < p(n) \\ 0 \le j \le p(n) \\ \alpha \in \Gamma \\ k \in Q \\ 1 \le l \le r}}$

$((q_{ik} \wedge p_{ij} \wedge r_{ij\alpha} \wedge c_{il}) \Rightarrow q_{(i+1)k'})$ ← l'état suivant est k'
$\wedge ((q_{ik} \wedge p_{ij} \wedge r_{ij\alpha} \wedge c_{il}) \Rightarrow r_{(i+1)j\alpha'})$ ← on écrit α'
$\wedge ((q_{ik} \wedge p_{ij} \wedge r_{ij\alpha} \wedge c_{il}) \Rightarrow p_{(i+1)(j+d)})$ ← la tête se déplace de $+1$ ou -1.

sachant que $(a \wedge b \wedge c \wedge d) \Rightarrow e$ s'écrit $\neg a \vee \neg b \vee \neg c \vee \neg d \vee e$
Les formules (4a) et (4b) ont une longueur $\mathcal{O}(p(n)^2)$.

→ enfin, il faut arriver dans un état acceptant :

(5) $\bigvee_{\substack{0 \le i \le p(n) \\ k \in F}} q_{ik}$.

• <u>Conclusion</u> la conjonction des formules (1a), (1b), (1c), (2), (3), (4a), (4b) et (5) est satisfaisable si et seulement si Π accepte w : une [illegible] la satisfaisant donne une exécution de Π acceptant w et inversement.
De plus, les c_{ik} donnent les choix faits dans l'exécution.
Enfin, la longueur de la formule est $\mathcal{O}(p(n)^3)$ donc elle peut être construite en temps polynomial. Elle doit ensuite être encodée, ce qui se fait aussi en temps polynomial : on a donc une transformation poly de L∈NP dans L_{SAT}.

DISTANCE D'EDITION ET FACTEURS

Leçons 907, 921

Référence Crochemore

Navarro, Flexible pattern matching in strings.

Théorème

Soit Σ un alphabet

$s, t \in \Sigma^*$.

On définit la distance d'édition de s à t comme le nombre d'opérations élémentaires (insertion, suppression, substitution) pour transformer l'une en l'autre.

On peut calculer cette distance en $O(|s| \cdot |t|)$ par programmation dynamique.

De plus, l'algorithme peut facilement être adapté pour rechercher les occurences à distance au plus k de s dans t en temps $O(k|t|)$.

Résumé

I - Relation de récurrence et sous-problème optimal.

II - Adaptation pour les facteurs.

III - Calcul de la complexité.

I - Un alignement de s et t est une façon d'observer leur similarité, de la forme, par exemple :

$$\begin{pmatrix} A & C & G & - & - & A \\ A & T & - & C & T & A \end{pmatrix}$$

(T : substitution, - : suppression, T : insertion)

C'est un mot sur $(\Sigma \cup \{\varepsilon\})^2 \setminus \{(\varepsilon, \varepsilon)\}$.

Le coût est le nombre d'opérations.

La distance d'édition $d(s,t)$ est donc

$$d(s,t) = \min_{\text{alignements}} \text{coût}$$

On a alors le sous-problème optimal suivant :

si α est un alignement optimal de longueur n,
$\alpha[1,\dots,n-1]$ est un alignement optimal.

Cela permet de définir la relation de récurrence :

$$\begin{cases} d(\varepsilon,\varepsilon) = 0 \\ d(x,\varepsilon) = |x| \\ d(\varepsilon,y) = |y| \\ d(x,y) = \min \begin{cases} d(x[1,\dots,|x|-1],y) + 1 \\ d(x,y[1,\dots,|y|-1]) + 1 \\ d(x[1,\dots,|x|-1],y[1,\dots,|y|-1]) + \delta_{x[|x|],y[|y|]} \end{cases} \end{cases}$$

On peut le traduire en algorithme :

```
DISTANCE(s,t):
    D = [[0]]
    Pour i = 0 à |s| :   D[i][0] = i
    Pour j = 0 à |t| :   D[0][j] = j
    Pour i = 1 à |s| :
        Pour j = 1 à |t| :
            δ = 0
            Si s[i] ≠ s[j] : δ = 1
            D[i][j] = min(D[i-1][j]+1, D[i][j-1]+1, D[i-1][j-1]+δ)
    Renvoyer D[|s|][|t|].
```

$\Rightarrow$ Complexité $O(|s| \cdot |t|)$.

Exemple

	ε	b	a	a	c	b
ε	(0,0)	(1,0)	2,0	3,0	4,0	5,0
a	1,1	1,1	(1,0)	2,0	3,1	4,1
b	2,2	1,1	2,1	(2,1)	3,2	3,1
c	3,3	2,2	2,2	3,2	(2,1)	3,2
a	4,4	3,3	2,2	2,2	(3,2)	4,2
b	5,5	4,4	3,3	3,3	3,3	(3,2)

→ $\begin{pmatrix} b & a & a & c & - & b \\ - & a & b & c & a & b \end{pmatrix}$

on obtient un alignement en se rappelant des choix qu'on a faits.

II) On peut adapter l'algo pour rechercher des facteurs ~~d'édition~~ à distance d'édition $\leq k$ par exemple.

Pour cela on remarque que le mot vide est facteur à distance d'édition nulle de tout mot.

→ on change donc juste $d(\varepsilon, y) = 0$.

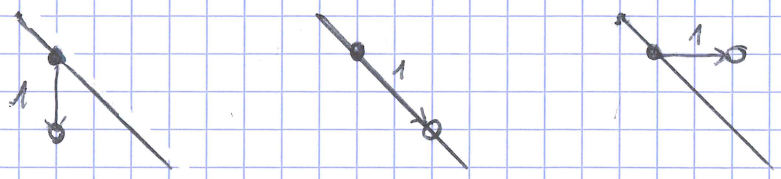
On trouve donc tous les facteurs en $O(|s| \cdot |t|)$

III) En fait on peut faire mieux. Pour cela on définit les cellules actives : une cellule du tableau est active si elle a une valeur $\leq k$.

La position de la cellule active la plus basse ne peut alors être incrémentée que de 1 à chaque lettre du texte, dans le cas où on lit deux fois la même lettre.

III] On a besoin de définir deux types de nœuds : sur chaque diagonale orientée ↘ :

* un d-nœud : la dernière paire (i,j) telle que $D[i][j] = d$ sur une diagonale
* un d-nœud spécial : un nœud que l'on peut atteindre en un coup depuis un $(d-1)$-nœud.



On doit ensuite descendre sur les diagonales selon le plus long chemin de poids 0 pour trouver les d-nœuds.

→ c'est-à-dire, pour un d-nœud spécial (i,j), on cherche le plus long préfixe de $s[i-|s|]$ commun à $t[j-|t|]$.

Comme il y a $|t|$ diagonales, on peut passer des $(d-1)$-nœuds aux d-nœuds en $O(|t|)$. Ensuite, on peut descendre le long d'une diagonale en $O(1)$ car cela revient à chercher le dernier ancêtre commun de deux suffixes dans l'arbre des suffixes communs à s et t (*)

De plus le cas $d=0$ correspond à une recherche sans erreur, qui se fait en temps linéaire.

(*) La construction linéaire de l'arbre se fait avec l'algorithme d'Ukkonen, mais c'est pas évident, et après il faut calculer les ancêtres communs en temps linéaire, ce qui en rajoute une couche.

ALGORITHME DE FLOYD-WARSHALL

Leçons 925, 931

Références Cormen p. 609

Types de données et algorithmes, Froidevaux p. 477

Théorème

Soit $G = (S, A)$ un graphe et $w : A^2 \to \mathbb{R}$ une fonction de poids. Si G n'a pas de cycle de poids négatif, on peut trouver toutes les plus courtes distances entre deux sommets en $O(|S|^3)$, et, s'il a un tel cycle, on le trouve aussi.

Résumé

I – Structure d'un plus court chemin : sommets intermédiaires plus petits que k.

II – Une relation de récurrence sur la distance n'utilisant que les k premiers sommets

III – Calcul des distances par programmation dynamique.

Remarque si $u \xrightarrow{p_1} v \xrightarrow{p_2} w$ est un plus court chemin, alors p_1 et p_2 sont aussi des plus courts chemins : c'est ce qui va nous permettre de construire l'algorithme.

I | • Pour $p = \langle v_1, v_2, \dots, v_p \rangle$, les sommets v_2 à v_{p-1} sont intermédiaires.

• Supposons que $S = [\![1 \dots n]\!]$, on pose $S_k = [\![1 \dots k]\!]$.

→ on s'intéresse aux chemins de i à j ne passant que par des sommets de S_k.

Pour un chemin $u \xrightarrow{p} v$ dans S_k, il y a deux possibilités :

* soit k n'est pas un sommet intermédiaire de p, ces sommets sont donc tous dans S_{k-1} et le chemin est le même dans S_{k-1} et S_k.
* soit k est un sommet intermédiaire de p, et on peut diviser p en $u \xrightarrow{p_1} k \xrightarrow{p_2} v$. p_1 et p_2 n'ont alors que des sommets intermédiaires dans S_{k-1} et sont les plus courts chemins $u \to k$ et $k \to v$.

II] - Posons alors $d_{ij}^{(k)}$ le poids d'un plus court chemin de i à j dont tous les sommets intermédiaires sont dans S_k.

- La remarque de I] donne la relation de récurrence :

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{si } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{si } k \geq 1. \end{cases}$$

$d_{ij}^{(k-1)}$: on ne passe pas par k

$d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$: on passe par k.

si $k = 0$: il n'y a pas d'intermédiaire, c'est donc une arête de G !

- En itérant, on trouve $d_{ij}^{(n)}$, la distance entre i et j qui s'autorise de passer par tous les sommets de G.

III] - On peut ainsi définir l'algorithme de Floyd-Warshall ainsi, prenant en entrée la matrice $W = (w_{ij})_{i,j}$, coïncidant avec $(d_{ij}^{(0)})_{i,j}$:

```
FLOYD-WARSHALL(W)
    n = nombre de lignes de W.
    (d_ij^(0))_ij = (w_ij)_ij
    Pour k = 1 à n
        Pour i = 1 à n
            Pour j = 1 à n
                d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
    Renvoyer D = (d_ij^(n))_i,j
```

Le temps d'exécution de cet algorithme est $O(1)$ par passage dans la boucle, c'est-à-dire $O(|S|^3)$.
Et il est correct par définition de la relation de récurrence sur $d^{(k)}$.

Remarque on n'utilise qu'un tableau de taille fixe, sans opération ni structure de données élaborée : la constante du O est donc petite, ce qui rend l'algorithme intéressant même sur des graphes de taille modérée.

Knuth-Morris-Pratt.

Leçons : 907, 927.

Ref :

Théorème

L'algorithme de recherche (K)MP est correct et termine en $O(m)+n)$, où l'on cherche un mot de longueur m dans un mot de longueur n.

Résumé

- On cherche $u \in \Sigma^*$ dans $t \in \Sigma^*$ sur un alphabet Σ.

I - Déroulement de l'algorithme sur un exemple significatif.

II - Définition de l'automate des occurences, reconnaissant $\Sigma^* u$.

III - Étude de la fonction Bord pour un calcul efficace de l'automate.

IV - Algorithme de Morris-Pratt + l'amélioration de Knuth.

I | • Prenons $t = ABADABABAC$

$u = ABAC$.

• L'algorithme de Morris-Pratt donne :

$t = A\ B\ A\ D\ A\ B\ A\ B\ A\ C \ \ldots$

1) $A^{\checkmark}\ B^{\checkmark}\ A^{\checkmark}\ C^{\times}$

2) $A^{\checkmark}\ B^{\times}$

3) $A^{\times}$

4) $A^{\checkmark}\ B^{\checkmark}\ A^{\checkmark}\ C^{\checkmark}$

• L'idée, c'est que quand il y a un problème, on peut décaler la fenêtre de plus d'un caractère.

• On se rend compte que le motif déjà matché avant décalage, c'est à la fois un préfixe de u et un suffixe de $t[k+1]$.

II] · Le déroulement de KMP peut être vu comme le parcours de l'automate suivant : $A = (Q, \Sigma, u, \delta)$ où :

* Q est l'ensemble $Pref(u)$ des préfixes de u. "ce qu'on a déjà matché".
* ε (le mot vide) est l'état initial "on n'a encore rien matché".
* la fonction de transition δ est :

$$\delta(v,a) = \begin{cases} va & \text{si } va \in Pref(u) \\ Bord(va) & \text{sinon.} \end{cases} \quad \text{où } Bord(w) \text{ est le plus long bord strict de } w.$$

* u l'état final : "on a reconnu le mot entier".

· Ce qu'il se passe, c'est que l'état indique la partie du mot qu'on a déjà matché, puis :

→ soit on lit la lettre a qui suit v dans u, et on continue vers va.

→ soit on lit une autre lettre a, et là, on veut éviter de recommencer depuis ε, donc on recommence après un préfixe de va, qu'on aurait déjà reconnu à la fin de ce que l'on vient de lire, c'est-à-dire à la fin de va. (sans que ça soit va !)
C'est donc $Bord(va)$ = le plus long bord strict de va.

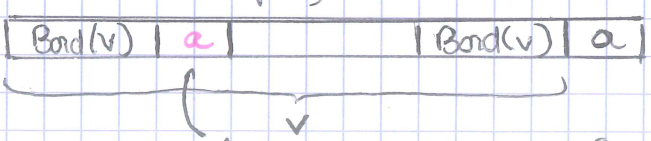
III] · Mais comment calculer $Bord(va)$ efficacement ?

· Grâce au lemme :

$$Bord(va) = \begin{cases} Bord(v)\,a & \text{si } Bord(v)\,a \in Pref(u) \\ Bord(Bord(v)\,a) & \text{sinon.} \end{cases}$$

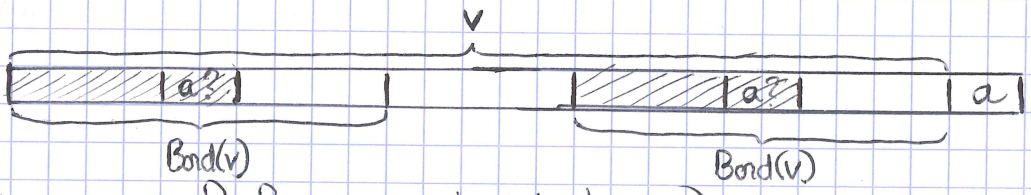
· Intuitivement, on a : → si $va \in Pref(u)$ on n'a qu'à continuer.

→ si $Bord(v)\,a \in Pref(u)$, on a :



il y a un a ici car $Bord(v)\,a \in Pref(u)$ et $v \in Pref(u)$.

→ sinon, $Bord(v)a \notin Pref(u)$, et on cherche un préfixe de u qui finisse par a. C'est-à-dire un mot au début de v, suivi d'un a, qui apparait aussi à la fin de va ... et donc de $Bord(v)a$. (*)



(Les deux zones hachurées sont identiques)

(*) Si ce n'était pas dans $Bord(v)a$, on serait dans le premier cas car on aurait un mot, bord de va, finissant par a, et donc un bord de v plus long que $Bord(v)$, c'est-à-dire ... $Bord(v)$!

• On peut maintenant prouver le lemme :

- soit $w = Bord(va)$, il s'écrit $w'a = w$ car c'est un suffixe de va.
  → $w \in Pref(va)$ donc $w' \in Pref(v)$
  → $w \in Suff(va)$ donc $w' \in Suff(v)$
  ⇒ de plus, w est maximal donc w' est le plus grand bord de v tel que $w'a$ est un préfixe de u.
- cela laisse deux possibilités :
  1 - $Bord(v)a \in Pref(u)$, auquel cas $w' = Bord(v)$ par maximalité.
  2 - w' est un préfixe strict de $Bord(v)$, et comme $w'a = Bord(va)$, w' est aussi à la fin de v et donc de $Bord(v)$.
  De là, w' est un préfixe de $Bord(Bord(v))$, d'où $w = Bord(Bord(v)a)$.

• En itérant ce lemme, on obtient

$$\text{Bord}(va) = \begin{cases} \text{Bord}(v)a & \text{si } \text{Bord}(v)a \in \text{Pref}(u) \\ \text{Bord}(\text{Bord}(v))a & \text{sinon si } \text{Bord}(\text{Bord}(v))a \in \text{Pref}(u) \\ \cdots & \\ a & \text{sinon si } a \in \text{Pref}(u) \\ \varepsilon & \text{sinon tout court.} \end{cases}$$

III] • On peut traduire tout ça en algorithme directement, mais pour cela, on va seulement garder en mémoire la taille du bord des préfixes de la forme $u[1 \dots k]$.
On définit aussi $\pi(k) = |\text{Bord}(u[1 \dots k])|$.

```
CalculPi(u) :
    π(1) ← 0        k ← 0.
    Pour i = 2 ... |u| :
        Tant que k > 0 et u[k+1] ≠ u[i] :
            k ← π(k).
        Si u[k+1] = u[i] :
            k ← k+1
        π(i) ← k.
```

• Terminaison : on a $\pi(k) < k$ donc le while termine et CalculPi aussi.
• Correction : c'est le lemme 1, on calcule bien les bords !
• Complexité : on remarque que k est toujours positif, le while ne peut pas désincrémenter k de plus que sa valeur.
De plus, k augmente de 1 au plus une fois par passage dans le for, dans lequel on passe n fois. On passe donc moins de n fois dans le while, d'où une complexité en $O(n)$, avec $n = |u|$.

- On peut (enfin) écrire l'algorithme de recherche : il suffit de chercher un bord de $u\#t$, pour $\# \notin \Sigma$, de taille $|u|$.

On a en effet :

> $\forall k \geq 0$, $\pi(k) \leq |u|$ et $\pi(k) = u$ ssi u apparaît dans t en position $k - 2|u| - 1$.

- Car, d'une part, $\# \notin \Sigma$, donc $\pi(k) \leq |u|$ sinon on aurait une occurence de $\#$ dans t.

- D'autre part, si $\pi(k) = |u|$, alors u est un préfixe et... un suffixe de $u\#t[1 \ldots k]$. C'est donc un suffixe ~~de t~~ de $t[1 \ldots k - |u| - 1]$ en position $k - 2|u| - 1$.

→ on trouve donc une occurence de u dans t en :

$$O(|u| + |t| + 1) = O(|u| + |t|).$$

THÉORIE DES ORDRES DENSES.

Leçons : 914, 918, 924.

Référence : David Nour Raffali (DNR) p. 130

Théorème

- La théorie des ordres T_0 est écrite au le langage $\mathcal{L}_0 = \{<, =\}$ avec les axiomes :

  $(\mathcal{O}_1)$ $\forall x,y \quad \neg\{x<y \wedge y<x\}$ } ordre strict

  $(\mathcal{O}_2)$ $\forall x,y,z \quad \{x<y \wedge y<z \longrightarrow x<z\}$ } ordre strict

  $(\mathcal{O}_3)$ $\forall x,y \quad \{x<y \vee x=y \vee y<x\}$ } ordre total

  $(\mathcal{O}_4)$ $\forall x,y \exists z \quad \{x<y \longrightarrow x<z \wedge z<y\}$ } ordre dense

  $(\mathcal{O}_5)$ $\forall x \exists y \quad \{x<y\}$ } sans plus petit/grand élément.

  $(\mathcal{O}_6)$ $\forall x \exists y \quad \{y<x\}$ } sans plus petit/grand élément.

- Remarques : Les modèles de T_0 sont infinis (pas de plus grand élément)
  ne sont pas tous isomorphes ($\mathbb{Q}$ et $\mathbb{R}$)
  existent (T_0 non contradictoire)

- La théorie T_0 est complète et décidable.

Résumé

I - Théorème d'élimination des quantificateurs (admis) et lemme associé.

II - T_0 admet l'élimination des quantificateurs (par le lemme)

i - enlever les négations des atomes de la formule

ii - écrire la formule sous forme disjonctive et décrire les atomes.

iii - "distribuer" les $\exists$ et étudier les conjonctions d'atomes

I] **Théorème** (admis) : Soit T une théorie qui admet l'élimination des quantificateurs. Si pour tout A formule atomique close, $T \vdash A$ ou $T \vdash \neg A$, alors T est complète.

De plus, si T est récursive sur un langage au plus dénombrable, alors T est décidable.

Lemme T admet l'élimination des quantificateurs si pour toute formule sans quantificateurs $F[x, x_1, \dots, x_n]$ il existe une formule sans quantificateurs $G[x_1, \dots, x_n]$ telle que

$$T \vdash \forall x_1, \dots, x_n \, (\exists x \, F \leftrightarrow G).$$

Preuve : On peut écrire F en n'utilisant que $\exists$, $\vee$ et $\neg$.

Par récurrence sur la taille de F il existe une formule G telle que

$$VL(G) \subseteq VL(F) \quad \text{et} \quad T \vdash F \leftrightarrow G.$$

En effet, pour $\vee$ et $\neg$ il n'y a pas de problème, regardons le $\exists$: c'est exactement l'hypothèse du lemme.

II] • En remplaçant les formules de la forme $A \to B$, $\neg(A \wedge B)$, $\neg(A \vee B)$ et $\neg\neg A$ par leurs homologues $\neg B \vee A$, $\neg A \vee \neg B$, $\neg A \wedge \neg B$ et A, puis en distribuant les $\wedge$, on peut réécrire la formule $F[x, x_1, \dots, x_n]$ sous la forme disjonctive :

$$\bigvee_k \bigwedge_\ell H_{k\ell}$$

• Décrivons la forme des $H_{k\ell}$, en remarquant que

$$T_0 \vdash \neg(x = y) \leftrightarrow x < y \vee y < x$$

$$T_0 \vdash \neg(x < y) \leftrightarrow x = y \vee y < x$$

on peut donc supposer qu'il n'y a pas de négation dans $H_{k\ell}$.

- H_{kl} est donc de la forme :

$$x=x \mid x=x_i \mid x_i=x_j \mid x_i<x_j$$
$$\mid x_i<x \mid x<x_i \mid x<x \mid \top \mid \bot .$$

- On peut éliminer $x=x$, $x<x$, $x_i=x_i$, $x_i<x_i$ qui sont équivalentes à $\top$ ou $\bot$.

- Mais on peut aussi éliminer $\top$ et $\bot$ car :

$$\vdash \top \wedge A \leftrightarrow A \quad \text{et} \quad \vdash \bot \wedge A \leftrightarrow \bot$$

- Enfin $\vdash \exists x\, A \vee B \leftrightarrow \exists x A \vee \exists x B$, il suffit qu'une des formules $\exists x \bigwedge_r K_r$ puisse être écrite sans quantificateur. où les K_r sont de la forme :

$$x=x_i \mid x_i=x_j \mid x_i<x_j \mid x_i<x \mid x<x_i$$

- Distinguons plusieurs cas :

* si K contient un $x=x_i$, $\exists x K$ équivaut à $K[x:=x_i]$

* sinon, on peut écrire la formule équivalente à $\exists x K$:

$$K_1 \wedge \exists x K_2 = \left(\bigwedge_n K_{1,n}\right) \wedge \left(\exists x \bigwedge_m K_{2,m}\right)$$

où K_1 contient les formules atomiques de la forme $x_i=x_j$, $x_i<x_j$. et K_2 celles de la forme $x<x_i$, $x_i<x$

Il existe alors des ensembles d'indices I, J tels que K_2 équivaut à

$$\exists x \left[\left(\bigwedge_{i \in I} x<x_i \right) \wedge \left(\bigwedge_{j \in J} x_j<x \right) \right]$$

Dès lors : × si $I \cap J \neq \emptyset$, la formule équivaut à $\bot$

× sinon, si $I \neq \emptyset$ et $J \neq \emptyset$, la formule équivaut à $\bigwedge_{i \in I, j \in J} x_j < x_i$ par densité de l'ordre.

× si $I=\emptyset$ ou $J=\emptyset$, la formule équivaut à $\top$ par (Θ_5) et (Θ_6).

On a donc bien vérifié les hypothèses du lemme, donc T_0 admet l'élimination des quantificateurs, et comme $\mathcal{L}_0$ est fini,

T_0 est complète et décidable.

DÉCIDABILITÉ DE L'ARITHMÉTIQUE DE PRESBURGER.

Leçons : 909, 914, 924.

Références : Carton, Langages formels Calculabilité et Complexité p. 179.

Théorème

L'arithmétique de Presburger est la théorie du premier ordre des entiers munis de l'addition mais pas de la multiplication.
Cette théorie est décidable.

Note

Pour la culture, l'arithmétique de Presburger est la théorie contenant les symboles +, 0 et 1, ainsi que les axiomes suivants :

- $\forall x \quad \neg(0 = x+1)$
- $\forall x,y \quad (x+1 = y+1) \Longrightarrow x = y$.
- $\forall x \quad x+0 = x$
- $\forall x,y \quad (x+y)+1 = x+(y+1)$.
- $\forall \bar{x} \; (P(0,\bar{x}) \wedge (\forall y \, P(y,\bar{x}) \to P(y+1,\bar{x}))) \longrightarrow \forall y \, P(y,\bar{x})$ pour toute formule $P(y,x_1,\dots,x_n)$.

Résumé (On montre que le langage des n-uplets qui satisfont une formule φ est rationnel)

I – Soit φ une formule close que l'on écrit sous forme prénexe :

$\varphi = Q_1 x_1 Q_2 x_2 \dots Q_n x_n \psi$ où $Q_1, \dots, Q_n$ sont des quantificateurs.

On encode ensuite une séquence d'arguments pour définir X_k.

II – Automate qui reconnait le langage correspondant à ψ.

III – On prouve par récurrence qu'on peut rajouter les quantificateurs.

IV – Un exemple : automate pour $x \equiv 0 \bmod 3$.

I/ On a $\varphi = Q_1 x_1 \cdots Q_n x_n \psi$.

• Définissons $\varphi_k = Q_{k+1} x_{k+1} \cdots Q_n x_n \psi$.

avec $\varphi_0 = \varphi$ et $\varphi_n = \psi$.

Dans φ_k, $x_1, \dots, x_k$ sont libres et on écrit donc $\varphi_k(x_1, \dots, x_k)$.

• Pour reconnaître φ_k, on a besoin d'utiliser un codage. Pour cela, on écrit chacun des $x_1, \dots, x_k$ en binaire (ou $\Sigma = \{0,1\}$).

Un k-uplet d'entiers peut alors s'écrire ou Σ_2^k, quitte à ajouter des zéros à la fin pour avoir des nombres de même longueur:

$$(1,4,10) \text{ s'écrit } \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \in (\Sigma^k)^*.$$

• On peut alors définir le langage reconnu par φ_k:

$$X_k = \left\{ (x_1, \dots, x_k) \in (\Sigma^k)^* \;\middle|\; \varphi_k(x_1, \dots, x_k) \text{ est vraie} \right\}.$$

II/ • Construisons l'automate $\mathcal{A}_n$ qui reconnaît X_n, ie "quand ψ est vraie ?"

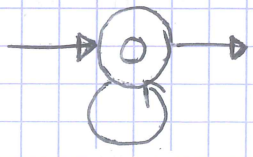
• ψ s'écrit comme une combinaisons booléennes de formules de type :

i) $x_i = x_j$ ou ii) $x_i + x_j = x_k$.

• La classe des langages rationnels est close pour les opérations booléennes. Il suffit donc de construire des automates pour les formules de type i) et ii)

i) Pour l'égalité, on définit l'automate suivant:

$x_i = x_j$:

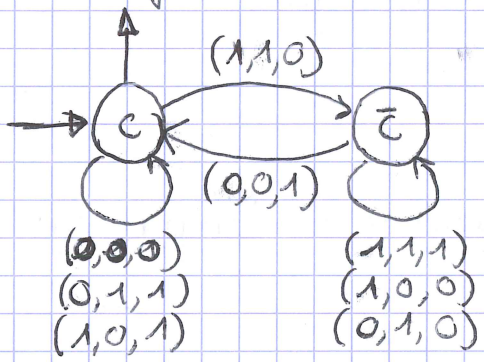


$(*, *, 1, *, \dots, 1, *, \dots, *)$
$(*, *, 0, *, \dots, 0, *, \dots, *)$

i ↑ ↑ j

ii) Et pour l'addition :

$x_i + x_j = x_k$.



où $\bar{c}$ correspond à la retenue.

(on s'épargne l'écriture des variables inutiles)

→ on a ainsi un automate qui reconnaît X_n.

III • Montrons qu'on peut rajouter un quantificateur, c'est-à-dire que étant donné $\mathcal{A}_k$ reconnaissant X_k, on peut construire $\mathcal{A}_{k-1}$ reconnaissant X_{k-1}.

■ Supposons alors que Q_k est un quantificateur d'existence $\exists$.

• Définissons avant tout l'opérateur de projection :

$$\pi_k : \begin{array}{ccc} \Sigma_k & \longrightarrow & \Sigma_{k-1} \\ (x_1, \dots, x_k) & \longmapsto & (x_1, \dots, x_{k-1}) \end{array}.$$

• Cela permet de définir l'automate $\mathcal{A}_{k-1}$ dont :

- l'ensemble d'états est le même que pour $\mathcal{A}_k$.
- les états finaux sont les mêmes que pour $\mathcal{A}_k$.
- les états initiaux sont ceux de $\mathcal{A}_k$, auxquels on rajoute les états atteints en lisant $(0, \dots, 0)$
- la fonction de transition est telle que

$$p \xrightarrow{z} q \text{ dans } \mathcal{A}_k \quad \text{ssi} \quad p \xrightarrow{\pi_k(z)} q$$

→ en quelque sorte, l'automate $\mathcal{A}_{k-1}$ devine une valeur qui conviendrait à x_k (ou plutôt il fait semblant d'oublier !), ce qui correspond bien au quantificateur ~~universel~~ existentiel

- N'oublions pas que $\exists_k$ peut aussi être un quantificateur universel $\forall$.

→ en fait, on a déjà gagné, parce que :

$$\varphi_{k-1} = \forall x_k \; \varphi_k$$
$$= \neg \exists x_k \, (\neg \varphi_k)$$

Et la clôture par complémentation des langages rationnels donne le résultat.

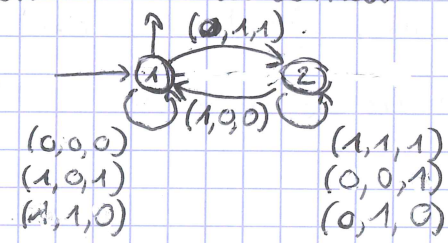
⇒ Pour finir, il suffit de remarquer que A_φ reconnaît au moins un mot si et seulement si φ est vraie !

IV • L'expression $x \equiv 0 \bmod 3$, c'est en fait l'expression :

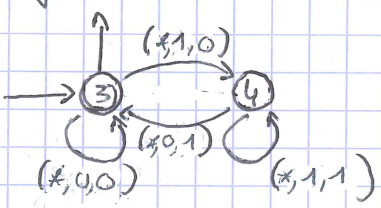
$$\exists y \; \exists z \; (x = y+z) \wedge (z = y+y).$$

Cela donne les automates suivants :

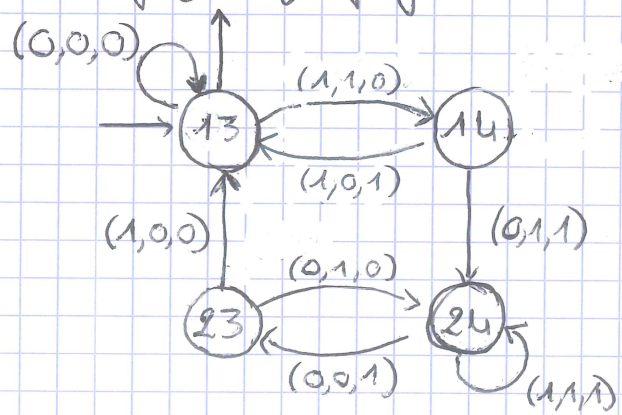
$x = y+z$



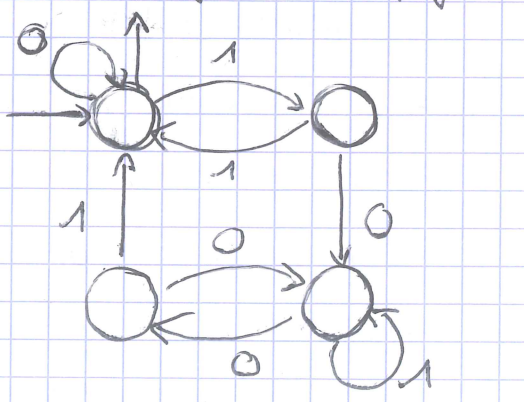
$z = y+y$



$(x = y+z) \wedge (z = y+y)$



$\exists y \exists z \; (x = y+z) \wedge (z = y+y)$



UNE INVOLUTION EST UNE BIJECTION + LOIS DE MORGAN

Leçons 918

Références

Théorème

On va montrer deux choses

I - Lois de Morgan : pour A et B deux formules du premier ordre :

$$\neg A \wedge \neg B \leftrightarrow \neg(A \vee B).$$

II - Une involution est une bijection

I • Cherchons à démontrer le résultat dans le formalisme du calcul des séquents LK, par double implication.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,ax}{A \vdash A, B}\,aff_d}{A \vdash A \vee B}\,\vee_d}{\neg(A \vee B), A \vdash}\,\neg_g \quad \dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}\,ax}{B \vdash A, B}\,aff_d}{B \vdash A \vee B}\,\vee_d}{\neg(A \vee B), B \vdash}\,\neg_g}{\neg(A \vee B) \vdash \neg A}\,\neg_d \quad \neg(A \vee B) \vdash \neg B}{\neg(A \vee B) \vdash \neg A \wedge \neg B}\,\wedge_d}{\vdash \neg(A \vee B) \longrightarrow \neg A \wedge \neg B}\,\rightarrow_d$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,ax}{\neg A, A \vdash}\,\neg_g}{\neg A, \neg B, A \vdash}\,aff_g \quad \dfrac{\dfrac{}{B \vdash B}\,ax}{\neg B, B \vdash}\,aff_g}{\neg A, \neg B, A \vee B \vdash}\,\vee_g}{\neg A \wedge \neg B, A \vee B \vdash}\,\wedge_g}{\neg A \wedge \neg B \vdash \neg(A \vee B)}\,\neg_d}{\vdash \neg A \wedge \neg B \longrightarrow \neg(A \vee B)}\,\rightarrow_d$$

II • On a besoin de l'égalité, pour cela on doit rajouter
→ l'élimination de l'égalité à gauche et à droite

$$\frac{\Gamma \vdash A[x:=t] \wedge (t=u), \Delta}{\Gamma \vdash A[x:=u], \Delta} =d \qquad \frac{\Gamma, A[x:=t] \wedge (t=u) \vdash \Delta}{\Gamma, A[x:=u] \vdash \Delta} =g$$

→ l'introduction de l'égalité comme axiome :

$$\frac{}{\Gamma \vdash t=t}\ ax$$

où t et u sont des termes du langage.

• Soit f un symbole d'arité 1, on peut définir

→ f est une involution : $Inv(f) = \forall x\ f f x = x$
→ f est injective : $Inj(f) = \forall x \forall y\ fx = fy \longrightarrow x = y$
→ f est surjective : $S(f) = \forall y \exists x\ fx = y$
→ f est bijective : $B(f) = Inj(f) \wedge S(f)$.

• Si l'on trouve des dérivations pour $\overset{Inv(f)\vdash}{Inj(f)}$ et $\overset{Inv(f)\vdash}{S(f)}$, on a montré le résultat car :

$$\dfrac{\dfrac{Inv(f) \vdash Inj(f) \quad Inv(f) \vdash S(f)}{Inv(f) \vdash Inj(f) \wedge S(f)} \wedge d}{\vdash Inv(f) \mapsto Inj(f) \wedge S(f)} \longrightarrow d$$

• Puis $Inv(f) \leftrightarrow S(f)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{ffy = y \vdash ffy = y}\ ax}{Inv(f) \vdash ffy = y}\ \forall g\ (y/x)}{Inv(f) \vdash \exists x\ fx = y}\ \exists d\ (fy/x)}{Inv(f) \vdash S(f)}\ \forall d}{\vdash Inv(f) \longrightarrow S(f)} \longrightarrow d$$

• Enfin, $Inv(f) \longrightarrow Inj(f)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x=ffx \vdash x=ffx}\,ax}{Inv\,f \vdash x=ffx}\,\forall_g}{Inv\,f, fx=fy \vdash x=ffx}\,\forall_g \quad \dfrac{\dfrac{\dfrac{}{y=ffy \vdash y=ffy}\,ax}{Inv\,f \vdash y=ffy}\,\forall_g}{Inv\,f, fx=fy \vdash y=ffy}\,\forall_g}{Inv\,f, fx=fy \vdash x=ffx \wedge y=ffy}\,\wedge d \quad \dfrac{\dfrac{\dfrac{}{fx=fy \vdash fx=fy}\,ax \quad \dfrac{}{fx=fy \vdash fx=fy}\,ax}{fx=fy \vdash (fx=fy) \wedge (fx=fy)}\,\wedge d}{fx=fy \vdash (fx=fy)[z:=fy] \wedge (fx=fy)}\,=d \quad \dfrac{\dfrac{}{fx=fy \vdash fx=fy}\,ax}{Inv\,f, fx=fy \vdash ffx=ffy}\,\forall_g}{Inv\,f, fx=fy \vdash (y=z)[z:=ffy] \wedge ffx=ffy}\,\wedge d \quad \dfrac{}{Inv\,f, fx=fy \vdash y=ffx}\,=d$$

$$\dfrac{Inv\,f, fx=fy \vdash x=ffx \wedge y=ffx}{Inv\,f \vdash fx=fy \longrightarrow (x=z)[z:=fx] \wedge (y=fx)}\,\longrightarrow d \quad =d$$

$$\dfrac{Inv\,f \vdash fx=fy \longrightarrow x=y}{Inv\,f \vdash Inj\,f}\,\forall d, \forall d$$

C'est encore un peu court comme ça, on peut montrer la commutativité de l'égalité après l'avoir introduite :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash y=y}\,ax}{x=y \vdash y=y} \quad \dfrac{}{x=y \vdash x=y}\,ax}{x=y \vdash (y=z)[z:=y] \wedge x=y}\,\wedge d \quad =d}{x=y \vdash y=x}\,\longrightarrow d$$

$$\dfrac{\vdash x=y \longrightarrow y=x}{\vdash \forall x\, \forall y\ x=y \longrightarrow y=x}\,\forall d \forall d$$

LES FONCTIONS RÉCURSIVES PRIMITIVES SONT λ-DÉFINISSABLES

Leçons 912, 929.

Références Lassaigne, Rougemont - Logique et fdt de l'info p.193
Odifreddi - Classical Recursion Theory p.84.

Théorème

Les fonctions primitives récursives sont λ-définissables.

Résumé

0 - Entiers de Church
Définition de λ-définissable.
I - Fonctions élémentaires
II - Composition
III - Récursion : combinateurs de points fixes

0 • Les entiers de Church sont les $\underline{n} = \lambda f x. \underbrace{f \dots f}_{n \text{ fois}} x$

• On dit qu'un λ-terme F représente la fonction $f : \mathbb{N}^n \to \mathbb{N}$
si $\forall k_1, \dots, k_n \in \mathbb{N}$ $f(k_1, \dots, k_n) = k \Rightarrow F\, \underline{k_1} \dots \underline{k_n} =_\beta \underline{k}$.

I • Écrivons les fonctions élémentaires :

- Zéro : $\lambda x. \underline{0}$.
- Projection : $\lambda x_1 \dots x_n. x_i$
- Successeur : $\lambda z f x. f z f x$

car alors $(\lambda z f x. f z f x)(\lambda g y. g^n y)$
$= \lambda f x. f (\lambda g y. g^n y) f x$
$= \lambda f x. f f^n y$
$= \underline{n+1}$

III] • Schéma de composition :

Soit $h : \mathbb{N}^n \to \mathbb{N}$

$g_1, \dots, g_n : \mathbb{N}^p \to \mathbb{N}$ récursives primitives.

On les représente par $H, G_1, \dots, G_n$, et on pose

$$\begin{aligned} F &= \lambda x_1 \dots x_p . H (G_1 \vec{x}) \dots (G_n \vec{x}) \\ &= H\ g_1(\vec{x}) \dots g_n(\vec{x}) \\ &= \underline{h(g_1(\vec{x}), \dots, g_n(\vec{x}))} \end{aligned}$$

IV] • La récurrence, c'est un petit peu compliqué, on va utiliser un combinateur de point fixe Y :

$$Y = \lambda f . (\lambda x . f\,x\,x)(\lambda x . f\,x\,x)$$

On a alors pour tout terme F :

$$\begin{aligned} YF &= (\lambda x . F\,x\,x)(\lambda x . F\,x\,x) \\ &= F\,(\lambda x . F\,x\,x)(\lambda x . F\,x\,x) \\ &= F\,(YF) \end{aligned}$$

• On va aussi admettre qu'il existe une fonction λ prédécesseur P telle que $\begin{cases} P\,\underline{k+1} = \underline{k} \\ P\,\underline{0} = \underline{0} \end{cases}$

• Prenons maintenant des fonctions f, g, h récursives primitives telles que

$$f(\vec{x}, n) = \begin{cases} g(\vec{x}) & \text{si } n = 0 \\ h(\vec{x}, n-1, f(\vec{x}, n-1)) & \text{sinon} \end{cases}$$

où g et h sont récursives primitives représentées par les termes G, H.

Cela peut se traduire par la quête du λ-terme vérifiant :

$$\begin{cases} R\,u\,v\,\underline{0} = u \\ R\,u\,v\,\underline{k+1} = v\,\underline{k}\,(R\,u\,v\,\underline{k}) \end{cases}$$

C'est l'opérateur de récursion !

Avec un tel terme, on peut représenter f par :

$$F = \lambda \vec{x}\, y.\ R\,(G\vec{x})\,(\lambda ab.\ H\,\vec{x}\,ab)\,y.$$

Car on a alors

- $F\,\vec{x}\,\underline{0} = R\,(G\vec{x})\,(\lambda ab.\ H\vec{x}ab)\,\underline{0}$
  $= G\vec{x}$

- $F\,\vec{x}\,\underline{k+1} = R\,(G\vec{x})\,(\lambda ab.\ H\vec{x}ab)\,\underline{k+1}$
  $= (\lambda ab.\ H\vec{x}ab)\,\underline{k}\,(R\,(G\vec{x})\,(\lambda ab.\ H\vec{x}ab)\,\underline{k})$
  $= (\lambda ab.\ H\vec{x}ab)\,\underline{k}\,(F\,\vec{x}\,\underline{k})$
  $= H\vec{x}\,\underline{k}\,(F\,x\,\underline{k})$

- Il ne reste plus qu'à construire R !

Regardons le terme

$$u = \lambda xyz.\ z\,(\lambda d.v)\,x.$$

Il correspond à $u\,u\,v\,\underline{0} = \underline{0}\,(\lambda d.v)\,u = u$

$$u\,u\,v\,\underline{k} = \underline{k}\,(\lambda d.v)\,u = (\lambda d.v)^k\,u = v.$$

Alors R est le point fixe de

$$\lambda c\,xyz.\ u\,(x)\,(y\,(Pz)\,(c\,x\,y\,(Pz)))\,z$$

d'où $R = Y\,(\lambda c\,xyz.\ u\,x\,(y\,(Pz)\,(c\,x\,y\,(Pz)))\,z)$

MINIMISATION DE REQUÊTES TABLEAUX

Leçon 932

Références Abiteboul, Foundations of databases p.115
Handbook of Computer Science, vol B p.1091

Théorème

- Équivalence entre les requêtes projection/join et les tableaux.
- (Homomorphisme) Si $q=(T,u)$ et $q'=(T',u')$ sont des tableaux sur le schéma R, alors $q \subseteq q'$ si et seulement s'il existe un homomorphisme de q à q'.
- Si $q=(T,u)$ est un tableau, il existe un sous-ensemble T' de T tel que $q'=(T',u)$ est un tableau minimal et $q' \equiv q$.

I] • Preuve par induction :

- si $E=R$ alors T est le tableau contenant un R-tuple de variables distinguées.
- si $E=\pi_X(E_1)$, et T_1 un tableau de E_1.
  → on change les variables distinguées de T_1 qui correspondent à un attribut $A \notin X$ par un nouveau symbole non distingué.
  → on enlève les variables de u qui ne correspondent pas à X.
- si $E=E_1 \bowtie E_2$, et T_1 et T_2 sont les tableaux pour E_1, E_2.
  → T est l'union des ensembles de tuples de T_1, T_2.

Rappel : $\Theta : q' \to q$ est un homomorphisme ssi $\Theta(T') \subseteq T$ et $\Theta(u')=u$.

II] • Pour deux requêtes q, q' sur R, on écrit

$$q_1 \subseteq q_2 \iff \forall I \text{ instance sur } R,\ q_1(I) \subseteq q_2(I)$$

⇐ Supposons qu'il existe un homomorphisme $\Theta : q' \to q$.
Soit $w \in q(I)$, alors il existe une valuation v qui réalise T dans I telle que $v(u)=w$.

Alors comme $\theta(T') \subseteq T$ et $\theta(u') = u$, on peut appliquer θ à $\theta(u')$, et on obtient

$$v \circ \theta(u') = v(u) = w$$

Donc $w \in q(I')$

$\boxed{\Rightarrow}$ Supposons maintenant que $q \subseteq q'$: $(T,u) \subseteq (T',u')$

Soit V l'ensemble des variables de T.

Pour tout $x \in V$, posons a_x une nouvelle constante n'apparaissant pas dans T, T', et posons $\mu(x) = a_x$.

On définit alors $I_T = \mu(T)$.

Dès lors, remarquons que $\mu : V \to \mu(V)$ est bijective, et on peut définir μ^{-1} sur toutes les constantes de I_T car on a évité les collisions entre les a_x et les constantes de T.

Remarquons également que $\mu(u) \in q(I_T)$

on par hypothèse $q(I_T) \subseteq q'(I_T)$ donc $\mu(u) \in q'(I_T)$

Il existe donc une valuation de u' qui réalise T' dans I_T telle que $V(u') = \mu(u)$.

Dès lors, $\mu^{-1} \circ v$ envoie bien q' dans q : c'est l'homomorphisme recherché.

III] • Soit (S,v) un tableau minimal équivalent à q.

Il existe des homomorphismes $\theta : q \to (S,v)$

$\lambda : (S,v) \to q$

Posons alors $T' = \dots \circ (S)$

On a bien $(T', U) \equiv q$ par homomorphismes, et ses lignes sont dans q !

et $(T', U) \subseteq (S, V)$ donc $|T'| \leq |S|$

→ par minimalité de S, on a bien prouvé que T' est minimal.

THÉORÈME DE RICE

Leçons : 914

Références : LFCC, Carton p. 160-162.
Introduction à la calculabilité, Wolper.

Théorème

Toute propriété non triviale des langages récursivement énumérables est indécidable

(= accepté par une machine de Turing)

Résumé

Définissons le langage ("Π accepte w")

$$L_\epsilon = \{ \langle \Pi, w \rangle \mid w \in L(\Pi) \}$$

I - Soit P une propriété, le langage

$$L_P = \{ \langle \Pi \rangle \mid L(\Pi) \text{ satisfait } P \}$$

se réduit au langage L_ϵ.

II - L_ϵ est indécidable

I) • Quitte à remplacer P par $\overline{P}$, on peut supposer que le langage vide ne vérifie pas P.

• Comme P n'est pas triviale, il existe une machine de Turing N telle que $L(N)$ vérifie P.

• Posons M une machine de Turing et w un mot, définissons une machine $\Pi' = f(\Pi, w)$ telle que :

Soit x l'entrée.

Si M accepte w :

. Simuler N sur x et retourner le résultat.

Sinon rejeter

• On associe ainsi $\langle M' \rangle$ à $\langle M, w \rangle$, telle que :

- si M accepte w, alors $L(M') = L(N)$

ie. $L(M')$ vérifie P

- si M rejette w, alors $L(M') = \emptyset$

ie. $L(M')$ ne vérifie pas P.

• En somme : M accepte w si et seulement si $L(M')$ vérifie P.

• Donc on a bien réduit L_ϵ à L_P, $\langle M' \rangle$ étant calculable.

II] • On a $L_\epsilon \leq_m L_P$, mais ça serait bien qu'il soit indécidable pour conclure...

• Pour rappel, $L_\epsilon = \{ \langle M, w \rangle \mid w \in L(M) \}$.

• Supposons par l'absurde que L_ϵ est décidable : il existe alors une machine de Turing A telle que $L_\epsilon = L(A)$ qui s'arrête sur toute les entrées.

• Construisons une machine B qui encode A en dure telle que :

Pour une entrée $\langle M \rangle$,

Si A accepte $\langle M, \langle M \rangle \rangle$ alors rejeter

Sinon accepter

• Lançons B sur $\langle B \rangle$:

→ Si B accepte $\langle B \rangle$, alors A accepte $\langle B, \langle B \rangle \rangle$ par définition de A. mais alors B rejette $\langle B \rangle$ ...

→ Si B rejette $\langle B \rangle$, alors A rejette $\langle B, \langle B \rangle \rangle$ par définition de A. mais alors B accepte $\langle B \rangle$ ...

• Dans les deux cas il y a une contradiction, donc L_ϵ est indécidable

DONC L_P est indécidable.

Remarque il s'agit bien ici d'une propriété sur le langage d'une machine, et pas sur les machines elles mêmes,

Par exemple :

$L(k) = \{ \langle \Pi \rangle \mid \Pi$ s'arrête en au plus k étapes$\}$ pour $k \in \mathbb{N}^*$

est décidable, car il suffit de compter k étapes et de regarder si Π s'est arrêtée ou non.

Théorème de Savitch

Leçons 913, 915

Références : LFCC, Carton p. 219.

Théorème

Soit $s : \mathbb{N} \to \mathbb{R}^+$ telle que $s(n) \geq n$ pour n assez grand. Toute machine de Turing non déterministe qui fonctionne en espace $s(n)$ est équivalente à une machine de Turing déterministe en espace $O(s(n)^2)$.

Résumé

I - Se ramener à une machine avec un unique état final.

II - Définition d'une fonction ACCESS(C, C', t, r), vraie s'il existe un calcul entre deux configurations C et C' de longueur au plus t, n'utilisant que des configurations intermédiaires de longueur au plus r.

III - Complexité de ACCESS

i - si $s(n)$ est calculable

ii - si $s(n)$ n'est pas calculable

IV - Utilisation de ACCESS pour conclure.

I] • Si M fonctionne en espace $s(n)$,

On peut modifier M pour que, avant d'accepter, elle remplace tous les symboles de la bande par # et se place dans un état q_f.

→ ainsi, il n'y a qu'une configuration acceptante.

II] • On définit la fonction ACCESS par récurrence, avec le cas de base :

si $t=0$: est-ce que $C = C'$?

si $t=1$: est-ce que $C = C'$ ou $C \to C'$ en une étape de calcul ?

• Sinon, on parcourt toutes les configurations C'' en vérifiant s'il existe un calcul $C \to C'$ avec C'' en position médiane.

```
ACCESS(C, C', t, r)
    si t = 0 : renvoyer C = C'
    sinon si t = 1 : renvoyer C = C' ou C → C'.
    sinon
        pour tout C'' de taille r :
            si ACCESS(C, C'', ⌈t/2⌉, r) et ACCESS(C'', C', ⌊t/2⌋, r)
                renvoyer VRAI
        renvoyer FAUX
```

III. • Remarque

→ il y a au plus $\log_2 t$ appels récursifs imbriqués car t est divisé par 2 à chaque fois

→ chaque appel récursif utilise : trois variables C, C' et C'', chacune de taille au plus r.
• un entier t

→ r ne change pas, il suffit de le stocker une fois

Conclusion : il faut un espace

$$O(\log r + (\log t + r)\log t)$$

- $\log r$: stocker r en binaire
- $\log t$: stocker t en binaire
- r : stocker C, C', C''
- $\log t$: nombre d'appels imbriqués
- $(\log t + r)$: pour chaque appel imbriqué

i Supposons que $s(n)$ est calculable.

- M fonctionne en espace $s(n)$, et elle effectue donc un nombre d'étapes $t_M(n) \leq 2^{ks(n)}$ pour une certaine constante k.
- Un mot w de taille n est donc accepté par M si et seulement si $\text{ACCESS}(q_0 w, q_f, 2^{ks(n)}, s(n))$

car il n'y a qu'une seule configuration finale

ie $w \in L(M) \iff \text{ACCESS}(q_0 w, q_f, 2^{ks(n)}, s(n))$

- On peut alors définir une machine de Turing déterministe qui
  - calcule $s(n)$
  - utilise ACCESS avec $t = 2^{ks(n)}$ et $r = s(n)$.
- L'espace utilisé est alors

$$O(k^2 s(n)^2) = O(s(n)^2).$$

car $\log r = O(\log(s(n))) = O(s(n))$

$\log t = O(\log(2^{ks(n)})) = O(ks(n)) = O(s(n))$

ii - Si $s(n)$ n'est pas calculable, il va falloir ruser pour trouver un majorant de l'espace nécessaire.

- Posons n : la taille de l'entrée w

  m : la taille maximale d'une configuration accessible depuis $q_0 w$.
- Calculons m en espace $O(m^2) = O(s(n)^2)$, avec la propriété suivante : pour $k \geq n+1$
  - soit N_k = le nombre de configurations de taille au plus k accessibles à partir de $q_0 w$.
  - $(N_k)_{k \geq n+1}$ est croissante !
  - et elle est bornée par le nombre de configurations de taille au plus $s(n)$.

• Donc $(N_k)_{k \geq n+1}$ converge, et on a :

$$m = \min \{ k \geq n+1 \mid N_k = N_{k+1} \}$$

Comme la taille d'une configuration ne varie que d'un à chaque étape du calcul, soit $N_{k+1} > N_k$ soit $N_{k'} = N_k$ pour $k' \geq k$.
→ m est donc aussi bien défini.

• On en déduit l'algorithme :

CALCUL $m(w)$ où $|w| = n$

```
N := -1,   Γ := ∅,   k := 0
Tant que N ≠ Γ
    k := k+1,   N := Γ,   Γ := ∅
    Pour tout C de taille au plus k :
        Si ACCESS(q0w, C, 2^{K·k}, k) : Γ := Γ + 1
```

cette constante peut être prise comme étant égale à $\log_2(1+|Q|+|\Gamma|)$

• On a toujours $k \leq m$ donc les appels à ACCESS sont en $O(m^2)$, et toutes les variables se stockent en $O(m)$.
→ CALCUL m tourne donc en $O(m^3) = O(s(n)^2)$.

IV • Il suffit finalement de définir Γ' qui calcule m avec CALCUL m, puis qui utilise la fonction ACCESS$(q_0 w, q_f, 2^{Km}, m)$.
→ elle donne le résultat en $O(s(n)^2)$.

Simulation d'un AFD à partir d'une expression régulière.

Leçons

Références : Dragon Book, Aho

Théorème

Soit E une expression rationnelle.
Après un prétraitement sur l'arbre syntaxique de E, on peut simuler l'AFD parcouru pour la lecture d'un mot w en temps $O(|w| \cdot |E|^2)$.

Remarque : une preuve montrant la possibilité de construire un AFD reconnaissant une expression rationnelle est de construire par récurrence un AFND la reconnaissant, puis de le déterminiser en construisant l'automate des parties.
La taille de l'automate obtenu est alors exponentielle, ce qui induit un algorithme de conversion s'exécutant en temps exponentiel.

Problème : cette borne est optimale, il suffit de regarder l'expression $(a|b)^* a (a|b)^{n-1}$. L'automate doit alors garder en mémoire les n dernières lettres, ce qui induit un nombre exponentiel d'états à garder en mémoire.
L'automate $A = (Q, q_0, F, \delta)$ est en effet minimal, avec :

$\Sigma = (a|b)^n$, $q_0 = b^n$, $F = a(a|b)^{n-1}$ et

$\delta(xw, y) = wy$ pour $w \in (a|b)^{n-1}$ et $x, y \in \{a, b\}$.

Et il a 2^n état !

Solution On peut éviter de générer l'automate complet en simulant uniquement le chemin emprunté, quitte à le garder en mémoire pour après.

Résumé

I - Construction d'un arbre syntaxique pour r#.

II - Calcul des fonctions suivantes :

eps (n) : l'expression enracinée en n reconnaît ε.

début (n) : feuilles ~~lettres~~ pouvant être au début d'un mot

fin (n) : feuilles ~~lettres~~ pouvant être à la fin.

suivant (n) : les positions pouvant correspondre à une lettre qui peut apparaître à la suite d'un mot reconnu par l'expression enracinée en n. l'ensemble des feuilles de l'arbre correspondant aux possibilités pour la lettre qui suit.

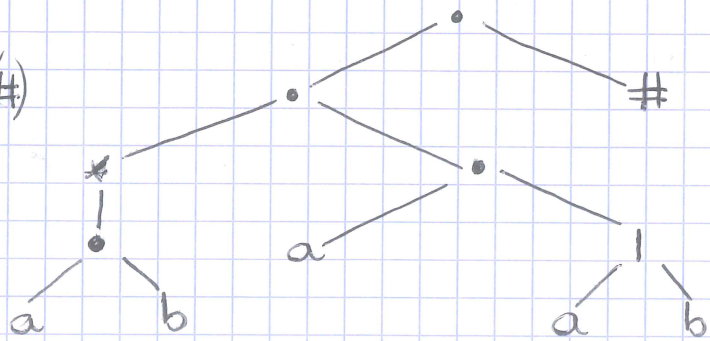
III - Construction, simulation et complexité de l'AFD.

I • Une expression rationnelle c'est :

* une lettre : a — une feuille
* $(r)^*$ — un noeud * avec un enfant r
* $(r_1 \mid r_2)$ — un noeud | avec deux enfants r_1 et r_2.
* $r_1 r_2$ — un noeud • avec deux enfants r_1 et r_2.

Cela définit bien un arbre, par induction structurelle sur les expressions rationnelles.

$((ab)^*(a(a \mid b))\#)$



On rajoute $\# \notin \Sigma$ à la fin pour savoir facilement en tête le moment où l'on arrive au bout d'un mot.

II | • On associe un numéro à chaque noeud de l'arbre généré, que l'on appellera position de ce noeud.

• Les fonctions eps, début, fin peuvent alors se calculer avec les règles suivantes :

Noeud n	eps(n)	début(n)	fin(n)
feuille ε	T	$\emptyset$	$\emptyset$
feuille $a \neq \varepsilon$ en position i	F	$\{i\}$	$\{i\}$
disjonction $r_1 \mid r_2$	$eps(r_1) \vee eps(r_2)$	$début(r_1) \cup début(r_2)$	$fin(r_1) \cup fin(r_2)$
concaténation $r_1 \cdot r_2$	$eps(r_1) \wedge eps(r_2)$	si $eps(r_1)$: $début(r_1) \cup début(r_2)$ sinon : $début(r_1)$	si $eps(r_2)$ $fin(r_1) \cup fin(r_2)$ sinon $fin(r_2)$
étoile r^*	T	début(r)	fin(r)

• La fonction suivant se construit alors facilement, car deux lettres consécutives d'un mot ne peuvent être obtenues que

- par concaténation : en avançant
- par une étoile : en reculant.

On obtient donc naturellement la fonction :

$$suivant(i) = \bigcup_{\substack{(\cdot, T_1, T_2) \\ i \in fin(T_1)}} début(T_2) \bigcup_{\substack{(*, T) \\ i \in début(T)}} fin(T)$$

III] • Il ne reste plus qu'à définir un AFD reconnaissant $r\#$ et une façon de calculer sa fonction de transition à la volée.

Cet automate est défini par :

* $Q = \mathcal{P}(\llbracket 1, n \rrbracket)$, où chaque nombre représente un des n nœuds de l'arbre syntaxique de $r\#$.
* $q_0 = \text{début}(n_0)$ où n_0 est la racine de T.
* $F = \{ q \in Q \mid n_\# \in q \}$ où $n_\#$ est la position de la feuille dont le symbole est $\#$.
* $\delta : Q \times \Sigma \longrightarrow Q$ tel que :

$$\delta(q, a) = \bigcup_{\substack{i \in q \\ \text{symbole}(i) = a}} \text{suivant}(i)$$

Remarque Cet automate, c'est vraiment un automate déterminisé, qui part de l'ensemble des états pouvant être initiaux, puis explore toutes les positions accessibles en lisant un mot.

Complexité : soit n la longueur de l'expression rationnelle r.

* prétraitement : calculs de eps, début, fin et suivant :
  $O(n)$ pour les trois premières
  $O(n^2)$ pour la dernière,

* parcours : $\delta(q, a)$ peut se calculer en $O(n^2)$ car $|q| = O(n)$ et $|\text{suivant}(i)| = O(n)$

$\Rightarrow$ on peut donc reconnaître $w \in \Sigma^*$ en $O(|w| \cdot n^2)$.

Complexité moyenne du tri rapide

Leçons 903, 926, 931

Références Eléments d'algorithmique, Beauquier p. 126.

Théorème

La complexité en moyenne du tri rapide sur la distribution uniforme des permutations de L est $O(n \log n)$

Résumé

I - Description de l'algorithme de tri rapide

II - La procédure de pivotage se fait en $n+1$ comparaisons.

III - Après pivotage, les permutations à gauche (/à droite) sont équiprobables.

IV - Analyse de la complexité moyenne

I] • L'algorithme classe récursivement et en place les éléments en deux listes: les éléments plus petits et ceux plus grands qu'un pivot. Pour un tableau $T[i..j]$, on définit récursivement l'algorithme de tri rapide:

```
TRI_RAPIDE ( T[i...j])
    Si i < j
        k = PIVOTER(T[i..j])
        TRI_RAPIDE(T[i..(k-1)])
        TRI_RAPIDE(T[(k+1)..j])
```

• La procédure PIVOTER est là où le tri s'effectue, et il repose sur une propriété des couples inversés par rapport au pivot γ, c'est-à-dire deux indices s et t tels que $i \leqslant s < t \leqslant j$

et $T[s] > \gamma \geqslant T[t]$

On peut donc trouver un couple inversé dans $T[i,j]$ avec :

```
COUPLE_INVERSÉ(T[i..j], γ)
    s = i , t = j
    Tant que s ≤ j et T(s) ≤ γ : s = s+1
    Tant que t ≥ i et T(t) > γ : t = t-1
    Renvoyer (s,t)
```

Un couple renvoyé par cet algorithme est inversé, et on est sûr qu'il n'y en a eu aucun avant (pour $s' \leqslant s$ et $t' \geqslant t$ (mais pas $s'=s$ et $t'=t$))

• La procédure de PIVOT se définit alors ainsi sur $T[i..j]$

- prendre $\gamma := T[i]$ comme pivot
- chercher un premier couple inversé (s,t) avec PREMIER_COUPLE
- inverser les deux et recommencer sur $T[s+1..t-1]$, jusqu'à ce qu'il soit réduit à un élément ou à l'ensemble vide.
- mettre le pivot à sa place en le mettant en position $t-1$.

γ 6	14	3	1	8	4	5	9
γ 6	5	3	1	8	4	14	9
4	5	3	1	γ 6	8	14	9

II • Le PIVOT se fait en moins de $n+1$ comparaisons :

$t-s$ vaut au plus n, et est décrémenté chaque fois qu'une comparaison est faite. Comme on s'arrête quand il est $\leqslant 0$, on a au plus $n+1$ comparaisons.

• Supposons que la permutation initiale de T est choisie de façon uniforme parmi les listes

III • Montrons qu'après pivotage, les distributions à gauche et à droite du pivot sont uniformes

• Soit r le rang du pivot, on associe des permutations de $[\![1 \dots r-1]\!]$ et $[\![r+1 \dots n]\!]$ aux listes gauche et droite, notées π_1 et π_2.

• supposons que $1 < r < n$, car sinon PIVOT n'a juste rien changé.

• On pose $\pi = \pi_1 \cdot (r) \cdot \pi_2$, et on va regarder combien de permutations initiales de $[\![1 \dots n]\!]$ mènent à π après PIVOT.

→ supposons que l'algorithme a échangé q couples

$(s_1, t_1), (s_2, t_2), \dots, (s_q, t_q)$

où $1 \leqslant s_1 < s_2 < \dots < s_q \leqslant r-1$ et $r+1 \leqslant t_1 < t_2 < \dots < t_q \leqslant n$. (*)

→ posons τ_0 la transposition qui échange $\pi(1)$ et $\pi(r)$

et τ_k la transposition qui échange $\pi(s_k)$ et $\pi(t_k)$

alors $\pi \, \tau_q \circ \tau_{q-1} \circ \dots \circ \tau_0 \circ \pi$ déroule l'algorithme à l'envers et redonne la permutation initiale du tableau.

→ on peut donc appliquer ça à π : on choisit $s_1 \dots s_q$ et $t_1 \dots t_q$ vérifiant (*), alors l'algorithme de pivot sur $\pi' = \tau_q \circ \dots \circ \tau_0 \circ \pi$ redonnera π.

+ la fonction $s_1, \dots, s_q, t_1, \dots, t_q \to \pi'$ est injective

⇒ le nombre de π' qui convient est donc déterminé par :

le choix de q puis des s_k et t_k ; il y en a donc :

$\left.\begin{array}{l} q \leqslant r-1 \\ \text{et } q \leqslant n-r \end{array}\right\}$

$$\sum_{q=1}^{\min(r-1,\, n-r)} \binom{r-1}{q} \binom{n-r}{q}$$

choix des s_k — choix des t_k

Cela ne dépend pas de la permutation π, on a donc autant de possibilités pour chacune des permutation π (donc pour π_1 et π_2 aussi !)

IV | • L'hypothèse d'équiprobabilité est conservée donc, en posant $M(n)$ le nombre de comparaisons pour une liste de taille n, on a :

$$M(n) \leqslant \sum_{r=1}^{n} \frac{1}{n}\left[n+1 + M(r-1) + M(n-r) \right]$$

PIVOT — tris récursifs

Or $\sum_{r=1}^{n} M(r-1) = \sum_{r=1}^{n} M(n-r) = \sum_{r=1}^{n-1} M(r)$

D'où $M(n) \leqslant n+1 + \frac{2}{n} \sum_{r=1}^{n} M(r)$

et $M(n) = O\left(n \sum_{k=1}^{n+1} \frac{1}{k}\right)$

$= O(n \log n)$.

TRI PAR TAS

Leçons : 901, 903, 927, 931

Références : Cormen

Théorème

Le tri par tas trie un tableau en place en $O(n \log n)$

Résumé

I - Définition d'une procédure ENTASSER_MAX permettant de placer correctement un élément du tas si le reste qui le suit est correct.

II - Construction d'un tas max à partir d'un tableau.

III - Algorithme du tri par tas

Remarque : un tas est représenté sous la forme d'un tableau, mais il peut être vu comme un arbre binaire presque complet, avec les relations

GAUCHE(i) = 2i

DROITE(i) = 2i+1

PARENT(i) = $\lfloor i/2 \rfloor$.

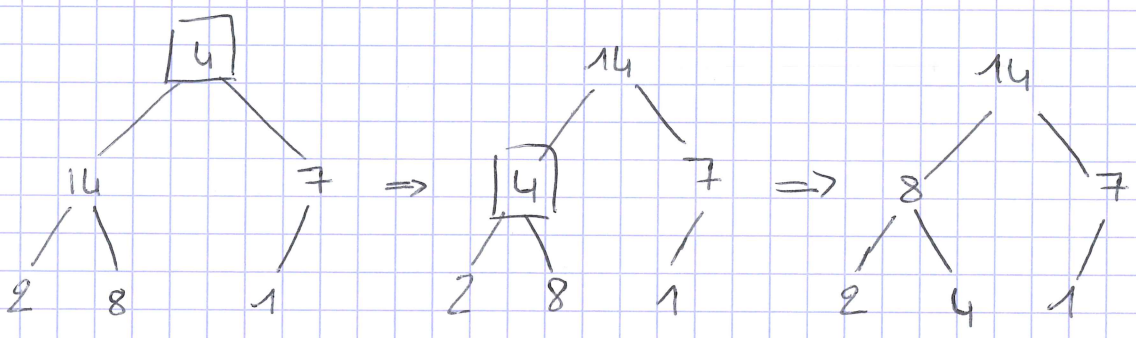
+ un tas max a la propriété, pour i différent de la racine :

A[PARENT[i]] ≥ A[i].

I - On suppose que les tas enracinés en GAUCHE[i] et DROITE[i] sont max. La procédure suivante fait descendre A[i] autant qu'il faut :

```
ENTASSER_MAX(A,i):
    max = argmax(A[i], A[GAUCHE[i]], A[DROITE[i]])
    si max ≠ i
        ECHANGER A[i] et A[max]
        ENTASSER_MAX(A, max).
```

Exemple :



À chaque étape, le tas non modifié est max et son parent est plus grand, et on continue de même récursivement jusqu'à ce que ce soit bon.

Complexité : à chaque appel récursif de ENTASSER_MAX, la hauteur de l'arbre considéré, h, diminue de 1. L'algorithme s'exécute donc en $O(h)$

Comme l'arbre est presque complet, $O(h) = O(\log n)$.

II • En partant d'un tableau A de taille n, on peut construire en place un tas max en entassant à partir de l'étage juste au dessus des feuilles.

• Entre $\lfloor n/2 \rfloor + 1$ et n, l'arbre binaire ne contient que des feuilles, on peut donc commencer à entasser en $\lfloor n/2 \rfloor$:

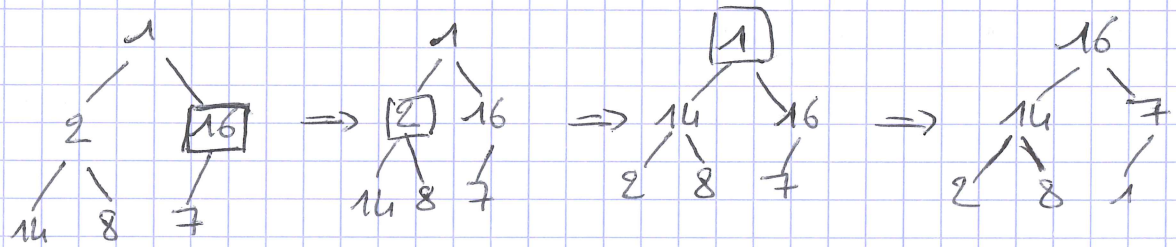
```
CONSTRUIRE_TAS_MAX
    n = taille[A]
    Pour i = ⌊n/2⌋ à 1,
        ENTASSER_MAX(A,i)
```

Correction On utilise l'invariant de boucle suivant :

"Au début de chaque itération, les noeuds i+1, i+2, ..., n sont racines d'un tas max."

Initialisation : pour $i = \lfloor n/2 \rfloor$, les noeuds i+1 à n sont des feuilles, donc des racines de tas max à un élément.

Conservation : les indices des enfants de i sont strictement plus grands que lui : ce sont donc des racines de tas max.
L'appel d'ENTASSER_MAX construit donc un tas max enraciné en i, et les noeuds d'indices plus grands restent des racines de tas max.



Complexité : la complexité d'ENTASSER_MAX sur un tas de hauteur h est $O(h)$, et il y a au plus $\lceil n/2^{h+1} \rceil$ noeuds à hauteur h.
Le coût en temps est donc

$$\sum_{h=1}^{\lfloor \log_2 n \rfloor} \lceil n/2^{h+1} \rceil O(h) = O\left(n \sum_{h=1}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right) = O(n) \quad \text{car} \sum_{h=1}^{+\infty} \frac{h}{2^h} = 2.$$

III • On a presque l'algorithme de tri :
- on construit déjà un tas max dans le tableau A.
- on peut échanger $A[1]$ (le plus grand élément) et $A[n]$ (le dernier), dès lors A privé de son dernier élément est un tas max à l'exception de $A[1]$, qu'il suffit d'entasser, en ignorant le dernier élément du tableau :

```
TRI_PAR_TAS(A)
    CONSTRUIRE_TAS_MAX
    Pour i = longueur(A) jusqu'à 2 :
        Echanger A[1] ↔ A[i]
        Taille(A) = Taille(A) - 1.
        ENTASSER_MAX(A, 1)   (on ignore dans cet appel la fin de A)
```

Correction découle directement de celles de ENTASSER_MAX et de CONSTRUIRE_TAS_MAX : on a un tas max donc T[1] est toujours le plus grand élément → et c'est celui qu'on met à la fin !

Complexité

- Construction du tas max : $O(n)$
- $O(\log n)$ pour chaque appel à ENTASSER_MAX

→ d'où une complexité en temps en $O(n \log n)$.

VOYAGEUR DE COMMERCES EUCLIDIEN

Leçons 925, 928.

Références Cormen p. 978

Théorème

Soit G un graphe complet à $n \in \mathbb{N}$ sommets

w une fonction de poids sur les arêtes de G.

k un entier positif.

(VC) Existe-t-il un chemin visitant chaque sommet exactement une fois, finissant sur le sommet de départ, dont la somme des poids des arêtes empruntées est au plus k.

I - (VC) est NP-complet

II - Si w vérifie l'inégalité triangulaire, il existe un algorithme d'approximation avec une garantie de performance 2 à temps polynomial pour le problème (VC)

III - Si $P \neq NP$ et sans l'inégalité triangulaire, il n'existe aucun algorithme d'approximation polynomial et à garantie de performance $\rho \geq 1$ pour (VC)

I • VC $\in$ NP : Etant donné un chemin, il suffit de vérifier

- qu'il passe par tous les sommets une fois et revient au départ
- que son poids est plus petit que k. (on l'appelle aussi coût)

Ce qui se fait en temps polynomial.

• VC est NP-difficile : montrons que CYCLE-HAM $\leq_p$ VC.

Soit $G=(S,A)$ une instance de CYCLE-HAM.

Définissons $G'=(S,A')$, où $A' = \{(i,j) : i,j \in S \text{ et } i \neq j\}$.

Définissons la fonction de poids w par :

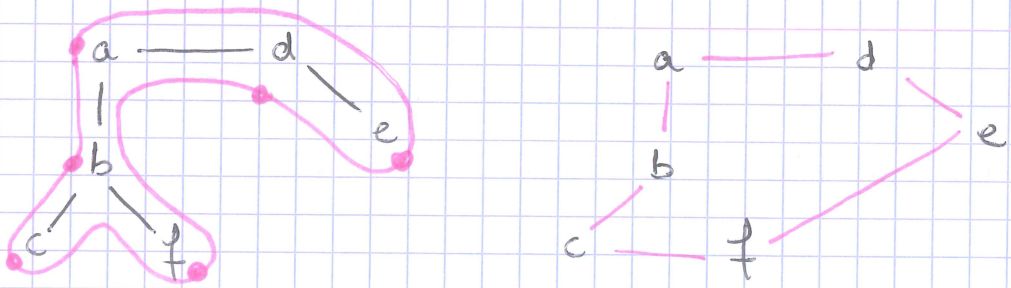
$$w(i,j) = \begin{cases} 0 & \text{si } (i,j) \in A \\ 1 & \text{si } (i,j) \notin A \end{cases}$$

- Trouver une solution à VC de poids au plus zéro revient alors à trouver un cycle hamiltonien :
  - si h est un cycle hamiltonien de G, h est une tournée de G' de poids nul
  - si h' est une tournée de G' de poids au plus zéro, chacune de ses arêtes ayant un poids nul, elle doit donc avoir un poids nul. Donc h' ne contient que des arêtes de A, c'est donc un cycle hamiltonien de G.

$\Rightarrow$ CYCLE-HAM $\leq_p$ VC donc VC est NP-complet.

II) • Définissons l'algorithme suivant pour $G=(S,A)$:
- sélectionner une racine $r \in S$
- calculer un arbre couvrant minimal T pour G depuis r.
- le parcours préfixe de T définit un chemin hamiltonien H

→ renvoyer H.



→ le parcours préfixe visite exactement une fois chaque sommet, en l'ignorant lors de sa seconde rencontre.

+ l'algorithme s'exécute en temps polynomial car on peut construire l'arbre couvrant minimal en temps polynomial.

- On définit ainsi une 2-approximation polynomiale.

→ en enlevant une arête de une tournée H^* optimale, on obtient un arbre couvrant, les poids $C(T)$ et $C(H^*)$ vérifient donc

$$C(T) \leq C(H^*).$$

→ le parcours complet W associé au parcours préfixe de T passe exactement deux fois par chaque sommet, d'où

$$C(W) \leq 2C(T)$$

d'où $C(W) \leq 2C(H^*)$.

→ de plus, si l'on supprime la deuxième occurence des sommets dans W, on peut relier ceux qui sont autour (G est complet) et le poids de cette arête est inférieur à celui des arêtes enlevées (par inégalité triangulaire), on a donc :

$$C(H) \leq 2C(H^*).$$

III • Supposons qu'il existe un algorithme B polynomial qui renvoit une ρ-approximation de VC.

On peut arrondir ρ et donc le supposer entier.

• B permet de résoudre CYCLE-HAM en temps polynomial.

- soit $G=(S,A)$ une instance de CYCLE-HAM.

posons $G'=(S,A')$ où $A' = \{(i,j) : i,j \in S \ i \neq j\}$

$$w(i,j) = \begin{cases} 1 & \text{si } (i,j) \in A \\ \rho|S|+1 & \text{si } (i,j) \notin A. \end{cases}$$

G' peut être construit en temps polynomial à partir de G.

- si G a un cycle hamiltonien, (G',w) a une tournée de coût $|S|$.
- sinon, une tournée de (G',c) coûte au moins $\rho|S|+|S|$ car on emprunte au moins une arête de coût $\rho|S|+1$.

- comme B est une ρ-approximation;
  - → si G a un cycle hamiltonien, B renvoit ce cycle car c'est le seul à avoir un coût inférieur à $\rho|S|$.
  - → sinon il renvoit un cycle de coût supérieur strictement à $\rho|S|$.

- On peut donc résoudre CYCLE-HAM en temps polynomial, ce qui contredit $P \neq NP$.

ALGORITHME DE TYPE PRINCIPAL

Leçons 923, 929

Référence lectures on the Curry-Howard Isomorphism, Sørensen p. 95.

Théorème

Soit $\mathcal{B}$ un ensemble dénombrable de variable de type.

Les types sont définis inductivement par $T ::= \mathcal{B} \mid T \to T$.

On définit ensuite la relation de typage par les règles

$$\frac{\sigma \in \mathcal{B}}{\Gamma, x:\sigma \vdash x:\sigma}\ \text{variable} \qquad \frac{\Gamma \vdash t_1 : \sigma' \to \sigma \quad \Gamma \vdash t_2 : \sigma'}{\Gamma \vdash t_1 t_2 : \sigma}\ \text{application de fonction} \qquad \frac{\Gamma, x:\sigma' \vdash t : \sigma}{\Gamma \vdash \lambda x.t : \sigma' \to \sigma}\ \text{définition de fonction.}$$

Alors pour un contexte Γ, un type σ et un terme t, on a équivalence entre :

- $\Gamma \vdash t : \sigma$
- il existe un unificateur d'un ensemble E_t d'équations sur les variables τ_t où t' est un λ-terme tel que $\sigma = \mu(\tau_t)$ et $\Gamma(x) = \mu(\tau_x)$ pour $x \in FV(t)$.

Remarque : c'est intéressant de résoudre ce problème d'unification car alors on a des types pour les λ-termes concernés, ce qui implique qu'ils soient fortement normalisants !

Résumé

I - Définition des égalités E_t et des variables "de type"

II - Preuve du théorème par induction.

I| • On définit E_t et τ_t par induction :

* si $t = x$ $\quad E_x = \{\}$

$\tau_x = \alpha_x$ une nouvelle variable.

* si $t = t_1 t_2$ $\quad E_{t_1 t_2} = E_{t_1} \cup E_{t_2} \cup \{\tau_{t_1} = \tau_{t_2} \to \tau_{t_1 t_2}\}$

$\tau_{t_1 t_2} = \alpha$ une nouvelle variable.

* si $t = \lambda x.t_1$ $\quad E_{\lambda x.t_1} = E_{t_1}$

$\tau_{\lambda x.t_1} = \tau_x \to \tau_{t_1} (= \alpha_x \to \tau_{t_1})$

II| • Prouvons maintenant l'équivalence par induction.

* si $t = x$

$\Rightarrow$ On a $\Gamma \vdash x : \sigma$

donc $u = \alpha_x \to \sigma$ unifie $E_x = \{\}$

vérifie $u(\tau_x) = u(\alpha_x) = \sigma$.

$\Leftarrow$ On a u qui unifie $E_x = \{\}$

et $u(\tau_x) = u(\alpha_x) \in B$

Et on a $x : u(\alpha_x) \vdash x : u(\alpha_x)$

* si $t = t_1 t_2$

$\Rightarrow$ On a $\Gamma \vdash t_1 t_2 : \sigma$

D'où $\Gamma \vdash t_1 : \sigma' \to \sigma$

$\Gamma \vdash t_2 : \sigma'$

On a donc deux unificateurs u_1, u_2 qui unifient E_{t_1} et E_{t_2}

avec $\Gamma(x) = u_1(\alpha_x)$ pour $x \in fV(t_1)$

$\Gamma(x) = u_2(\alpha_x)$ pour $x \in fV(t_2)$.

Posons u tel que pour $x \in FV(t_1 t_2)$:

* $u(\alpha_x) = \begin{cases} u_1(\alpha_x) & \text{si } x \in FV(t_1) \\ u_2(\alpha_x) & \text{si } x \in FV(t_2) \end{cases}$

qui est bien définie car si $x \in FV(t_1) \cap FV(t_2)$, $u_1(\alpha_x) = \Gamma(x) = u_2(\alpha_x)$.

* $u(\tau_{t_1}) = \sigma' \to \sigma$
* $u(\tau_{t_2}) = \sigma'$
* $u(\tau_{t_1 t_2}) = \sigma$

Dès lors u unifie $E_{t_1 t_2} = E_{t_1} \cup E_{t_2} \cup \{\tau_{t_1} = \tau_{t_2} \to \tau_{t_1 t_2}\}$.

et on a bien $u(\tau_{t_1 t_2}) = \sigma$

$u(\alpha_x) = \Gamma(x)$ pour $x \in FV(t_1 t_2)$

$\Leftarrow$ On a u qui unifie $E_{t_1 t_2} = E_{t_1} \cup E_{t_2} \cup \{\tau_{t_1} = \tau_{t_2} \to \alpha\}$.

En particulier, u unifie E_{t_1} et E_{t_2} ~~ce sont nos hypothèses, on a alors~~

$\Gamma_1 \vdash t_1 : u(\tau_{t_1})$ où $\Gamma_1(x) = u(\alpha_x)$ pour $x \in FV(t_1)$

$\Gamma_2 \vdash t_2 : u(\tau_{t_2})$ où $\Gamma_2(x) = u(\alpha_x)$ pour $x \in FV(t_2)$

Et u unifie $\{\tau_{t_1} = \tau_{t_2} \to \alpha\}$ donc $u(\tau_{t_1}) = u(\tau_{t_2}) \to u(\alpha)$

soit $\Gamma_1 \vdash t_1 : u(\tau_{t_2}) \to u(\alpha)$

On peut donc utiliser la règle d'application de fonction avec $\Gamma = \Gamma_1 \cup \Gamma_2$:

$$\frac{\Gamma \vdash t_1 : u(\tau_{t_2}) \to u(\alpha) \qquad \Gamma \vdash t_2 : u(\tau_{t_2})}{\Gamma \vdash t_1 t_2 : u(\alpha)}.$$

et on a évidemment $u\Gamma(x) = u(\alpha_x)$ pour $x \in FV(t_1 t_2)$

* si $t = \lambda x.\, t_1$

$\Rightarrow$ On a $\Gamma \vdash \lambda x.\, t_1 : \sigma' \to \sigma$.

D'où $\Gamma, x : \sigma' \vdash t_1 : \sigma$

Il existe alors u unifiant E_{t_1} tel que $u(\tau_{t_1}) = \sigma$

$u(\alpha_y) = \Gamma(y)$ pour $y \in FV(t_1)$ $(y \neq x)$

$u(\alpha_x) = \sigma'$

Ce qui donne $u(\tau_{\lambda x.\, t_1}) = u(\tau_x \to \tau_{t_1})$

$= u(\tau_x) \to u(\tau_{t_1})$

$= \sigma' \to \sigma$

$\Leftarrow$ On a u qui unifie $E_t = E_{t_1}$

Et il existe Γ tel que :

$\Gamma \vdash t_1 : u(\tau_{t_1})$

$\Gamma(y) = u(\alpha_y)$ pour tout $y \in FV(t_1)$.

Dès lors,

* soit $x \in dom\,\Gamma$, et on a

$$\frac{\Gamma \vdash t_1 : u(\tau_{t_1})}{\Gamma \vdash \lambda x.\, t_1 : \underbrace{u(\alpha_x) \to u(\tau_{t_1})}_{= u(\tau_{\lambda x.\, t_1})}}$$

* soit $x \notin dom\,\Gamma$, et on a, avec $\sigma = u(\alpha_x)$

$$\frac{\Gamma, x : \sigma \vdash t_1 : u(\tau_{t_1})}{\Gamma, x : \sigma \vdash \lambda x.\, t_1 : \sigma \to u(\tau_{t_1})}.$$