

Table des matières

I	Développements	3
1	Illustration du théorème de Cook	5
2	Correspondance entre les arbres binaires et les arbres généraux	8
3	Illustration du paradigme orienté objet sur la modélisation d'une personne sur une carte	12
4	Approximation(s) gloutonne(s) de Indep(2)	15
5	Présentation de l'algorithme des k-plus proches voisins	18
6	Validité de la construction d'un ensemble inductif	22
7	Preuve de l'équilibrage des arbres rouges noirs et méthode d'insertion	25
8	Généralisation du tri par comptage à l'aide de dictionnaires	28
9	Présentation et terminaison de l'algorithme de Bellman-Ford	31
10	Protocole HTTPS	34
11	Illustration des différents aspects de la méthode diviser pour régner sur le problème de la pyramide	39
12	Premiers pas avec SQL	42
13	Avec ou sans agrégation SQL	45
14	Problème du Rendez-vous	48
15	Algorithme A*	53
16	Jeu de Nim	56
17	Intérêts et insuffisances des critères de test	58
18	Equivalence entre l'impératif en C et le fonctionnel en Ocaml	61
19	Correction du tri fusion	65
20	Equivalence entre expression booléenne et fonction booléenne	67
21	Explication de la pile d'appel	70
22	Passage d'une expression rationnelle à un automate	73
23	Construction d'une base de données relationnelles d'élèves à l'université	76

24 Présentation d'un algorithme d'analyse syntaxique descendant par retour sur trace	78
25 Présentation des arbres k-dimensionnel	81
26 Correction de l'insertion dans un tas min	84
27 2-SAT est résoluble en temps linéaire	88
28 Implementation d'une file PAPS avec des piles	90
29 Zoom sur le protocole TCP	93
30 Indécidabilité de la terminaison et de la correction partielle	96
31 LZW	98
32 Vérification du produit de matrice	101
33 Algorithme de Peterson	103
34 Construction d'un additionneur à retenue anticipée	106
35 Distance d'édition	108
36 Automate des motifs	110
37 3-SAT est NP complet	113

Première partie

Développements

Développement 1

Illustration du théorème de Cook

Auteur·e·s: Emile Martinez

Références :

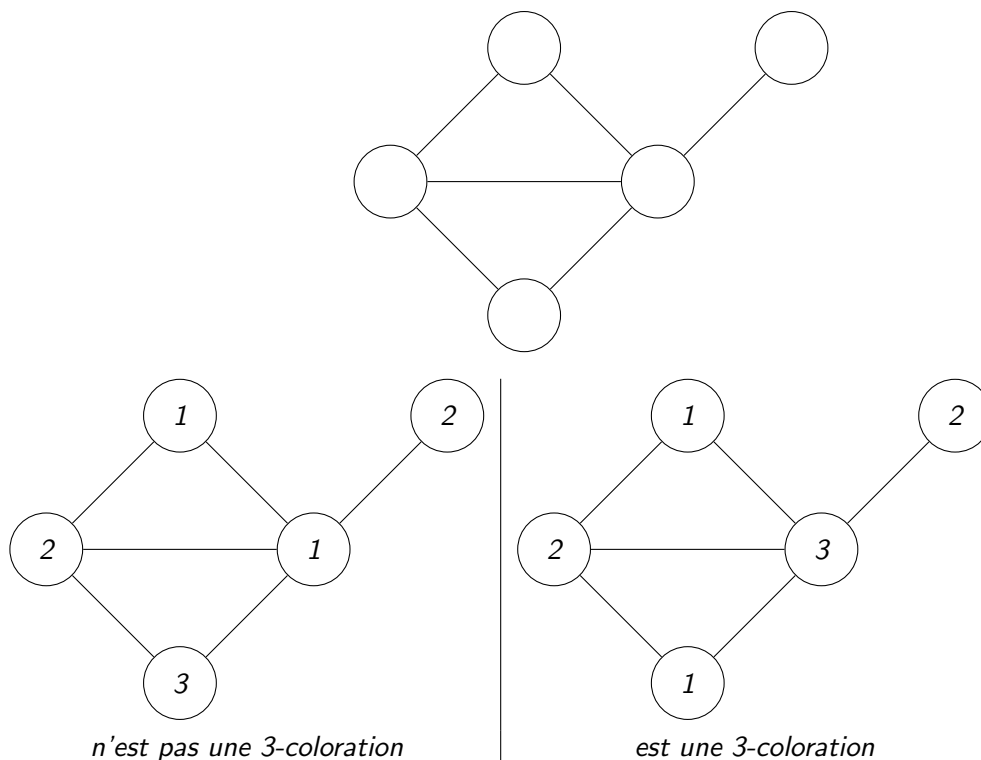
Illustration du théorème de Cook et de la puissance du problème SAT en présentant sur des exemples (3-coloration et Subset-Sum) comment se ramener au problème SAT. Il peut illustrer les leçons de NP-complétude ou les leçons parlant de formules propositionnelles.

Définition 1.1 Le problème **coloration** est un problème sur les graphes non orientés.
Une instance positive de coloration est un graphe $G = (S, A)$ et un entier k tel que

$$\exists c : S \rightarrow \llbracket 1, k \rrbracket : \forall (u, v) \in A, c(u) \neq c(v)$$

On dit alors que c est une k -coloration de G .

Exemple 1.1 Sur le graphe suivant :



Théorème 1.1 *Il existe une transformation polynomiale d'une instance de coloration vers une instance de SAT.*

Démonstration. Soit $G = (S, A)$ un graphe et $k \in \mathbb{N}^*$ une instance de coloration.

Créons les variables $x_{v,i}$ pour $v \in S$ et $i \in \llbracket 1, k \rrbracket$, dont on voudra donner la signification $x_{v,i}$ dira si v est à la couleur i .

Existence d'une couleur Créons pour $v \in S$, $E_v = \bigvee_{i=1}^k x_{v,i}$
Ainsi, si E_v est satisfaite, au moins un des $x_{v,i}$ est à vrai.

Unicité d'une couleur Créons pour $v \in S$, $U_v = \bigwedge_{i=1}^k \bigwedge_{j=1, j \neq i}^k \neg x_{v,i} \vee \neg x_{v,j}$
Ainsi, si U_v est satisfaite, on ne peut pas avoir deux $x_{v,i}$ différents qui sont satisfaits. On en a donc au plus un.

Coloration Créons $C = \bigwedge_{(u,v) \in A} \bigwedge_{i=1}^k \neg x_{u,i} \vee \neg x_{v,i}$

Si C est satisfaite, pour aucune couleur, deux voisins dans le graphe on leur variable à vrai. Dans notre interprétation, deux voisins ne peuvent pas avoir la même couleur.

On crée alors l'instance de SAT $I_{SAT} = \bigwedge_{v \in S} E_v \wedge \bigwedge_{v \in S} U_v \wedge C$

\Rightarrow Supposons que G admette une k -coloration c . Alors on choisit comme valuation $\sigma(x_{v,i}) = V \iff c(v) = i$. Alors, c étant une fonction, σ évalue bien à vrai E_v et U_v (car il y a un et un seul i tel que $c(v) = i$), et étant une coloration, σ évalue à vrai C car si $(u, v) \in A$, $\neg(c(u) = i \wedge c(v) = i)$ (car $c(u) \neq c(v)$)

\Leftarrow Supposons que σ évalue I_{SAT} à vrai. Alors, pour $v \in S$, U_v et E_v nous disent qu'un unique $x_{v,i}$ est à vrai.

$$\forall v \in S, \exists! i \in \llbracket 1, k \rrbracket : \sigma(x_{v,i}) = V$$

Notons $c(v)$ cet unique i . Alors par existence et unicité de i , $c : S \rightarrow \llbracket 1, k \rrbracket$ est bien défini. De plus, si $(u, v) \in A$, alors σ satisfaisant C , $\sigma(x_{u,c(u)}) = V \implies \sigma(x_{v,c(u)}) = F$ donc $c(v) \neq c(u)$.

Ainsi, I_{SAT} est une instance positive de SAT si et seulement si G, k en est une de coloration. De plus, cette instance est de taille $O(|A| \times k)$ (et créer en ce temps là), qui est polynomial si on se limite aux instances non triviales de coloration, où donc $k < |A|$. □

Définition 1.2 *Une instance de SOUS-SOMME est une famille de nombre $(s_i)_{i \in \llbracket 1, n \rrbracket} \in \mathbb{N}$ et une cible $K \in \mathbb{N}$. On dit qu'une instance est positive si*

$$\exists I \subset \llbracket 1, k \rrbracket : \sum_{i \in I} s_i = K$$

Remarque 1.1 *A priori, ce problème est difficile dès que n devient grand, car on a pour I , 2^n choix.*

Théorème 1.2 *Il existe une transformation polynomiale d'une instance de SOUS-SOMME vers SAT.*

Démonstration.

Notation

- On identifiera ici V avec 1 et F avec 0.
- Notons également $M = \max(\max\{\lceil \log s_i \rceil / i \in \llbracket 1, n \rrbracket\}, \lceil \log K \rceil)$.
- De plus, on notera $\bar{x} = (x_i)_{i \in \llbracket 0, M \rrbracket} \in \{0, 1\}^{M+1}$.
- Notons $b_{i,j} \in \{0, 1\}$ le j -ième bit de s_i et $b_{K,j}$ le j -ième bit de K .

La somme On cherche tout d'abord à créer une formule pour la somme. On cherche donc $F(\bar{x}, \bar{y}, \bar{z}, \bar{r})$ tel que F soit vrai si et seulement si $\sum_{i=0}^M x_i 2^i + \sum_{i=0}^M y_i 2^i = \sum_{i=0}^M z_i 2^i$ avec r_i la retenue de la i -ème addition.

Pour cela on crée

$$\begin{aligned}
 F(\bar{x}, \bar{y}, \bar{z}, \bar{r}) = & \neg r_{-1} && \text{(car la retenue d'entrée est 0)} \\
 \wedge & \bigwedge_{i=0}^M (x_i \oplus y_i \oplus r_{i-1}) = z_i && \text{(on ajoute les deux bits et la retenue précédente)} \\
 \wedge & \bigwedge_{i=0}^M r_i = ((x_i \wedge y_i) \vee (x_i \wedge r_{i-1}) \vee (r_{i-1} \wedge y_i)) && \text{(Il y a une retenue si au moins deux bits étaient 1)} \\
 \wedge & \neg r_M && \text{(Il ne faut pas de dépassement de capacité)}
 \end{aligned}$$

Le choix du sous-ensemble On introduit maintenant les variables $(c_i)_{i \in \llbracket 1, n \rrbracket}$ dont on voudra qu'elle représente $i \in I$.

Notre instance de SAT On crée alors l'instance

$$\begin{aligned}
 I_{SAT} = & F\left((b_{1,j} \times c_1)_{j \in \llbracket 0, M \rrbracket}, (b_{2,j} \times c_2)_{j \in \llbracket 0, M \rrbracket}, \bar{h}_2, \bar{r}_2\right) \\
 \wedge & \bigwedge_{i=3}^{n-1} F\left(\bar{h}_{i-1}, (b_{i,j} \times c_i)_{j \in \llbracket 0, M \rrbracket}, \bar{h}_i, \bar{r}_i\right) \\
 \wedge & F\left(\bar{h}_{n-1}, (b_{M,j} \times c_M)_{j \in \llbracket 0, M \rrbracket}, (b_{K,j})_{j \in \llbracket 0, M \rrbracket}, \bar{r}_M\right)
 \end{aligned}$$

où les $h_{i,j}$ et les $r_{i,j}$ sont des variables fraîches

Commentaire 1.1 *Là il faut expliquer à l'oral pourquoi ça marche, dans les deux sens. On peut ensuite discuter du fait qu'elle n'est pas en FNC, puisque F n'y est pas, mais presque puisque on peut remplacer les formules sur moins de 6 variables par des formules en FNC en faisant la table de vérité.*

On peut enfin discuter de la taille de l'instance, et pourquoi elle est linéaire en la taille de l'entrée.

□

Développement 2

Correspondance entre les arbres binaires et les arbres généraux

Auteur·e·s: Emile Martinez

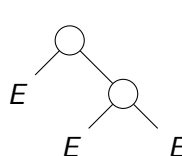
Références :

Présente en OCaml la correspondance entre les arbres binaires et les arbres généraux. Ce développement peut tout à fait s'insérer dans la leçon sur les arbres, tout comme dans la leçon sur le principe d'induction


Objectif Stocker les arbres généraux à n noeuds sous formes d'arbres binaires à n noeuds.

Remarque 2.1 Comment est-ce possible? Les arbres binaires ne sont ils pas inclus dans les arbres généraux?

Et bien non, car dans les arbres généraux, il n'y a pas d'arbres vides. En effet,



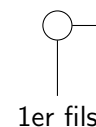
sont différents, alors que pour les arbres généraux, on a un seul arbre à 2 noeuds



Stratégie On va stocker un arbre général sous la forme

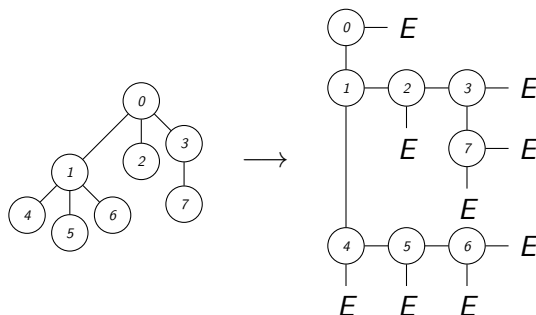


ou encore



1er fils

Exemple 2.1



Remarque 2.2 On remarque que le fils droit de la racine, c'est E . En effet, on a envie de dire que la racine n'a pas de frère droit.

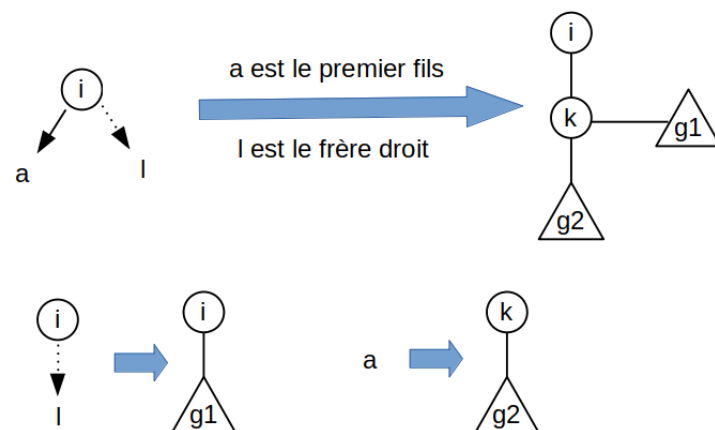

```

type bin = E | B of int*bin*bin;;

type arbre = N of int*arb_liste
and
type arb_liste = V | Cons of arbre*arb_liste;;

let rec conversion (arb : arbre): bin = match arb with
| N(i, E, E) -> B(i, E, E)
| N(i, Cons(a,l)) -> let B(j, g1, d1) = conversion N(i,l) in
                      let B(k, g2, d2) = conversion a in
                      B(i, B(k, g2, g1), E)
;;

```



Il faut alors montrer que :

1. conversion est bien définie
2. conversion $N(x, .)$ est de la forme $B(x, ., E)$
3. conversion est injective
4. $| \text{conversion } a | = | a |$
5. Pour $B(x, b, E)$, $\exists a : \text{conversion } a = B(x, b, E)$

Chaque preuve se fera par induction, on ne fera donc pas tout. Montrons 1 et 5.

Démonstration. Preuve de 1

Montrons par induction sur la structure d'arbre que conversion a termine toujours, et renvoie quelque chose de la forme $B(. , . , .)$.

★ Cas de bases : conversion($N(x, V)$) termine et renvoie $B(x, E, E)$ donc la propriété est vérifiée.

Commentaire 2.1 On peut dire ici que il n'y a pas de cas de bases pour les arbres, donc on prend le cas de bases pour les la définition des arbres en incluant celles des listes, ce qui nous fait un cas de bases pour les listes.

Commentaire 2.2 Pour justifier ce fait là, il faut pointer ce qui se passe et quel code s'exécute sur l'algorithme

* Supposons la propriété d'induction vraie pour $N(x, l)$ et a deux arbres. Alors, $\text{conversion}(N(x, l))$ termine et est de la forme $B(\dots)$ donc le premier let termine.

De même, par propriété d'induction, $\text{conversion}(a)$ termine et est de la forme $B(\dots)$ donc le deuxième let termine.

Donc $\text{conversion}(N(x, l))$ termine et renvoie $B(i, B(\text{etc} \dots))$ ce qui est bien de la forme $B(\dots)$.

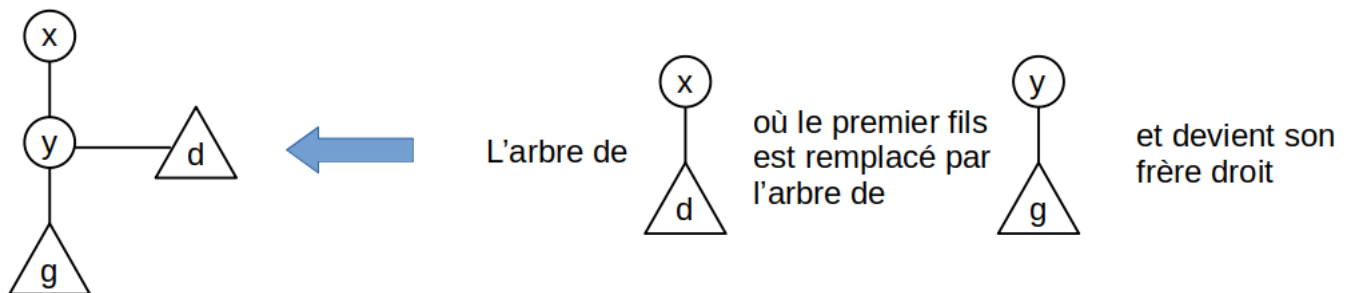
Ainsi par induction structurelle, conversion termine toujours. □

Démonstration. Preuve de 5

Montrons par induction sur la structure d'arbre binaire de b que pour tout $x \in \mathbb{N}$, $\exists a : \text{conversion } a = B(x, b, E)$

* Cas de bases : soit $x \in \mathbb{N}$. Alors $\text{conversion } N(x, V) = B(x, E, E)$. Donc la propriété est vraie sur les cas de bases

* Soient g et d deux arbres binaires vérifiant la propriété. Soit $x, y \in \mathbb{N}$. On cherche donc a tel que $\text{conversion } a = B(x, B(y, g, d), E)$.



Par hypothèse d'induction on a qu'il existe un arbre b tel que $\text{conversion } b = B(x, d, E)$ et il existe un arbre c tel que $\text{conversion } c = B(y, g, E)$.

En notant $b = N(x, l)$ (par 2), on a $\text{conversion}(N(x, c : l)) = B(x, B(y, g, d), E)$.

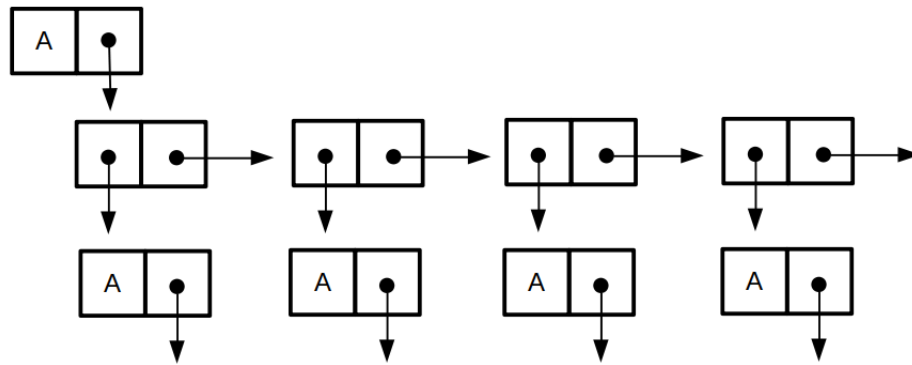
Commentaire 2.3 *La il faut montrer pourquoi quand on l'injecte dans l'algo, indéniablement, cela fonctionne, en le faisant étape par étape.* □

Application 2.1 *On peut utiliser ce code pour encoder les arbres en C par des arbres binaires.*

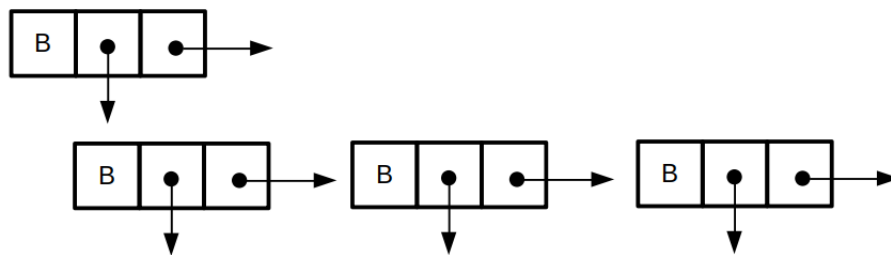
Commentaire 2.4 *Expliquer à l'oral pourquoi que c'est pratique, que les parcours se font plus facilement, que l'encodage d'un arbre binaire est quand même bien plus simple qu'un arbre générique.*

En réalité, notre codage revient au codage en C où mais en plus simple.

Commentaire 2.5 *Le codage en C étant celui où stocke un arbre comme une valeur et un pointeur vers une liste chaînée, elle même pointant vers les arbres fils. Dans notre transformation, on élude alors le problème de la liste chaînée.*



Transformation en arbre binaire



Développement 3

Illustration du paradigme orienté objet sur la modélisation d'une personne sur une carte

Auteur·e·s: Emile Martinez

Références :

Dans cette leçon on déroule un exemple d'utilisation de l'orienté objet, avec la modélisation et l'aspect modulaire, mais également une illustration de ce qui est fonctionnel, impératif, etc...

Objectif Représenter une personne voulant se déplacer sur une carte.

Commentaire 3.1 Construire le diagramme suivant petit à petit, et ne rajouter les éléments seulement quand on en a besoin. (par exemple, la File, ne la rajouter que quand on en aura besoin)

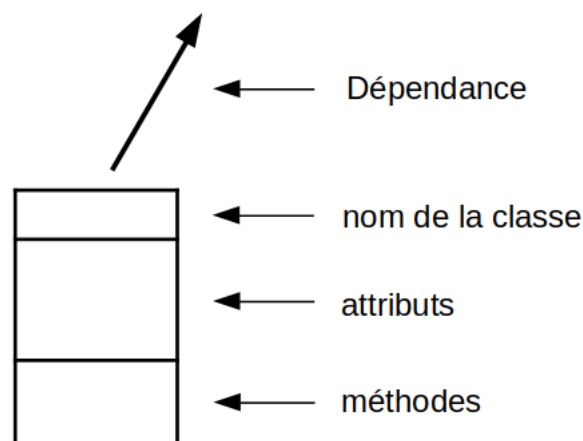


FIGURE 3.1 – Explication des champs de notre diagramme

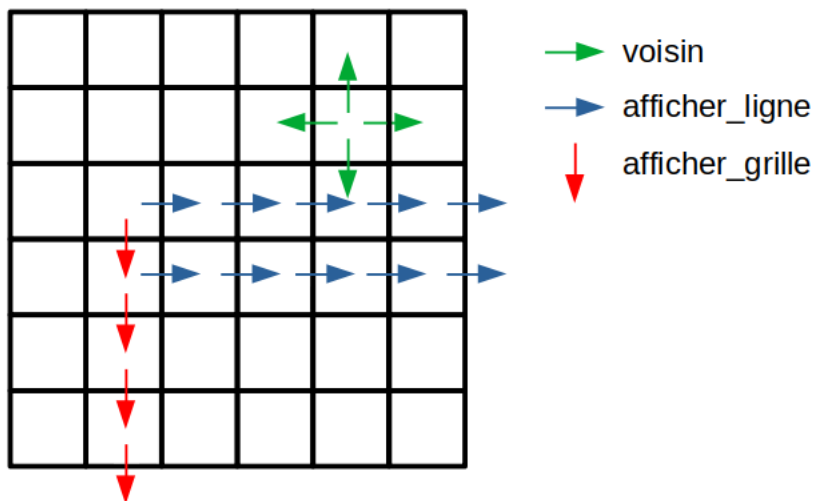
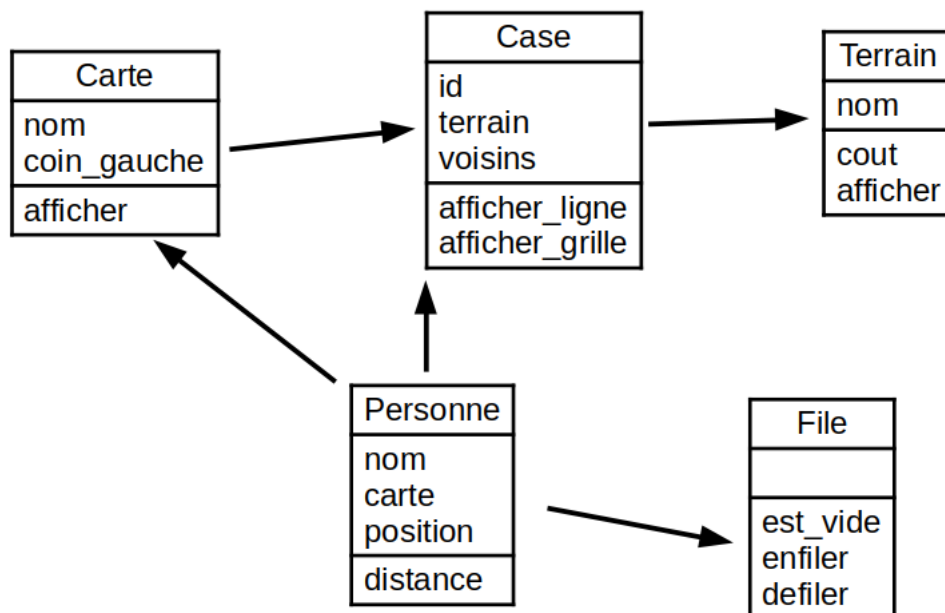


FIGURE 3.2 – Explication de la carte et du fonctionnement de afficher



```

class Carte:
    def __init__(self, ...):
        self.nom = ...
        self.coin_gauche = ...

    def afficher(self):
        self.coin_gauche.afficher_grille()
  
```

```

class Case:
    def __init__(self, ...):
        self.terrain = ...
        self.voisins = ... #dictionnaire de clé "haut", "gauche", etc...

    def afficher_ligne(self):
        self.terrain.afficher()
        if "droite" in self.voisins:
            self.voisins["droite"].afficher_ligne()
  
```

```

else :
    print() # on met une nouvelle ligne

def afficher_grille(self):
    self.afficher_ligne()
    if "bas" in self.voisins:
        self.voisins["bas"].afficher_grille()

```

Commentaire 3.2 Expliquer ici pourquoi on fait un parcours en largeur, en montrant sur la figure 3.2 que on peut ainsi trouver un chemin en plus ou moins ligne droite pour y aller

Commentaire 3.3 Introduire ici la nécessité d'une structure de file comme il n'en existe pas de natives intéressantes en python, et donc la rajouter au diagramme UML

```

class Personne:
    def __init__(self, ...):
        self.nom = ...
        self.carte = ...
        self.position = ...

    def distance(self, dest):
        distance = dict()
        a_voir = File()
        a_voir.ajouter(self)
        distance[self.id] = 0
        while not a_voir.est_vide():
            c = a_voir.defiler()
            if c.id == dest:
                return distance[c.id]
            else:
                for v in c.voisins.values():
                    if v.id not in distance:
                        distance[v] = distance[c] + v.terrain.cout()
                        a_voir.enfiler(v)

        return -1

```

Commentaire 3.4 Mentionner ici l'importance de la modularité, puisque on peut alors utiliser file indépendamment de son implémentation.

Commentaire 3.5 Dire que ici cohabite le fonctionnel et l'impératif, en marquant d'une couleur les lignes récursives (celles où on affiche) et d'une autre l'impératif (l'algo de la distance)

Commentaire 3.6 Dire pour le jury que là on a écrit au tableau mais que en vrai on le projetterai écrit, et que là on a pas mis beaucoup de commentaires, la spécification, etc..., mais que en vrai évidemment dans le code on le ferait, parce que c'est très important.

Développement 4

Approximation(s) gloutonne(s) de Indep(2)

Auteur·e·s: Emile Martinez

Références : Daphné Kany

Cette leçon présente une (ou éventuellement 2 suivant le temps qu'on met à la faire) approximation gloutonne de Indep(2). On suppose ici la définition formelle de Indep déjà faites dans le plan de cours

Instance n tâches de durée $\omega_1, \dots, \omega_n$

Pb Trouver un ordonnancement sur P_1, P_2 qui minimise la date de fin τ

Algorithme 4.1 : Glouton-1(w_1, \dots, w_n)

$\tau_1, \tau_2 \leftarrow 0$ # date de fin de P_1, P_2

pour i allant de 1 à n **faire**

si $\tau_1 \leq \tau_2$ **alors**

$alloc[i] \leftarrow 1$

$\sigma[i] \leftarrow \tau_1$

$\tau_1 = \tau_1 + \omega_i$

sinon

 #idem avec $\tau_1 \leftrightarrow \tau_2$

retourner $\max(\tau_1, \tau_2)$

Proposition 4.1 Glouton-1 n'est pas optimal.

Démonstration. Soit I une l'instance $\omega_1 = \omega_2 = 1$ et $\omega_3 = 2$. On obtient alors

P_1

1	2	
---	---	--

P_2

1		
---	--	--

$$\tau = \tau_1 = 3$$

$$\tau^* = 2$$

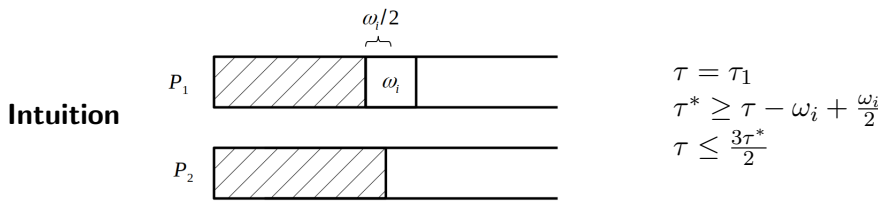
□

Théorème 4.1 Glouton-1 est une $\frac{3}{2}$ -approx de Indep(2)

Démonstration. • D'abord, Glouton-1 est bien polynomial

- Soit I une instance. On note τ la réponse de Glouton-1, τ^* l'optimal.

Il faut alors montrer que $\frac{\tau}{\tau^*} \leq \frac{3}{2}$



Commentaire 4.1 Ici pour expliquer, on commence par dire que on suppose que le processeur le plus occupé c'est le 1, puis on regarde la dernière tâche qu'on a mis dessus. Alors, quand on l'a mise, P_2 était plus avancé. Donc, ce moment là est plus grand que τ^* sans ω_i . Or quand on va rajouter ω_i , au mieux on pourra ne rajouter que $\frac{\omega_i}{2}$ à chacun des processeurs en réorganisant. Donc τ^* avec ω_i c'est au moins τ^* sans ω_i , plus $\frac{\omega_i}{2}$, donc $\underbrace{\tau - \omega_i}_{\leq \tau^* \text{ sans } \omega_i} + \frac{\omega_i}{2}$

Preuve de l'intuition On note $S = \sum_{i=1}^n \omega_i$

$$\tau^* \geq \frac{S}{2} \quad (1)$$

$$\tau_1 + \tau_2 = S \quad (2)$$

On considère $\tau = \tau_1$, et ω_i est la dernière tâche de P_1 .
 Au moment de l'insertion de ω_i : $\underbrace{\tau_1^{(i)}}_{=\tau - \omega_i} \leq \tau^{(i)} \leq \tau^{(i)} \quad (3)$

$$(3) \quad \tau_1 \leq \tau_2 - \omega_i \leq S - \tau_1 + \omega_i \quad (2)$$

$$2\tau_1 \leq \underbrace{S}_{\leq 2\tau^*} + \underbrace{\omega_i}_{\leq \tau^*} \leq 3\tau^* \quad (1)$$

□

Algorithme 4.2 : Glouton-2($\omega_1, \dots, \omega_n$)

$w_1, \dots, w_n \leftarrow \text{Tri}(\omega_1, \dots, \omega_n)$

retourner Glouton-1(w_1, \dots, w_n)

Théorème 4.2 Glouton-2 est une $\frac{7}{6}$ -approx

Démonstration. On reprend la preuve de Glouton-1, mais en essayant d'améliorer $\tau^* \geq \omega_i$

★ Soit $i \leq 4$, Glouton-2 est optimal

→ Faire tous les cas.

Commentaire 4.2 Si il reste du temps, dire que parce que soit il faut mettre une tâche toute seule, et on le fait, soit il faut mettre la plus grande avec la plus petite, et on le fait.

★ Sinon $i \geq 5$. Alors, on a $\tau^* \geq 3\omega_i$ (Car il y a un ruban avec 3 éléments valant au moins ω_i). D'où $2\tau_1 \leq 2\tau^* + \frac{\tau^*}{3}$ i.e. $\tau \leq \frac{7}{6}\tau^*$

□

Exemple où la borne est atteinte Sur l'instance $\omega_1 = \omega_2 = 3$ et $\omega_3 = \omega_4 = \omega_5 = 2$, on obtient

P_1	3	2	2	
P_2	3	2		

au lieu de

P_1	2	2	2	
P_2	3	3		

Remarque 4.1 Avec Glouton-2 on perd la propriété d'être en ligne.

Avec Glouton-2, l'approx est mauvaise surtout pour les cas où on a peu de tâches. On pourrait alors combiner une approche exhaustive pour les premières tâches, et une approche gloutonne pour les petites.

Commentaire 4.3 Si on a du temps, on peut aussi parler de ce qui se passe quand on a plus de processeur. On peut dire sur le dessin, que ω_i se répartit sur plus de processeurs, et donc $\tau^* \geq \tau_1 - \omega_i + \frac{\omega_i}{p}$. Ca nous fait au final une $2 - \frac{1}{p}$ approx pour Glouton-1 et une $\frac{4p-1}{3p}$ pour Glouton-2

Développement 5

Présentation de l'algorithme des k-plus proches voisins

Auteur·e·s: Emile Martinez

Références :

Mettre une description

Problème On a des exemples $E = \{(x, c) \in X \times Y\}$ et on a $x \in X$ dont on cherche sa classe. Ici $X = \mathbb{R}^d$

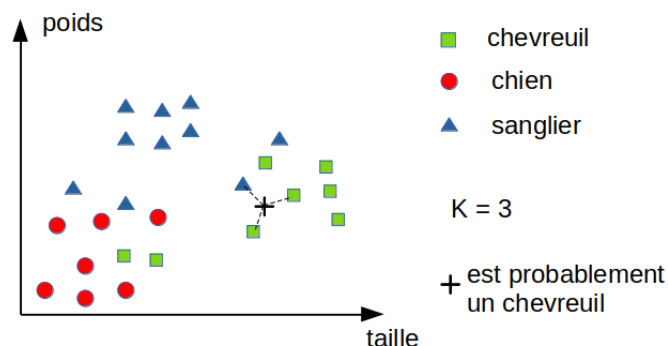
Commentaire 5.1 Ici on parle de Y , mais c'est une liste de classes.

Paramètre $k > 0$ un entier

Algorithme 5.1 Algorithme des k -plus proche voisins

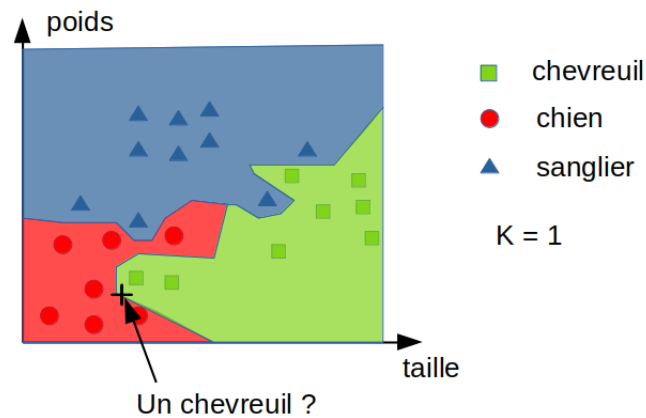
1. On cherche les k plus proches voisins de x dans T
2. On renvoie la classe majoritaire parmi ces k classes

Exemple 5.1 Imaginons qu'on ait un capteur permettant de détecter le poids d'un animal qui passe et sa taille. Comment savoir quel animal était-ce ?



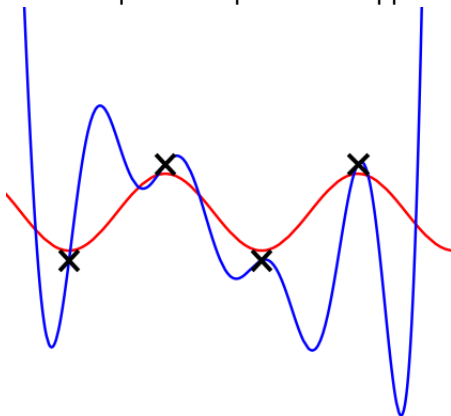
Question Quel choisir k ?

Commentaire 5.2 Si k est trop petit.



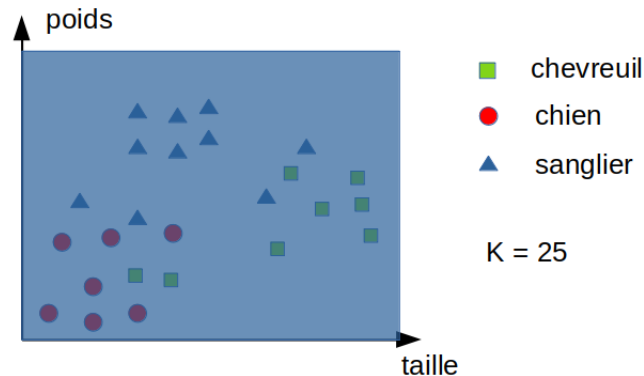
→ On a une trop grande influence des cas particuliers → sur apprentissage.

Exemple classique de sur apprentissage quand on veut approximer des points par une courbe :



La ligne bleue colle mieux aux données mais à l'air moins bien que la rouge.

Influence lointaine Si K est choisi trop grand, des points trop loin auront une trop grande influence.



Commentaire 5.3 *Super il faut le choisir bien, mais ça nous dit pas comment.*

Comment le choisir ? On en essaie plein.

On sépare E en D_A et D_T (avec $\frac{|D_A|}{|E|} \simeq 80\%$). On essaye alors de prédire D_T en utilisant D_A et on regarde pour quel k on a la meilleure performance.

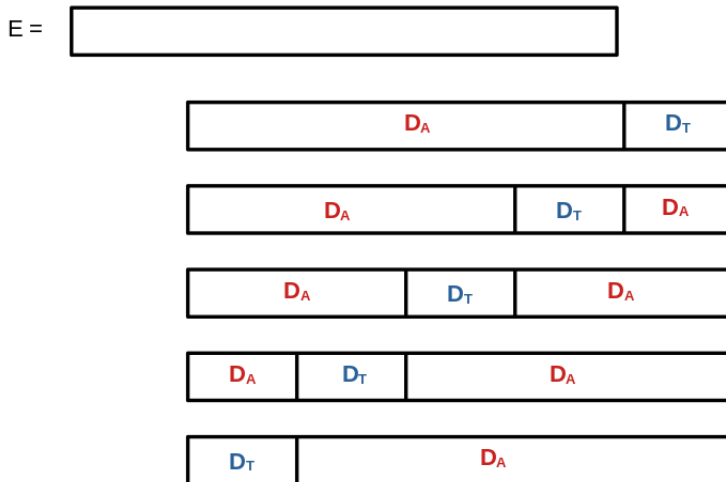
Question Pourquoi séparer ?

→ Car si on faisait sur les mêmes éléments, on éviterait pas le biais du surapprentissage.

Commentaire 5.4 Ici revenir sur l'exemple pour le montrer. En disant que avec $k = 1$ on aurait 100% de réussite.

Validation croisée Si on a pas assez de données ?

Commentaire 5.5 Chaque donnée joue alternativement le rôle de données de test et de données d'entraînement.



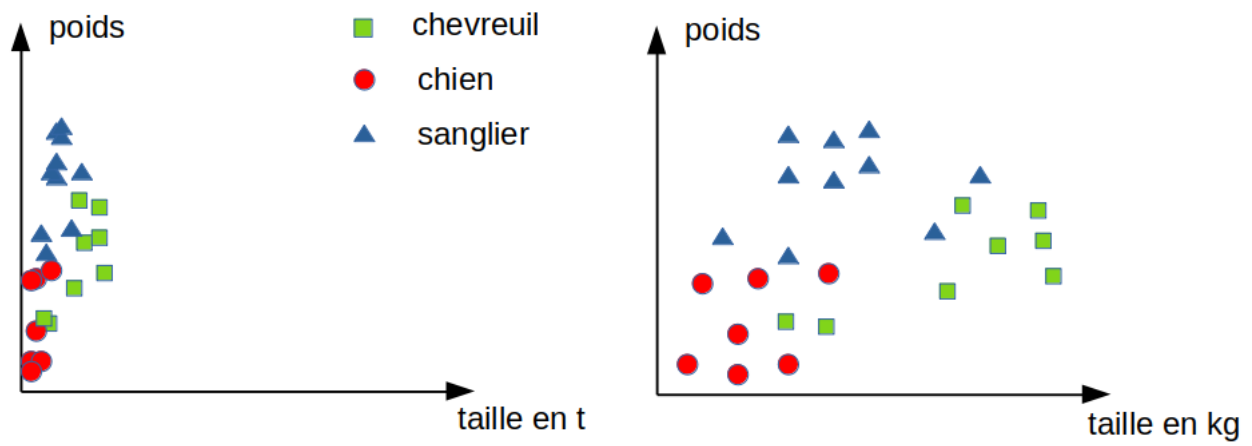
Complexité Comment chercher les K plus proches voisins parmi N points ?

- En cherchant pour i allant de 1 à K le point le plus proche puis en l'enlevant.
→ $O(N \times K)$
- En parcourant les N points et en gardant à chaque fois les K plus proches (quand on en trouve un plus petit, on enlève le plus loin parmi les K).
→ $O(N \times \log K)$ si on utilise les structures de données adéquates
- En faisant des pré-calculs, on peut obtenir une structure permettant de faire la recherche des K plus proche voisins en $O(\log N + K)$ en moyenne. → rentable pour beaucoup de recherche.

Commentaire 5.6 Faire l'analogie avec la dimension 1 et les listes triées.

Notion de distance On utilise ici la distance euclidienne mais on pourrait également utiliser d'autres distances, comme la distance de Manhattan.

Normalisation des paramètres Il se peut qu'une coordonnée soit beaucoup plus grande que toutes les autres et ait beaucoup trop d'importances. Et même, quand on compare des grandeurs différentes, comment choisir l'unité que l'on prend (influençant le poids de cette grandeur)



Développement 6

Validité de la construction d'un ensemble inductif

Auteur·e·s: Emile Martinez

Références :

Ce développement présente le fait qu'une construction inductive est valide. Il consiste à démontrer qu'on peut définir seulement le nom des fonctions et qu'on arrivera à en faire quelque chose. Puis l'équivalence des définitions par le haut et par le bas. On se place dans un contexte où on dit que les constructeurs sont de $X^{\alpha(i)}$ dans X , injectif et d'image disjointes. Si on le met dans le leçon principe d'induction, on niveau MP11, il faut dire que c'est à la fin du cours, on fait ça pour ceux qui veulent suivre en distribuant un poly et en faisant ça au tableau. Avec notamment le fait que c'est la même chose que ce qu'on écrit, et que ce qu'affiche Ocaml (d'où le fait de le faire après Ocaml)

Définition 6.1 Quelques définitions pour commencer

$$\Sigma = ['a' - 'z'] \quad \Gamma = \Sigma \cup \{ '[', ']', '\#' \}$$

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n \quad \Gamma^* = \bigcup_{n \in \mathbb{N}} \Gamma^n$$

$$\text{On définit également : } \begin{array}{ccc} \Gamma^* \times \Gamma^* & \rightarrow & \Gamma^* \\ (u, v) & \mapsto & (u_1, \dots, u_{|u|}, v_1, \dots, v_{|v|}) \end{array}$$

Remarque 6.1 $(\Sigma^n \times \Sigma^m) = \Sigma^{n+m}$

Commentaire 6.1 Dire que 'a', 'b', ..., '[' c'est vraiment simplement des symboles mathématiques différents les uns des autres. Que la puissance n c'est les suites finies, donc on prend l'ensemble des suites finies. La définition de la concaténation on peut la rajouter dans les def après le paragraphe suivant

De quoi on part Au début, on a seulement les noms des éléments de la signature $(\mathcal{B}, (f_i)_{i \in I})$:

$$\mathcal{B} \subset \Sigma^* \quad \forall i \in I, \begin{cases} nom_i \in \Sigma^* \\ nom_i \notin \mathcal{B} \\ nom_i \neq nom_j \text{ pour } j \neq i \end{cases}$$

Construction de nos ensembles Pour X on va prendre Γ^* .

$$\mathcal{B} = \mathcal{B} \subset \Sigma^* \subset \Gamma^*$$

$$f_i : \begin{array}{ccc} \Gamma^{\alpha(i)} & \rightarrow & \Gamma^* \\ (u_1, \dots, u_{\alpha(i)}) & \mapsto & \text{nom}_i \cdot [\cdot u_1 \cdot \# \cdot \dots \cdot \# \cdot u_{\alpha(i)} \cdot] \end{array}$$

Remarque 6.2 Qu'est-ce que c'est ce truc ?

C'est ce que l'on fait quand on écrit l'enchaînement des constructeurs et qu'il n'y a pas d'ambiguïté. C'est la même chose que fait Ocaml, quand dans l'interpréteur il vous affiche un objet construit par des constructeurs :

<pre>type bin = E B of bin*int*bin;; ajout (ajout (ajout E 1) 2) 3;;</pre>	→	$- : \text{bin} = B (B (E, 1, E), 2, B (E, 3, E))$
--	---	--

que nous on note $B[B[E\#1\#E]\#2\#B[E\#3\#E]]$

•

$$\begin{aligned} f_i(u_1, \dots, u_{\alpha(i)}) &= f_j(v_1, \dots, v_{\alpha(j)}) \\ \Downarrow \\ \text{nom}_i \cdot [\cdot u_1 \cdot \# \cdot \dots \cdot \# \cdot u_{\alpha(i)} \cdot] &= \text{nom}_j \cdot [\cdot u_1 \cdot \# \cdot \dots \cdot \# \cdot u_{\alpha(j)} \cdot] \\ \Downarrow * \\ \text{nom}_i &= \text{nom}_j \\ \Downarrow \\ i &= j \end{aligned}$$

*[$\notin \text{nom}_i$ et [$\notin \text{nom}_j$ donc le premier [est celui après nom
Donc pour $i \neq j$, $\text{Im}(f_i) \cap \text{Im}(f_j) = \emptyset$

•

$$\forall b \in \mathcal{B}, \forall i, \forall u_1, \dots, u_{\alpha(i)} \in \Gamma^*, \left\{ \begin{array}{l} \in f_i(u_1, \dots, u_{\alpha(i)}) \\ \notin b \end{array} \right\} \implies b \neq f_i(u_1, \dots, u_{\alpha(i)})$$

• Reste l'injectivité des f_i . Mais en fait elles ne sont pas injectives.

Commentaire 6.2 Là on peut dire que en fait il faudrait modifier légèrement X , mais la construction des f_i est la même, et c'est simplement technique. On peut également mentionner à l'attention du jury que ça ferait un super sujet pour plus tard en MPI.

Définition de l'ensemble inductif Maintenant qu'on a l'existence de nos constructeurs, il faut savoir si notre définition des ensembles inductifs est correctes.

Commentaire 6.3 Dans la leçon, on aurait la définition de $T_0 = \mathcal{B}$ et $T_{k+1} = T_k \cup \bigcup_{i \in I} f_i(T_k^{\alpha(i)})$

Notons P_S la propriété défini sur $\mathcal{P}(X)$ $P_S(A) : \mathcal{B} \subset A$ et $\forall i \in I, f_i(A^{\alpha(i)}) \subset A$

Commentaire 6.4 P_S veut dire contient \mathcal{B} et stable par tous les f_i

Prenons comme définition de l'ensemble inductif E défini par la signature $(\mathcal{B}, (f_i)_{i \in I})$, le plus petit ensemble contenant \mathcal{B} et stable par tous les f_i .

★ Premièrement, cet ensemble existe-t-il ?

Oui. J'annonce même que c'est $\bigcap_{\substack{A \in \mathcal{P}(X) \\ P_S(A)}} A$.

En effet,

- $(\forall A \in \mathcal{P}(X), P_S(A) \implies \mathcal{B} \subset A) \implies \mathcal{B} \subset \bigcap_{\substack{A \in \mathcal{P}(X) \\ P_S(A)}} A$
 - Soit $i \in I$, soit $x_1, \dots, x_{\alpha(i)} \in \bigcap_{\substack{A \in \mathcal{P}(X) \\ P_S(A)}} A$.
- Alors $\forall A \in \mathcal{P}(X), P_S(A) \implies \begin{cases} P_S(A) \\ x_1, \dots, x_{\alpha(i)} \in A \end{cases} \implies f_i(x_1, \dots, x_{\alpha(i)}) \in A$
- Donc $\bigcap_{\substack{A \in \mathcal{P}(X) \\ P_S(A)}} A$ est stable par tous les f_i
- Soit $A \in \mathcal{P}(X)$ tel que $P_S(A)$. Alors, $\bigcap_{\substack{A \in \mathcal{P}(X) \\ P_S(A)}} A \subset A$.

Ainsi, $\bigcap_{\substack{A \in \mathcal{P}(X) \\ P_S(A)}} A$ est bien le plus petit ensemble vérifiant P_S .

Commentaire 6.5 Si on manque de temps pendant le développement, on peut ne pas faire cette preuve.

Reste maintenant à montrer que $E = \bigcup_{k \geq 0} T_k$

★ \square

- $\mathcal{B} = T_0 \subset \bigcup_{k \geq 0} T_k$
 - Soit $i \in I$ et $x_1, \dots, x_{\alpha(i)} \in \bigcup_{k \geq 0} T_k$
Alors par définition, $\forall j \in \llbracket 1, \alpha(i) \rrbracket, \exists k_j \in \mathbb{N} : x_j \in T_{k_j}$
Posons $K = \max_j k_j$
Alors, comme $T_k \subset T_{k'}$ pour $k \leq k'$ (par une récurrence immédiate par définition des T_k),
 $\forall j \in \llbracket 1, \alpha(i) \rrbracket, x_j \in T_K$. Donc par définition de T_{K+1} , $f_i(x_1, \dots, x_{\alpha(i)}) \in T_{K+1} \subset \bigcup_{k \geq 0} T_k$
- Ainsi, $\bigcup_{k \geq 0} T_k$ est stable par tous les f_i

D'où $P_S\left(\bigcup_{k \geq 0} T_k\right)$, donc par définition de E , $E \subset \bigcup_{k \geq 0} T_k$

★ \square Soit \mathcal{P} la propriété défini pour $k \in \mathbb{N}$ par $\mathcal{P}(k) : \ll T_k \subset E \gg$

- Par définition de E , $T_0 = \mathcal{B} \subset E$ d'où $\mathcal{P}(0)$.
 - Soit $k \in \mathbb{N}$ tel que $\mathcal{P}(k)$. Soit $y \in T_{k+1}$. Alors par définition de T_{k+1} on a deux possibilités :
 - Si $y \in T_k$ alors par $\mathcal{P}(k)$, $y \in E$
 - Si $\exists i \in I : \exists x_1, \dots, x_{\alpha(i)} \in T_k : y = f_i(x_1, \dots, x_{\alpha(i)})$
Donc par $\mathcal{P}(k)$, $\forall j \in \llbracket 1, \alpha(i) \rrbracket, x_j \in E$. Or E est stable par tous les f_i donc $y = f_i(x_1, \dots, x_{\alpha(i)}) \in E$.
- D'où $\forall y \in T_{k+1}, y \in E$, soit $\mathcal{P}(k+1)$.

Ainsi par principe de récurrence, $\forall k \in \mathbb{N}, T_k \subset E$, donc $\bigcup_{k \geq 0} T_k \subset E$.

Développement 7

Preuve de l'équilibrage des arbres rouges noirs et méthode d'insertion

Auteur·e·s: Emile Martinez

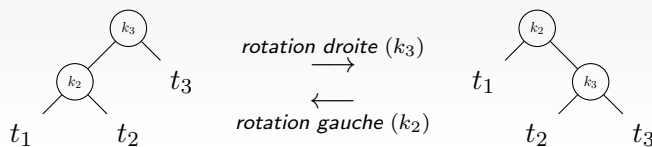
Références : Daphné Kany

On estime que la définition des ARN est déjà admise.

Insertion dans un ABR

1. On colorie la racine en noir
2. On insert le noeud en tant que feuille comme dans un ABR
On le colore en rouge
3. On rétablit la propriété (ii) par rotations successives en préservant la hauteur noire.

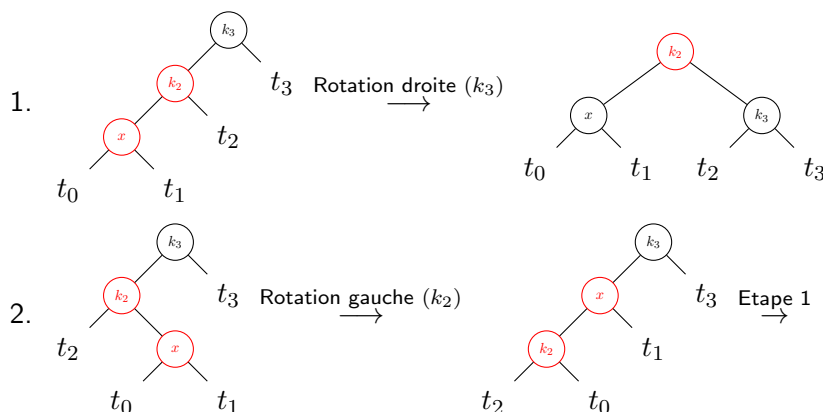
Définition 7.1 Rotation



Remarque 7.1 La rotation préserve la structure d'ABR

Commentaire 7.1 Expliquer la remarque à l'oral sur le dessin de la définition

Etape 3 de l'insertion 4 cas à considérer :

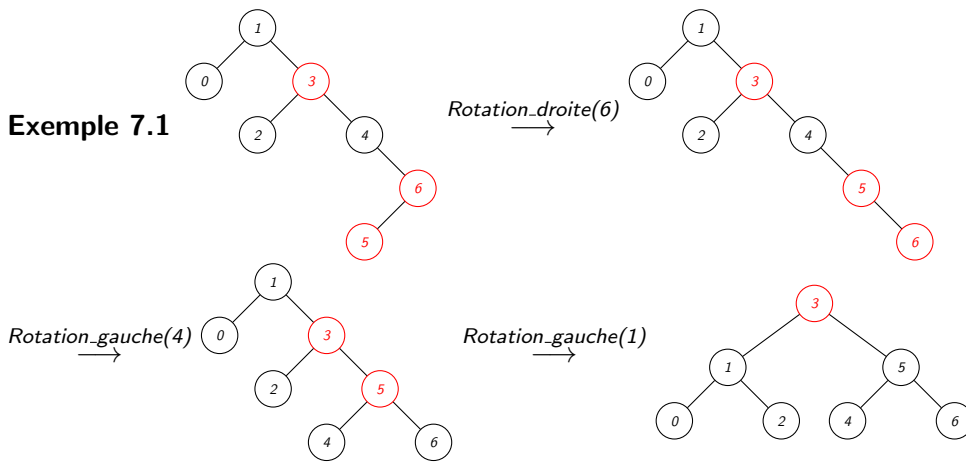


3. Pareil

4. Pareil

Invariant

- On a toujours au plus, en tout, 2 noeuds rouges consécutifs
- La hauteur noire est préservée

Exemple 7.1**Complexité :**

1. $O(h)$
2. $O(h)$
3. $O(1)$

Proposition 7.1 Soit A un ARN de hauteur h à n noeuds, alors $h = O(\log n)$

Lemme 7.1 Soit A un ARN. Alors :

1. $h \leq 2h_N$ où h_N est la hauteur noire.
2. $2^{h_N} \leq n + 1$

Démonstration.

1. Soit C un chemin de la racine à une feuille de longueur h .

On note $h_R(C)$ le nombre de noeuds rouges de C et $h_N(C)$ le nombre de noeuds noirs de C .

$$|C| = h = h_R(C) + h_N(C).$$

D'après (ii) (de la def) : $h_R(C) \leq h_N(C)$ D'après (iii) : $h_N(C) = h_N$.

Donc, $h \leq 2h_N$.

2. Procédons par induction

Commentaire 7.2 Si on a pas le temps ici on écrit simplement induction.

- ★ Si l'arbre est vide, le résultat est immédiat.
- ★ Soit \mathcal{A} un ARN dont les deux fils \mathcal{A}_g et \mathcal{A}_d vérifient la propriété (ce sont bien des ARN). Alors,
 - Si la racine de \mathcal{A} est rouge, $2^{h_N(\mathcal{A})} = 2^{h_N(\mathcal{A}_g)} \leq |\mathcal{A}_g| + 1 \leq |\mathcal{A}| + 1$

$$\begin{aligned}
 \text{— Si la racine de } \mathcal{A} \text{ est noire, alors } 2^{h_N(\mathcal{A})} &= 2^{h_N(\mathcal{A})-1} + 2^{h_N(\mathcal{A})-1} = 2^{h_N(\mathcal{A}_g)} + 2^{h_N(\mathcal{A}_d)} \leq \\
 &\underbrace{|\mathcal{A}_g| + 1 + |\mathcal{A}_d| + 1}_{=|\mathcal{A}|} \leq \text{HI}
 \end{aligned}$$

□

Ainsi, $h_N \leq \log(n+1)$ donc $h \leq 2\log(n+1)$. Donc $h = O(\log n)$

Insertion : $O(\log n)$

Développement 8

Généralisation du tri par comptage à l'aide de dictionnaires

Auteur·e·s : Emile Martinez

Références : Emile Martinez, Daphné Kany, sur une idée influencé par chat GPT

Algorithme 8.1 : Tri par comptage

Entrées : Un tableau T d'entiers positifs

$|T| = n$

Sorties : T trié

$N \leftarrow \max(T) \# O(n)$

$T2 \leftarrow$ tableau de taille $N + 1$ initialisé à 0.

pour x dans T **faire**

$T2[x] \leftarrow T2[x] + 1$

$indice \leftarrow 0$

pour i de 0 à N **faire**

pour j de 0 à $T2[i]$ **faire**

$T[indice] \leftarrow i$

$indice \leftarrow indice + 1$

retourner T

Exemple 8.1 $T = [2, 1, 5, 1, 2]$

$N = 5$

$T2 = [0, 2, 2, 0, 0, 1]$

Remarque 8.1 Complexité spatiale : $O(N)$

Complexité temporelle : $O(n + N)$

Commentaire 8.1 Dire que si N vaut 500 000 000 on l'a dans le

Ce tri n'est pas en place.

Problème : Comment généraliser cet algorithme pour des tableaux contenant d'autres valeurs ?

Algorithme 8.2 : Généralisation du tri par comptage**Entrées** : Un tableau T d'entiers / Un tableau T d'éléments de E d'éléments de E Une fonction $f : E \rightarrow \mathbb{N}$ **Sorties** : T trié $d \leftarrow$ dictionnaire vide**pour** x dans T **faire** $i \leftarrow \text{recherche}(d, x)$ / $l \leftarrow \text{recherche}(d, f(x))$ **si** $l \neq \text{None}$ **alors** Ajouter x à l $\text{Insertion}(d, x, i + 1)$ / $\text{Insertion}(d, f(x), l)$ **sinon** $\text{Insertion}(d, x, 1)$ / $\text{Insertion}(d, f(x), [x])$ $T_2 = []$ **pour** k dans d **faire** Ajouter k à T_2 Trier T_2 $\text{indice} \leftarrow 0$ **pour** i dans T_2 **faire** $k \leftarrow \text{recherche}(d, i)$ / $l \leftarrow \text{recherche}(d, i)$ **pour** j de 0 à $k - 1$ / $x \in l$ **faire** $T[\text{indice}] \leftarrow i$ / $T[\text{indice}] \leftarrow x$ $\text{indice} \leftarrow \text{indice} + 1$ **Complexité** :Spatiale : $O(n)$ Temporelle : $O(n \times (C_i + C_r) + C_{tri}(m))$ en notant $m = |T_2|$.Implémentation en ARN : $O(n \log m + C_{tri}(m)) = O(n \log m + m \log m) = O(n \log m)$ Implémentation en table de hachage : $\begin{cases} \text{pire cas} & O(n \times m + C_{tri}(m)) \\ \text{cas moyen} & O(n + C_{tri}(m)) = O(n + m \log m) \end{cases}$

Avec des tables de hachage, on obtient en moyenne sur les insertions, dans le pire cas (où $n = m$) une complexité en $O(n) + C_{tri}(n) \sim C_{tri}(n)$ (car les meilleurs algorithmes de tri par comparaison sont au pire en $\Omega(n \log n)$). Ainsi, dans le pire cas, on a asymptotiquement le même nombre de comparaison.

Remarque Ce tri ne concerne que des entiers. On peut passer à n'importe quelle structure que l'on compare à travers une fonction entière par les modifications en rouge.

Exemple 8.2 $T = ['abc', 'hello', 'world', 'aa', 'bc']$ f : fonction qui compte les caractères

3 : ['abc']

d : 5 : ['hello', 'world']

2 : ['aa']

Proposition 8.1 On obtient alors un tri stable (avec une complexité spatiale en $O(|T|)$)

Commentaire 8.2 On connaît des algorithmes de tri par comparaison qui dans leur meilleurs cas sont linéaires (le Tim Sort de python, quand la liste est déjà trié), et dans le pire en $O(n \log n)$. Notre algorithme n'est donc pas pertinent dans toutes les situations. Il l'est si l'on veut trier des éléments avec beaucoup de redondances (ex : les français par code postaux).

Commentaire 8.3 *Cette remarque peut éventuellement être écrite si on manque de temps*

Commentaire 8.4 *A ne faire que si vraiment on a trop trop de temps :*

On peut également utiliser cet algorithme pour gagner des constantes. En effet, si on connaît $f : E \rightarrow \mathbb{N}$ croissante où $|f^{-1}(i) \cap T| = \sqrt{n}$, alors on peut partitionner sur ces classes, trier ces classes, puis trier à l'intérieur de ces classes. (on obtient alors une complexité en $O(n) + \sqrt{n} \log \sqrt{n} + \sqrt{n} \times \sqrt{n} \log \sqrt{n} + \sqrt{n} \times o(\sqrt{n} \log \sqrt{n}) = \frac{1}{2}n \log n + o(n \log n)$)

Développement 9

Présentation et terminaison de l'algorithme de Bellman-Ford

Auteur·e·s: Emile Martinez

Références :

A présenter éventuellement dans les leçons de réseaux, en commençant par l'introduire comme un algo de routage (justifiant d'autant plus les remarques à la fin)

Algorithme 9.1 : Bellman Ford

Entrées : $G = (S, V, w)$ un graphe pondéré non orienté avec V le tableau des listes d'adjacence

$proch_saut \leftarrow$ tableau indexé par $S \times S$

$proch_saut[u, v]$ contiendra le premier noeud où aller pour aller de u à v

$D \leftarrow$ tableau indexé par $S \times S$

pour $u, v \in S$ **faire**

$proch_saut[u][v] \leftarrow None$

$D[u][v] \leftarrow +\infty$

pour $u \in S$ **faire**

$D[u][u] \leftarrow 0$

répéter

pour $u \in S$ **faire**

pour $v \in V[u]$ **faire**

pour $s \in S$ **faire**

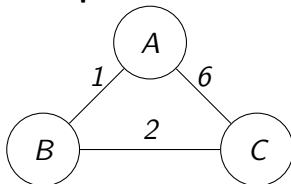
si $D[v][s] + w(u, v) < D[u][s]$ **alors**

$proch_saut[u, s] \leftarrow v$

$D[u][s] \leftarrow D[v][s] + w(u, v)$

jusqu'à stabilisation;

Exemple 9.1



$D :$

	A	B	C
A	0	$+\infty$	$+\infty$
B	$+\infty$	0	$+\infty$
C	$+\infty$	$+\infty$	0

 \rightarrow

	A	B	C
A	0	1	6
B	1	0	2
C	6	2	0

 \rightarrow

	A	B	C
A	0	1	3
B	1	0	2
C	3	2	0

Question Est-ce que notre algorithme termine ?

Première réponse Oui. A chaque passage, si on a pas stabilisation, alors une case de D diminue. Les cases de D ne pouvant que diminuer, et ne contenant que des entiers, $\sum_{(u,v) \in S} D[u][v]$ est donc un variant de boucle.

Commentaire 9.1 Ici, on raye au lieu de réécrire à chaque fois, et on explique oralement comment on obtient chaque case (pas toute mais voilà) en faisant référence à l'algo.

Deuxième Réponse Super ! Mais combien d'itérations fait-on ? Pour cela, montrons un résultat intermédiaires :

Lemme 9.1 En notant D^i le tableau après la i -ième itération de la boucle, on a $\mathcal{P}(i) : \ll D^i(u, s)$ est le plus court chemin (pcc) de u à s avec au plus i sauts \gg

Démonstration.

- ★ $\mathcal{P}(0)$ est vrai car le seul endroit où on peut aller en 0 sauts, c'est sur soi-même, qui est à distance 0.
- ★ Soit $i \in \mathbb{N}$ tel que $\mathcal{P}(i)$.

$$\text{Alors, } D^{i+1}[u][s] = \min \left(D^i[u][s], \min_{v \in V(u)} D^i(v, s) + w(u, v) \right)$$

Or, le pcc de u à s de au plus $i + 1$ sauts est :

- soit de au plus i sauts
- soit commence par aller vers un voisin v de u puis est un pcc de v à s de au plus i sauts.

Or, tous ces chemins sont valides, et par $\mathcal{P}(i)$, on prend bien le minimum de tout ça. D'où, $\mathcal{P}(i + 1)$

Ainsi, par principe de récurrence, $\forall i \in \mathbb{N}, \mathcal{P}(i)$

□

De plus, un pcc ne passe pas deux fois par le même sommet (car on suppose $w \geq 0$) donc un pcc est de longueur au plus $|S|$. Ainsi, $\mathcal{P}(|S|)$ et $\mathcal{P}(|S| + 1)$ impliquent que la $|S|$ -ième et la $|S| + 1$ -ième itérations sont les mêmes. On a donc au plus $|S|$ itérations (et en plus notre algo est correcte !)

Commentaire 9.2 Ce n'est pas le vrai résultat que l'on calcule car on met à jour D^i au fure et à mesure. (la on a fait la preuve comme si dans l'algo, on faisait au début de la boucle $D' \leftarrow D$, puis qu'on modifiait dans D' et enfin on finit la boucle principale par $D \leftarrow D'$)

Complexité Ainsi cet algorithme est en $O(|S| \times \sum_{u \in S} |V(S)| \times |S|) = O(|S|^2 \times |A|)$

Commentaire 9.3 Suivant le temps, on peut soit détailler un peu plus le calculs, soit même dire que c'est du $O(|S|^4)$ puis ensuite devenir raisonnable et faire remarquer qu'en fait c'est potentiellement moins. Et si on fait ça dire que c'est important, parce que souvent la plupart des éléments ne sont pas connectés.

Commentaire 9.4 Dire à l'oral que en fait on connaît mieux

Alors pourquoi l'utiliser ? A cela plusieurs raisons :

- On peut facilement le paralléliser : chaque nœud u peut calculer son propre $D[u]$
- On peut même le distribuer, car là montrer sur l'algo on n'a besoin que de $D[v]$ pour ses voisins v . Il suffit alors d'échanger avec ses voisins les vecteurs de distance (d'où le nom de routage par vecteur de distance)
- On ne dévoile pas trop la topologie du réseau : On ne connaît que les distances des voisins à tous les nœuds, mais pas comment y accéder → on ne connaît pas quels routeurs sont où dans les autres réseaux.

Inconvénients

- La convergence est plus lente que avec Dijkstra
- En cas de panne d'un lien ou d'un nœud, on peut se retrouver avec des boucles et une convergence encore plus lente

Remarque 9.1 *En théorie des graphes, cet algo peut aussi être utilisé pour détecter des cycles de poids négatifs (en ne prenant pas $w \geq 0$), en regardant si au bout de $|S|$ itérations on est pas encore à stabilité.*

Développement 10

Protocole HTTPS

Auteur·e·s: Daphné Kany

Références : NSI Tle

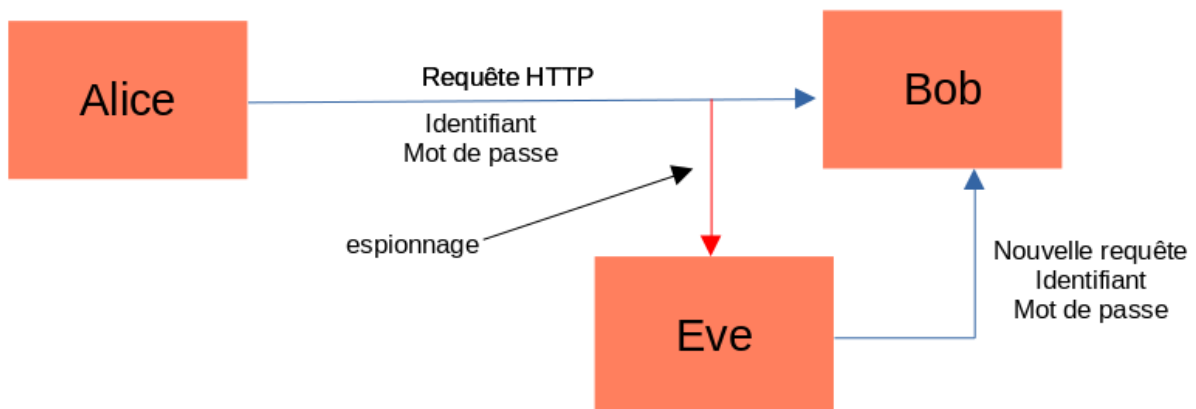
Dans cette leçon on présente le S du protocole HTTPS.

Motivation : les attaques HTTP

Le protocole HTTP est en clair : les messages ne sont pas chiffrés

Commentaire 10.1 Lorsque Alice et Bob communiquent en utilisant le protocole HTTP, ils peuvent subir plusieurs attaques. Premièrement, leurs messages ne sont pas chiffrés. Ainsi, si Alice communique une information sensible comme un mot de passe, il peut être intercepté par un utilisateur malveillant sur le réseau. Ensuite, Alice ne vérifie jamais l'identité de Bob. Elle peut être victime d'une attaque de l'homme au milieu : Eve, un serveur malveillant, se fait passer pour Bob auprès d'Alice et pour Alice auprès de Bob.

Attaque de l'espion



Comment éviter ces potentielles attaques ?

- Communication chiffrée : Un utilisateur qui intercepte le paquet ne peut plus le lire
- Authentification : Alice a l'assurance qu'elle est bien en communication avec Bob.

Chiffrement des données

Définition 10.1 Soit deux individus A et B qui communiquent un message m chiffré à l'aide d'une clef k , et déchiffrable à l'aide d'une clef k' . On dit que le chiffrement est symétrique si $k = k'$, et asymétrique sinon

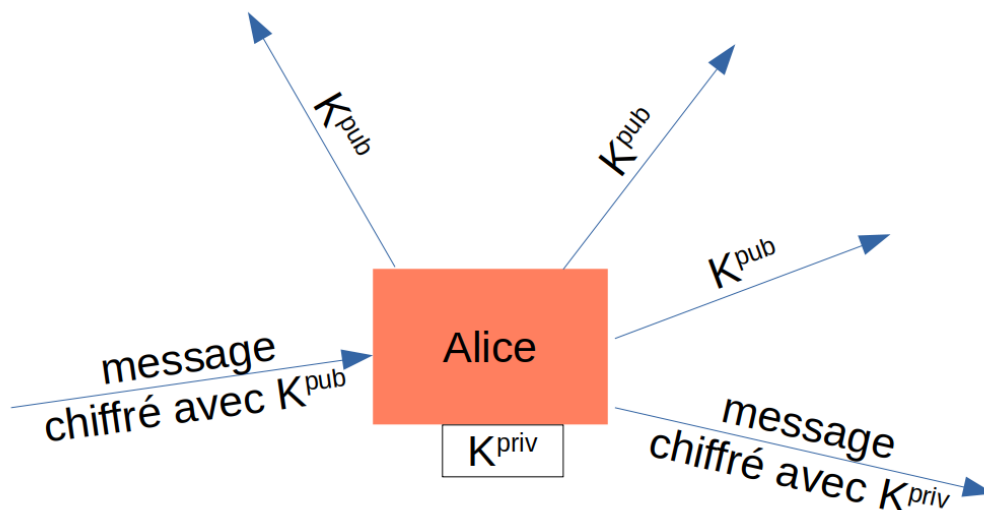
Exemple 10.1 Le chiffrement de César, qui consiste à décaler les lettres du message de k caractères, est un chiffrement symétrique.

Commentaire 10.2 Problème : Avant d'utiliser une clef commune, A et B doivent se la communiquer. S'ils le font en clair, un espion pourra plus tard déchiffrer leurs messages. D'où l'intérêt des chiffrements asymétriques.

Chiffrement asymétrique

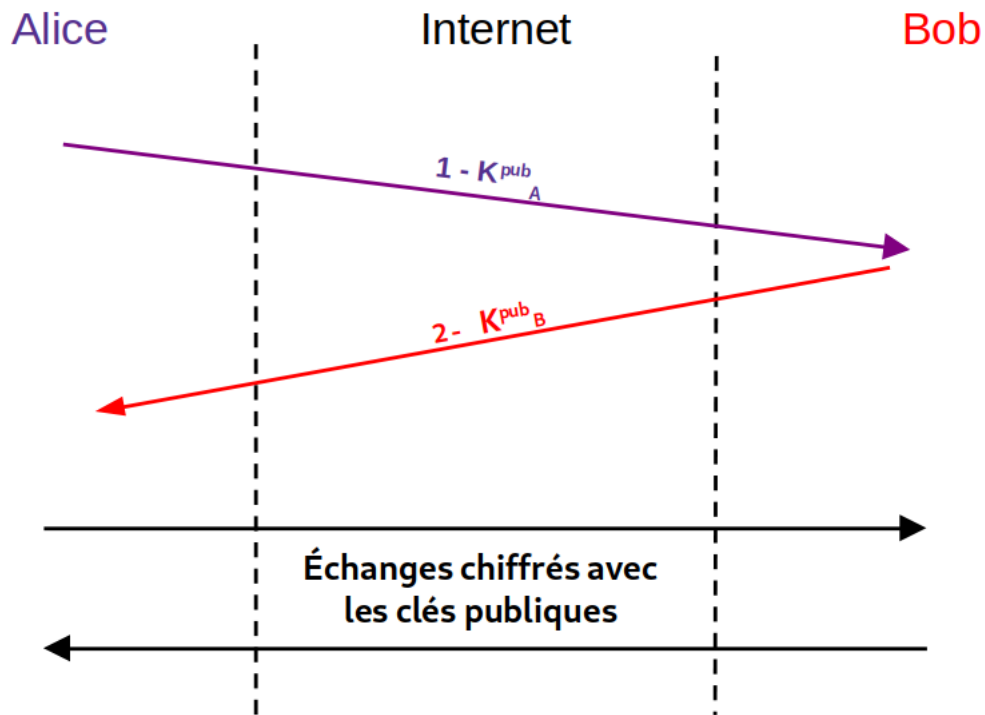
Le chiffrement RSA est un chiffrement asymétrique très utilisé. Les utilisateurs A et B disposent chacun d'une clef publique K_A^{pub} ; K_B^{pub} et d'une clef privée K_A^{priv} ; K_B^{priv} qu'ils ne communiquent jamais. Les messages ont la particularité d'être chiffrable et déchiffrable par les deux clefs :

$$dechiffre(K_A^{priv}, chiffré(K_A^{pub}, m)) = dechiffre(K_A^{pub}, K_A^{priv}, m) = m$$

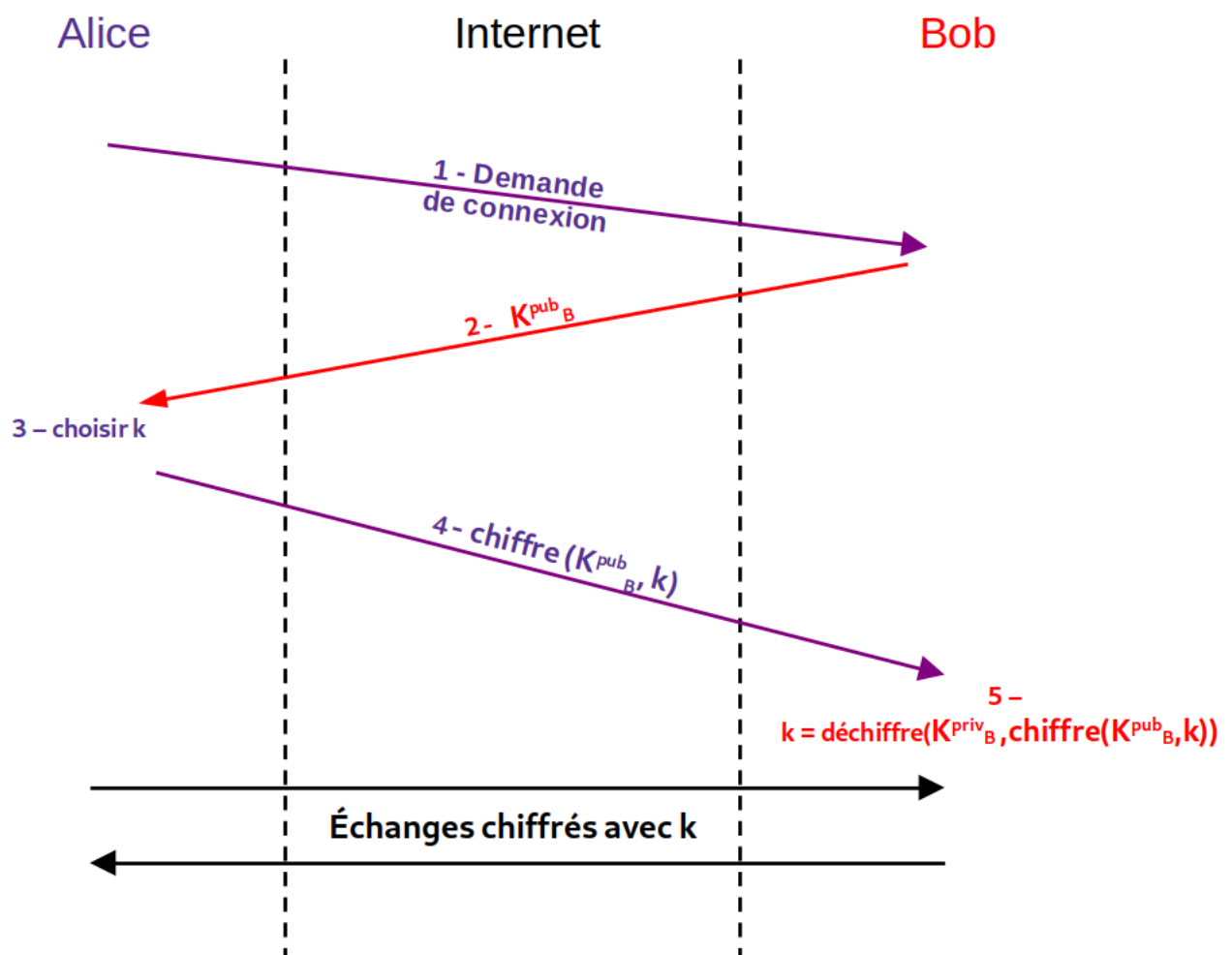


Alice peut donc envoyer des messages qu'elle seule peut envoyer et elle peut recevoir des messages qu'elle seule peut décoder.

Idee On peut alors utiliser cela :



Probleme La fonction de chiffrement asymetrique est lourde à calculer. On l'utilise donc au début d'une communication, pour partager une clef symétrique qui servira au chiffrement des messages ultérieurs.



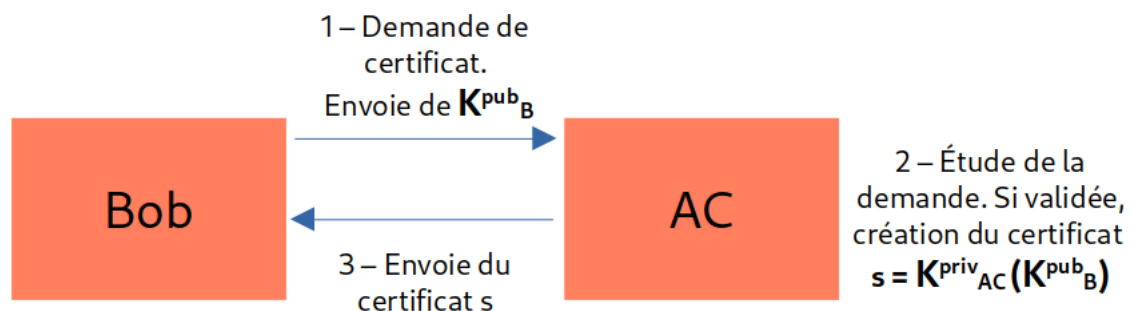
Attaque de l'homme au milieu



Certificat

Le protocole HTTPS repose sur l'authentification du serveur grâce à un certificat délivré par un tiers de confiance (une autorité de certification ou AC). Parmi les AC, on trouve des entreprises spécialisées, des associations à but non lucratifs et des états.

Ces certificats sont créés à partir des clefs RSA des participants.

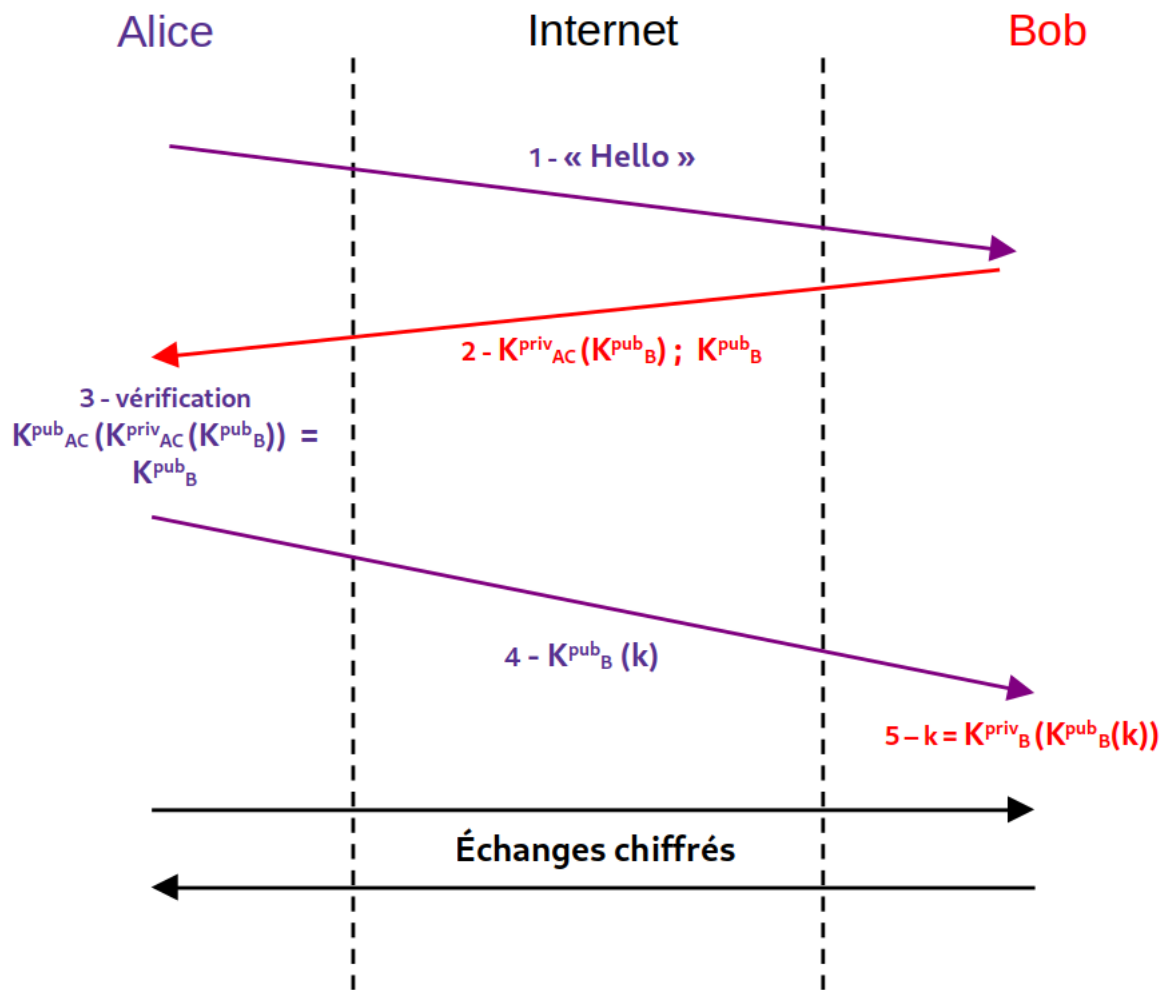


Obtention d'un certificat

Commentaire 10.3 Les certificats ont des dates de péremption (validité de qq mois à qq années) et doivent donc être redemandés régulièrement. Les AC sont très surveillées et peu nombreuses : Modzilla en reconnaît une centaine.

Commentaire 10.4 Protocole HTTPS

- Etape 1 : Alice envoie un message initial "Demande de connexion" et indique les différents algorithmes cryptographiques qu'elle supporte.
- Etape 2 : Le serveur Bob lui répond en envoyant son certificat $s = K_{AC}^{priv}(K_B^{pub})$ et sa clef publique K_B^{pub} .
- Etape 3 : Alice vérifie le certificat grâce à la clef publique de l'AC qu'elle doit posséder.
 $K_{AC}^{pub}(K_{AC}^{priv}(K_B^{pub})) = K_B^{pub}$
- Etape 4 : Alice utilise la clef publique de Bob pour lui communiquer de façon chiffrée une clef symétrique de chiffrement k . Elle lui envoie donc $K_B^{pub}(k)$
- Etape 5 : Bob déchiffre k grâce à sa clef privée.



Authentification HTTPS

La suite du protocole est identique à HTTP, mais tous les messages sont chiffrés avec k .

Robustesse aux attaques

Dans le cas d'une attaque de l'homme au milieu, Eve ne connaît pas la clef privée de Bob et ne pourra donc pas récupérer la clef symétrique k envoyée par Alice.

Si dans le réseau un individu intercepte les messages, il ne pourra pas non plus récupérer k . Les données sont protégées.

Développement 11

Illustration des différents aspects de la méthode diviser pour régner sur le problème de la pyramide

Auteur·e·s: Daphné Kany

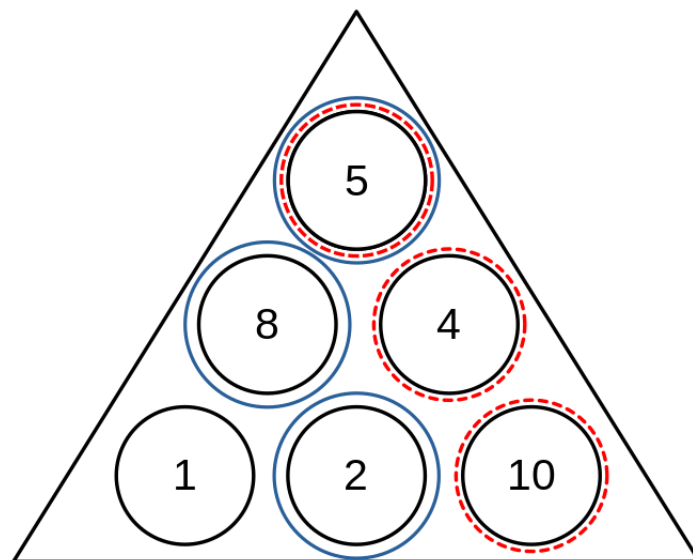
Références : Balabonski MPI

Dans cette leçon on présente le problème de la pyramide. C'est un exemple introductif à la programmation dynamique.

Problème de la pyramide

Entrée : Une pyramide Π de hauteur h remplie d'entiers.

Sortie : La valeur maximale d'un chemin du sommet de la pyramide à sa base.



Exemple : pyramide de hauteur 3. En rouge, la valeur optimale d'un chemin depuis le sommet (19).

Remarque 11.1 Combien de chemins possibles ? 2^h
Un algorithme exhaustif aura une complexité exponentielle.

Approche gloutonne

A chaque étape, on choisit le sommet de plus grande valeur. Complexité : $O(h)$.

Commentaire 11.1 Faire le glouton sur l'exemple au dessus. On trouve 15. Cela nous prouve que l'algorithme glouton n'est pas optimal.

Remarque 11.2 L'algorithme glouton n'est pas optimal.

Programmation dynamique

Étape 1 (création des sous pb) : Si p est une sous pyramide de Π , on note $S(p)$ la valeur max d'un chemin du sommet de p à sa base.

Étape 2 (relation de récurrence) : On note Π_g et Π_d les pyramides filles gauche et droite de Π .

$$S(\Pi) = v(\Pi) + \max(S(\Pi_g), S(\Pi_d))$$

où $v(\Pi)$ est la valeur du sommet de la pyramide.

Étape 3 : Implémentation

Représentation informatique de la pyramide

On va stocker notre pyramide dans un tableau T de dimension $h \times h$.

$T[i,j]$ = jème élément en partant de la gauche à la profondeur i si $j \leq i$

$T :$

5		
8	4	
1	2	10

Remarque 11.3 Les sous pyramides gauches et droites de (i,j) sont $(i+1, j)$ et $(i+1, j+1)$

Méthode naïve sans mémorisation

Algorithme 11.1 : cheminOpt(T, i, j)

Entrées : T le tableau représentant Π ; i et j les indices du sommet considéré

$h = \text{len}(T)$

si $i == h$ **alors**

$_$ retourner $T[i,j]$

sinon

$_$ retourner $T[i,j] + \max(\text{cheminOpt}(T, i+1, j), \text{cheminOpt}(T, i+1, j+1))$

Remarque 11.4 Complexité : $C(h) = 1 + 2C(h-1)$ donc $C(h) = 2^h$. On retombe sur l'algorithme exhaustif qui énumère tous les chemins.

Méthode descendante :

Commentaire 11.2 Faire les modifications directement sur l'algo naïf au tableau

Algorithme 11.2 : $\text{cheminOpt}(T, R, i, j)$

Entrées : T le tableau représentant Π ; i et j les indices du sommet considéré; R stocke les résultats intermédiaires

```

si  $R[i,j] > -\infty$  alors
  └ retourner  $R[i,j]$ 
 $h = \text{len}(T)$ 
si  $i == h$  alors
  └  $R[i,j] = T[i,j]$ 
    └ retourner  $R[i,j]$ 
sinon
  └  $R[i,j] = T[i,j] + \max(\text{cheminOpt}(T, i+1, j), \text{cheminOpt}(i+1, j+1))$ 
    └ retourner  $R[i,j]$ 

```

Remarque 11.5 $C(h) = h^2$ car on remplit le tableau R une seule fois

Méthode ascendante :**Algorithme 11.3 :** $\text{cheminOpt}(T)$

Entrées : T le tableau représentant Π

$h = \text{len}(T)$

$R = \text{tableau } h \times h$

pour j allant de 1 à h **faire**

└ $R[h,j] = T[h,j]$

pour i de $h-1$ à 1 **faire**

└ **pour** j de 1 à i **faire**

└ └ $R[i,j] = T[i,j] + \max(R[i-1, j], R[1-1, j-1])$

retourner $R[1,1]$

Commentaire 11.3 Dérouler la méthode ascendante sur l'exemple du début pour obtenir le tableau :

$T :$

19		
10	14	
1	2	10

Commentaire 11.4 S'il reste du temps : expliquer comment retrouver le chemin à l'aide d'un tableau prochainNoeud .

Développement 12

Premiers pas avec SQL

Auteur·e·s: Emile Martinez

Références :

Cette leçon est là pour présenter vaguement comment interpréter des requêtes SQL

Commentaire 12.1 *On dessine ces tables au milieu du tableau, et on ne les écrits que quand on en a besoin (car si on les écrit toutes au début c'est long)*

num_prod	nom	prix	poids
1	patate	1	1
2	canard	8	0,4
3	haricots	4	2
4	carottes	3	1,5

Table produit

num_prod	num_client	qte
1	1	3
1	2	150
3	1	2

Table commande

num_client	nom	adresse	ville
1	Radis radieux	3 allée du swag	Tarbes
2	Navet navigant	1 allée du caca	Chartres

Table client

Commentaire 12.2 *Ici commencer par dire : examinons cette requête.*

```
SELECT nom, prix
FROM produits
WHERE poids > 1
```

num_prod	nom	prix	poids
1	patate	1	1
2	canard	8	0,4
3	haricots	4	2
4	carottes	3	1,5

→

nom	prix
haricots	4
carottes	3

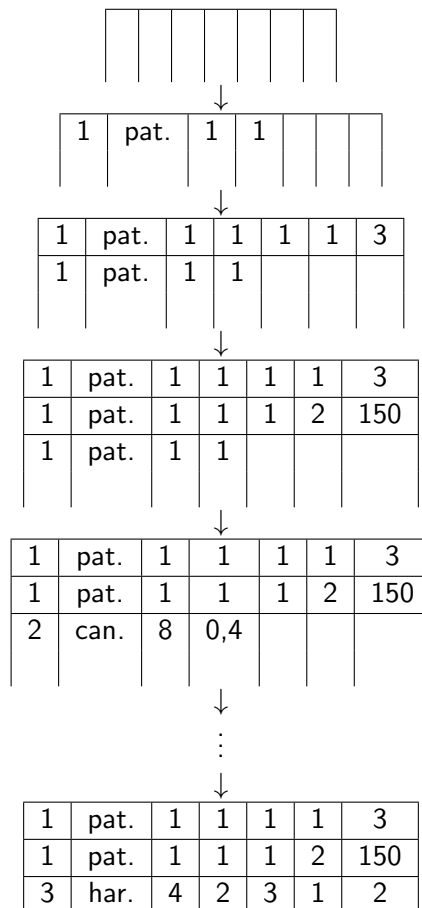
Commentaire 12.3 *Ca vaut le coup de réécrire la table (même si on mets des abréviations pour les éléments)*

On obtient donc les noms et les prix des produits pesant strictement plus de 1 kilo.

Objectif Afficher les noms des produits de chaque commande avec leur quantité

```
SELECT nom, qte
FROM produit AS p JOIN
commande AS c ON p.num_prod = c.num_prod
```

Commentaire 12.4 La j'écris les premières étapes, à faire évidemment sur le même tableau, et j'élude les dernières. Et expliquez que dans la jointure, on cherche les indices qui correspondent, les entrées qui mettent la condition à vrai



→

nom	qte
patate	3
patate	150
haricots	2

Commentaire 12.5 Là on peut passer sur la droite du tableau (pour avoir d'un côté les requêtes de selection et de l'autre celles d'insertion)

Commentaire 12.6 Faire les modifications sur la table, en écrivant de la même couleur que la requête

Objectif Ajouter des commandes

```
INSERT INTO commande
VALUES (2,2,10), (4,1,1)
```

Commentaire 12.7 Quand on fait ça sur la table, parler de la vérification des conditions faites par SQL

Objectif Inflation de la patate

```
UPDATE produit
SET prix = 1.1
WHERE nom = 'patate'
```

Objectif Doublement des commandes du client 1

```
UPDATE commande
SET qte = 2*qte
WHERE num_client = 1
```

Objectif Réalisation de la livraison du produit 1 au client 1

```
DELETE FROM commande
WHERE num_client = 1 AND num_prod = 1
```

num_prod	nom	prix	poids
1	patate	1	1,1
2	canard	8	0,4
3	haricots	4	2
4	carottes	3	1,5

Table produit

num_prod	num_client	qte
1	1	3
1	2	150
3	1	2
2	2	10
4	1	2

Table commande

Une requête plus intéressante Les noms des produits et des clients qui les commandent, dès que la commande dépassent 10 unités

```
SELECT DISTINCT p.nom, c.nom
FROM produit AS p JOIN
      commande AS co ON co.num_prod = p.num_prod JOIN
      client AS c ON c.num_client = co.num_client
WHERE qte > 10
ORDER BY p.nom ASC
```

Remarque 12.1 Le *DISTINCT* est il utile ?

Développement 13

Avec ou sans agrégation SQL

Auteur·e·s: Emile Martinez

Références :

Ce développement a pour but de présenter des exercices avancés de SQL, en présentant différentes manières de faire la division et le max. On peut le présenter comme la correction d'exo dont on aurait dit aux élèves : «pour la prochaine fois, cherchais différentes manières de répondre à ces questions», et qui en vrai, partirait donc de ce qu'ont proposé les élèves

Trouver le(s) produit(s) le(s) plus cher(s)

★ La première manière de faire consiste à trouver le max puis à regarder quels sont les produits qui ont ce prix là.

```
SELECT MAX(prix)
FROM produit
```

→ 10

```
SELECT nom
FROM produit
WHERE prix = 10
```

→
Ce n'est pas très portable

```
SELECT nom
FROM produit ,
      (SELECT max(prix) as m
       FROM produit)
WHERE m = prix
```

Commentaire 13.1 Sur un dessin montrer comment on rajoute m à la fin de chaque ligne.

Remarque 13.1 Peut-on faire sans agrégation ?

★ On peut également ruser avec la clause LIMIT :

```
SELECT nom
FROM produit
ORDER BY prix DESC
LIMIT 1
```

Remarque 13.2 On n'obtient pas tous les produits les plus chers, seulement un.

- ★ On peut néanmoins réussir sans agrégation et proprement.

Pour cela on commence par chercher tous les éléments qui ne sont pas maximum.

```
SELECT DISTINCT p1.num_produit
FROM produit AS p1,
     produit AS p2
WHERE p1.prix < p2.prix
```

	3
	2
	4

produit

	3		3
	3		2
	3		4
	4		3
	4		2
	4		4
	2		3
	2		2
	2		4

Il ne ne reste
bien que les
→ éléments plus
petits que quel-
qu'un d'autre

Il ne suffit alors que d'enlever à tous les produits, ce qui ne sont pas maximaux

```
SELECT nom
FROM produit AS p JOIN
  (SELECT num_produit AS n1
   FROM produit

   EXCEPT

   SELECT DISTINCT p1.num_produit
   FROM produit AS p1,
        produit AS p2
   WHERE p1.prix < p2.prix
  )
ON p.num_produit = n1
```

Commentaire 13.2 On peut éventuellement construire cette requête par étape, en partant de la précédente, et en ajoutant à chaque fois des choses (d'abord le fait de récupérer tous les numéros, puis de les utiliser, quite à mettre des accolades sur chaque portion pour expliquer comment cela fonctionne)

Commentaire 13.3 On peut dire qu'on aurait pu se passer du JOIN, en mettant dans le truc que on excepte le num_produit et le nom, puis en ne selectonnant que le nom

Remarque 13.3 Cela peut évidemment se généraliser à toutes tables avec un attribut comparable, pouvant donc remplacer le MAX agrégatif.

Trouver les clients qui ont commandé tous les produits

- ★ La solution avec agrégation

```
SELECT num_client , COUNT(*)
FROM commande
GROUP BY num_client
```

→ donne le nombre de commandes de chaque client ayant une commande

→ donc son nombre de produits commandés (car (num_prod, num_client) est une clé)

→

```
SELECT num_client
FROM commandes
GROUP BY num_client
HAVING COUNT(*) IN (SELECT COUNT(*)
                     FROM produits)
```

Supposons qu'il y ait 20 produits.

```
SELECT num_client
FROM commandes
GROUP BY num_client
HAVING COUNT(*) = 20
```

Commentaire 13.4 *Suivant comment on annonce ce développement et le temps qu'il prend, on peut rajouter à côté du premier bloc une explication de pourquoi ça marche, en dessinant le fait que on fait des paquets par num_client et que on compte le nombre de lignes pour chaque*

★ On peut aussi le faire sans utiliser d'agrégation

```
SELECT num_client
FROM commande

EXCEPT

SELECT num_client
FROM (
    SELECT num_produit , num_client
    FROM produit , client

    EXCEPT

    SELECT num_produit , num_client
    FROM commande
)
```

tous les couples
produit client
n'étant pas une
commande

tous les
clients ne
commandant
pas au moins
un produit

Tous les
clients
ayant
commandé
tous les
produits

Développement 14

Problème du Rendez-vous

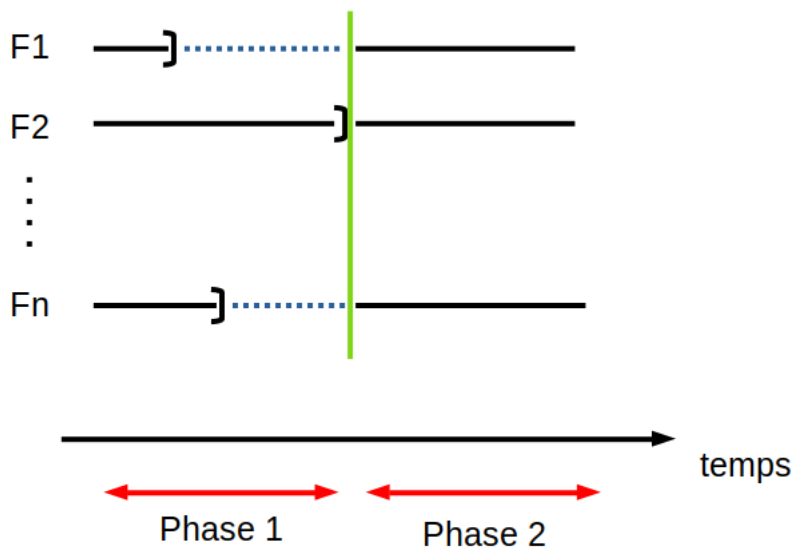
Auteur·e·s: Emile Martinez

Références :

Construction de solutions au problème du RDV

Objectif Synchroniser n fils d'exécution

Commentaire 14.1 Eventuellement dire que la def plus formelle est dans le cours, si elle est écrit. Eventuellement aussi la rappeler, pour dire que y a deux phases, et que la phase 2 commencent quand tous les fils ont fini la phase 1



But : Créer la barrière verte

Version naïve :

```
int compteur = 0;

  Fi :

// Phase 1
compteur ++;
while(compteur < n);

// Phase 2
```


Commentaire 14.2 Dire à l'oral que chaque fils dit qu'il a fini puis attend que tout le monde ait fini, que compteur c'est simplement le nombre de fils qui ont fini.

Remarque

1. On a un problème d'accès concurrent à compteur
2. On a de l'attente active

Commentaire 14.3 Mentionner le fait que l'accès concurrent on pourrait simplement mettre un verrou, mais que pour l'attente active c'est plus pénible.

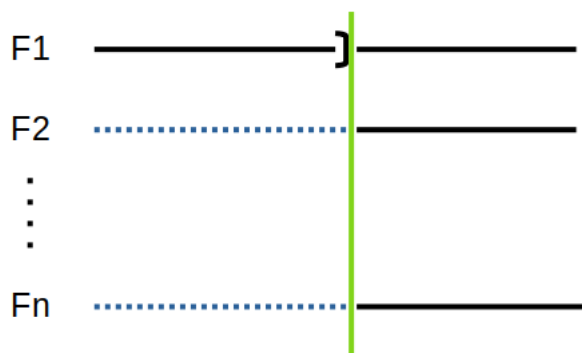
1ere cas facile Voyons le cas où on a seulement deux fils et où l'on sait que c'est F1 qui finit en premier. On peut alors considérer que F2 n'a pas de phase 1.



sem s initialisé à 0

<p>F1 :</p> <pre>// Phase 1 incrémenter(&s) // Phase 2</pre>	<p>F2 :</p> <pre>décrémenter(&s) // Phase 2</pre>
--	---

2ème cas T1 finit en dernier mais on a n fils



sem s initialisé à 0

<p>F1 :</p> <pre>// Phase 1 incrémenter(&s) // Phase 2</pre>	<p>Fi :</p> <pre>décrémenter(&s) incrémenter(&s) // Phase 2</pre>
--	---

Commentaire 14.4 On reprend la même idée que avant. Mais on veut libérer plusieurs fils. On fait alors en sorte que chaque fil en libère un autre. On obtient ainsi des libérations en cascades

Retour au cas général : On reprend la même idée mais en essayant de bloquer que les fils qui ne sont pas les derniers.

Commentaire 14.5 On a maintenant besoin de savoir qui termine en dernier. Pour cela, comme chaque fil, attend d'être libéré, on va faire en sorte que seul le dernier fil puisse être libéré. On a qu'a pour cela avoir un sémaphore initialement négatif, qui ne passera positif que pour le dernier fil.

```
sem s initialisé à -n+1
```

```
Fi :
```

```
// Phase 1
1 incrementer(&s)
2 decrements(&s)
3 incrementer(&s)
// Phase 2
```

Commentaire 14.6 Dire que la première fois que s va devenir strictement positif, c'est quand le n -ième (donc dernier) fil va incrementer s , puis qu'ensuite c'est la même chose que tout à l'heure.

Exemple 14.1 Imaginons que l'on ait 3 fils qui exécute ce code

F1	F2	F3	s
			-2
1			-1
2			-1
⋮	1		0
⋮	2		0
⋮	⋮	1	1
⋮	⋮	2	0
⋮	⋮	3	1
⋮	2		0
⋮	3		1
2			0
3			1

Remarque 14.1 On peut ne pas avoir le droit d'utiliser un sémaphore négatif (dont la définition dit qu'il réveille un fil quand on l'augmente et qu'il passe positif)

Solution Mélanger la solution précédente et la solution naïve.

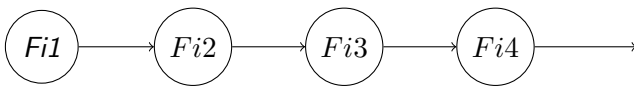
Commentaire 14.7 En effet, dans la naïve on arrive à savoir qui était le dernier fils, mais on arrive pas à faire attendre correctement, et là c'est l'inverse.

```
sem s initialisé à 0
compteur = 0
verrou v_c
```

```
Fi :
```

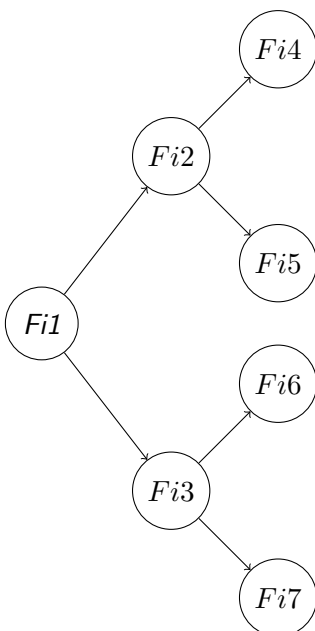
```
//Phase 1
prendre(v_c)
compteur ++;
if (compteur == n){
    rendre(v_c)
    incrementer(&s)
}
else {
    rendre(v_c)
    decrementer(&s)
    incrementer(&s)
}
//Phase 2
```

Remarque 14.2 Ici, chaque fil en libère un autre, faisant une longue chaîne de libération.



Commentaire 14.8 Si on a beaucoup de coeur et beaucoup de fils, que l'on veut qu'il redémarre tous en même temps vraiment, ça peut poser problème. On a supposé nous que les instructions là étaient courtes par rapport à la phase de travail, mais ça peut en pas être le cas si jamais on répète beaucoup de fois cette phase de rendez-vous

On pourrait préférer alors une libération plus arborescente



Remarque 14.3 Ici notre sémaphore termine avec comme valeur 1. On pourrait vouloir le réutiliser pour pouvoir faire une nouvelle barrière. On peut alors remplacer la libération en cascade par seulement le premier fil qui libère tout le monde :

```
for(int i = 0; i < n-1; i++) incrementer(&s)
```

De plus, en gardant le verrou de compteur, on peut empêcher que le rendez vous des phases 2 et 3 interfère avec celui de la phase 1 et 2.

Remarque 14.4 *On pourrait vouloir faire se réunir les fils avec des joins, mais cela impose de tuer les fils (or nous on les conserve) et cela ne s'appliquerait pas à différents processus (ce que permet notre implémentation avec sémaphore).*

Développement 15

Algorithme A*

Auteur·e·s : Daphné Kany

Références :

Ce développement vise à prouver certaines propriétés vérifiées par l'algorithme A selon la nature de l'heuristique utilisée.*

Commentaire 15.1 On peut se contenter d'écrire l'algorithme dans le plan de la leçon pour gagner du temps

Algorithme 15.1 : Algorithme A*

Entrées : W la matrice de poids du graphe ; h le tableau pour l'heuristique ; s_0 et s_f les sommets initiaux et finaux

Sorties : la distance d'un plus court chemin de s_0 à s_f

$D \leftarrow$ tableau initialisé à ∞

$D[s_0] \leftarrow 0$

$P \leftarrow$ file de priorité vide

Ajouter $(s_0, h[s_0])$ à P

while P non vide **do**

$(s, _)$ \leftarrow extract(P)

si $s = s_f$ **alors**

 retourner $D[s]$

pour s' successeur de s **faire**

$c \leftarrow D[s] + W[s, s']$

si $c < D[s']$ **alors**

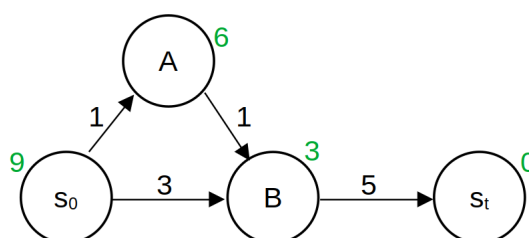
$D[s'] \leftarrow c$

 Ajouter $(s', c + h[s'])$ à P

retourner *NotFound*

Notation Notons $pcc : S^2 \rightarrow \mathbb{N} \cup \{-\infty, +\infty\}$ la fonction renvoyant la distance d'un plus court chemin.

Exemple 15.1 On applique l'algorithme sur le graphe ci dessous. L'heuristique est précisée pour chaque nœud.



$D :$	s_0	A	B	s_f	P
	0	$+\infty$	$+\infty$	$+\infty$	$s_0 : 9$
	0	1	3	$+\infty$	$A : 7; B : 6$
	0	1	3	8	$A : 7; s_f : 8$
	0	1	2	8	$B : 7; s_f : 8$
	0	1	2	7	$s_f : 7$

Remarque 15.1 Contrairement à l'algorithme de Dijkstra, on peut visiter un même sommet plusieurs fois.

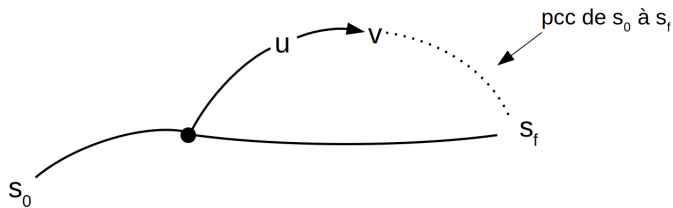
Temps 15.1 3 :20 sans écrire l'algo

Commentaire 15.2 On peut ensuite modifier l'heuristique du noeud A (par exemple en 8). La nouvelle heuristique n'est pas admissible, et l'algorithme va extraire s_f à la quatrième itération, et donc renvoyer une distance qui n'est pas min.

Théorème 15.1 Correction de A^* : Si l'heuristique est admissible, A^* renvoie la distance d'un plus court chemin entre s_0 et s_f s'il existe, et Not found sinon.

Démonstration. Squelette de la preuve :

- ★ On a le variant de boucle suivant : si $D[u] < +\infty$, alors $D[u]$ est la distance d'un chemin de s_0 à u .
 - ★ Si il y a un chemin de s_0 à s_f , alors A^* ne renvoie pas Not Found. En effet, par récurrence, chaque élément du chemin est inséré.
 - ★ A^* renvoie $pcc(s_0, s_f)$.
- Par l'absurde : On a alors, $D[s_f] > pcc(s_0, s_f)$.



Alors dans P , v a au plus la valeur $D[u] + w(u, v) + h(v) \leq pcc(s_0, u) + w(u, v) + pcc(v, s_f) = pcc(s_0, s_f) < D[s_f]$ donc v est extrait avec sa valeur minimale avant s_f .

□

Commentaire 15.3 Si le temps fait défaut, on peut faire uniquement l'initialisation de la récurrence.

Théorème 15.2 Si l'heuristique est monotone et vérifie $h(s_f) = 0$, l'algorithme A^* est correct. De plus, chaque sommet u est extrait au plus une fois de P (en ayant à ce stade $D[u] = dist(s_0, u)$).

Démonstration.

- Correction : Soit h une heuristique monotone tq $h(s_f) = 0$. Montrons que h est admissible.
Soit $u \rightarrow u_1 \rightarrow \dots \rightarrow u_n \rightarrow s_f$ un plus court chemin de u à s_f .

$$h(u) \leq w(u, u_1) + h(u_1) \leq \dots \leq h(s_f) + \sum_{i=0}^{n-1} w(u_i, u_{i+1}) = \text{dist}(u, s_f)$$

- On raisonne par l'absurde et on considère v le premier sommet extrait pour lequel $D[v] > \text{dist}(s_0, v)$.

Soit $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n \rightarrow v$ un plus court chemin de s_0 à v .

On considère s_i le dernier sommet de ce chemin extrait. Alors s_{i+1} non extrait mais inséré dans P .

Lors de l'extraction de v , on a :

$$D[v] + h(v) \leq D[s_{i+1}] + h(s_{i+1}) \leq D[s_i] + w(s_i, s_{i+1}) + h(s_{i+1})$$

Or $D[s_i] + w(s_i, s_{i+1}) = \text{dist}(s_0, s_i) + w(s_i, s_{i+1}) = \text{dist}(s_0, s_{i+1})$

et $h(s_{i+1}) \leq w(s_{i+1}, s_{i+2}) + h(s_{i+2}) \leq \text{dist}(s_{i+1}, v) + h(v)$ (h monotone)

Finalement, on a $D[v] + h(v) \leq \text{dist}(s_0, s_{i+1}) + \text{dist}(s_{i+1}, v) + h(v)$

donc $D[v] \leq \text{dist}(s_0, v)$, ce qui est absurde, d'où la conclusion.

□

Remarque 15.2 On en déduit la complexité de A^* dans le cas d'une heuristique monotone : $O((|V| + |E|)\log(|V|))$ si la file de priorité est implémentée à l'aide d'un tas-min.

Développement 16

Jeu de Nim

Auteur·e·s: Daphné Kany

Références :

Cette leçon présente le jeu de Nim et ses stratégies gagnantes dans le cas d'une puis plusieurs lignes de bâtons. Les conventions d'écritures sont pour l'instant celles du plan de la leçon sur les jeux.

I - Jeu de Nim à un tas

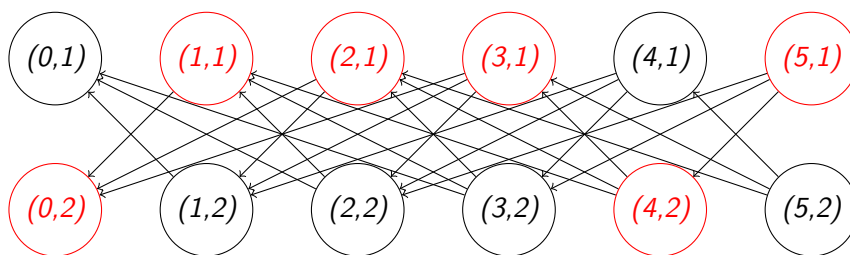
Définition 16.1 Le jeu de Nim est un jeu à deux joueurs J_1 et J_2 . Initialement, on dispose de n bâtons. A tour de rôle, les joueurs retirent au choix 1, 2, ou 3 bâtons. Le joueur qui retire le dernier a gagné.

Modélisation :

Soit $G = (V_1 \sqcup V_2, A)$ le graphe représentant un jeu de Nim à n bâtons.

- Pour $i \in \{1, 2\}$, $V_i = \{(k, i) \mid k \in \{0, \dots, n\}\}$ et $G_i = \{(0, 3 - i)\}$
- $A = \{(k, i) \rightarrow (k', j) \mid k - k' \in \{1, 2, 3\} \text{ et } i \neq j\}$

Exemple 16.1 On représente le graphe pour $n = 5$.



Calcul de l'attracteur de G_1 (positions gagnantes de J_1) :

$$Attr_0(G_1) = \{(0, 2)\}$$

$$Attr_1(G_1) = \{(0, 2)\} \cup \{u \in V_2 \mid N^+(u) = (0, 2)\} \cup \{u \in V_1 \mid (0, 2) \in N^+(u)\} = \{(0, 2), (1, 1), (2, 1), (3, 1)\}$$

$$Attr_2(G_1) = \{(0, 2), (1, 1), (2, 1), (3, 1), (4, 2)\}$$

On montre par récurrence :

$$Attr_i(G_1) = \{(4 * k, 2) \mid k \leq (i - 1)\} \cup \{(u, 1) \mid u \bmod 4 \neq 0 \text{ et } u \leq 4 * i\}$$

Conclusion :

J_1 a une stratégie gagnante ssi (il commence et $n \bmod 4 \neq 0$) ou (J_2 commence et $n \bmod 4 = 0$).

Commentaire 16.1 On voudrait maintenant généraliser le jeu de Nim : Les joueurs ont devant eux k lignes de bâtons. Ils retirent à tour de rôle un nombre strictement positif de bâtons d'une des lignes. Le joueur qui retire le dernier gagne.

II - Jeu de Nim à k tas

Proposition 16.1 Soit un jeu de Nim à k lignes N_1, \dots, N_k , de position initiale respective s_1, \dots, s_k . On pose $S = s_1 \oplus s_2 \oplus \dots \oplus s_k$. Alors J_1 a une stratégie gagnante ssi (il commence et $S > 0$) ou (J_2 commence et $S = 0$).

Démonstration.

Sens indirect : Soit P_i : «Pour une partie de i coups (J_1 commence et $S > 0$) ou (J_2 commence et $S = 0$) $\implies J_1$ a une stratégie gagnante»

Si $i = 0$: Alors l'état initial est $(0, \dots, 0)$ donc J_1 gagne si J_2 commence.

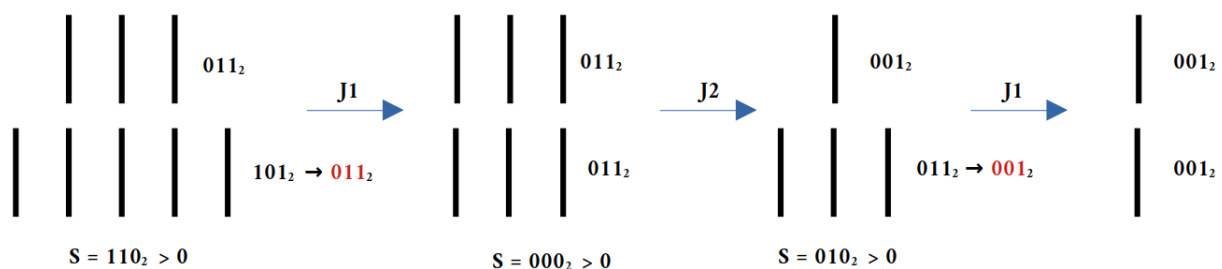
Hérédité : Soit $i \geq 0$ tq P_{i-1} . Montrons P_i .

Supposons J_1 commence et $S \neq 0$. Il existe une ligne N_l tq le bit de poids fort de S est à 1 dans s_l . J_1 retire des bâtons de cette ligne de façon à annuler S . On se ramène alors à une partie à $n-1$ coups dans laquelle J_2 commence et $S = 0$, donc par HR, J_1 a une stratégie gagnante.

Supposons J_2 commence et $S = 0$. Alors quelque soit le coup joué par J_2 , il modifie un bit d'une ligne et donc la valeur de S qui devient $\neq 0$. On se ramène alors à une partie à $n-1$ coups dans laquelle J_1 commence et $S \neq 0$, donc par HR, J_1 a une stratégie gagnante. Ce qui prouve la récurrence.

Par ailleurs, toute partie finit (nombre de bâtons total décroît strictement à chaque tour).

Sens direct : Par contraposée, J_2 a une stratégie gagnante donc J_1 n'en a pas. □



Exemple avec deux lignes

Développement 17

Intérêts et insuffisances des critères de test

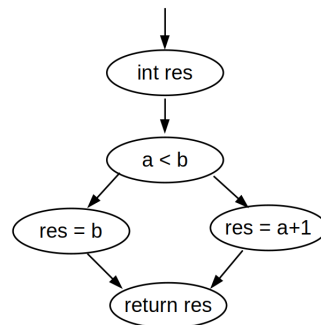
Auteur·e·s: Emile Martinez

Références :

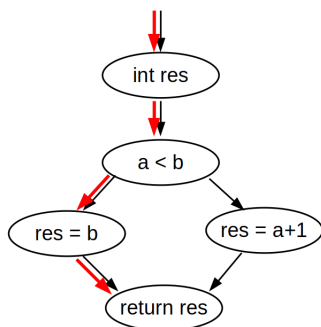
Ce développement a pour but de décrire les critères de test, en illustrant sur l'exemple du maximum ou on illustre les erreurs. On finit par une preuve de l'indécidabilité de la satisfiabilité de ces critères de test

Tous les noeuds On souhaite au moins vérifier chaque ligne de code

```
max(a, b):  
    int res;  
    if(a < b)  
        res = b;  
    else  
        res = a + 1;  
    return res
```

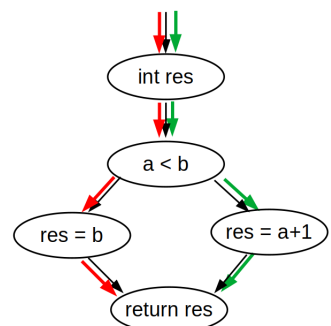


$DT = [(3, 5), 5]$



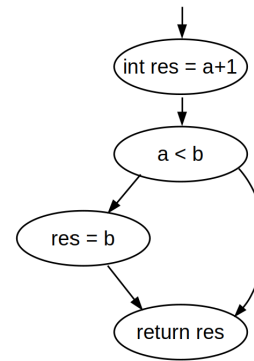
On ne détecte pas l'erreur.

On cherche alors à couvrir tous les noeuds $DT = [(3, 5), 5], [(5, 4), 5]$

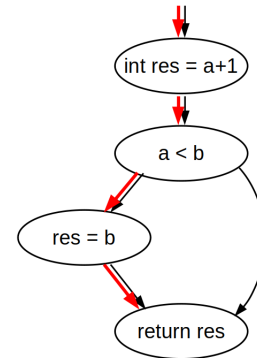


Tous les arcs Néanmoins, ce critère n'est pas suffisant.

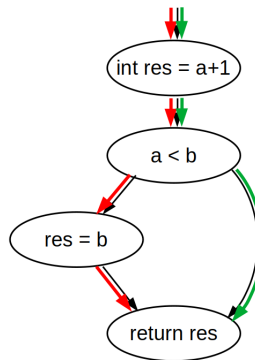
```
max(a, b):
  int res = a + 1;
  if (a < b)
    res = b;
  return res;
```



$DT = [(3, 5), 5]$ couvre tous les noeuds mais ne détecte pas l'erreur



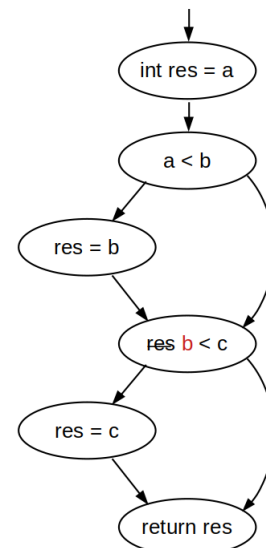
$DT = [(3, 5), 5], [(5, 4), 5]$



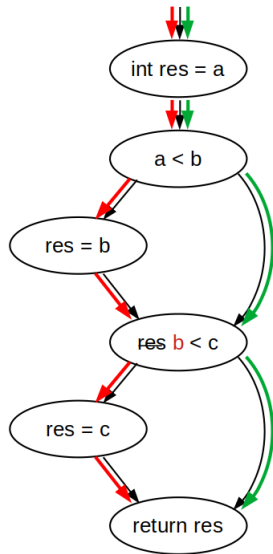
détecte tous les arcs et donc l'erreur.

Critère tous les chemins Malheureusement ce n'est pas suffisant.

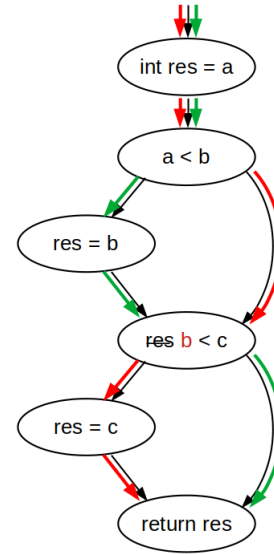
```
max(a, b, c):
  int res = a;
  if (res < b)
    res = b
  if (res < c)
    res = c
  return res
```



$DT = [((3, 4, 5), 5), ((5, 4, 3), 5)]$ respecte le critère tous les arcs mais ne détecte pas l'erreur



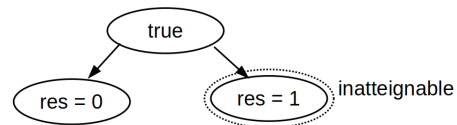
On rajoute alors le critère tous les chemin en ajoutant $[((5, 3, 4), 5), ((4, 5, 3), 5)]$ et le premier renvoie 4



Insuffisance $DT = [((3, 4, 5), 5), ((5, 4, 3), 5), ((4, 5, 3), 5), ((4, 3, 5), 5)]$ Avec ça on teste tous les chemins mais on ne détecte pas le problème.

Insatisfiabilité En plus de cela, les critères ne sont pas toujours satisfiables.

```
if(true):
    res = 0
else
    res = 1
```



Proposition 17.1 *Savoir si un critère est satisfiable est indécidable.*

Démonstration. Pour cela, on va prouver que détecter du code mort est indécidable (ce qui revient à la satisfiabilité du critère tous les noeuds, qui implique les autres).

Procédons par l'absurde. Supposons que l'on connaisse un algorithme \mathcal{A} qui décide si du code d'un algorithme est mort. Décidons alors le théorème de l'arrêt.

Construisons l'algorithme \mathcal{B} qui sur l'entrée \mathcal{C} , crée l'algorithme \mathcal{C} puis $\text{print}(0)$. On demande alors à \mathcal{A} si $\text{print}(0)$ est du code mort. La réponse nous dit alors si \mathcal{C} termine ou non. Ainsi, \mathcal{B} décide du problème de l'arrêt, pourtant indécidable. \square

On cherche alors plutôt à maximiser $\frac{\text{nombre d'arc visité}}{\text{nombre total d'arc}}$ (de même avec noeuds, chemins, etc...)

Développement 18

Equivalence entre l'impératif en C et le fonctionnel en Ocaml

Auteur·e·s: Emile Martinez

Références : mon cul

Ce développement présente comment faire du C sans fonction avec du Ocaml sans boucle, et réciproquement. Il peut donc servir à illustrer les paradigmes de programmation ou encore l'universalité de la notion de calculabilité

Fonctionnel : Ocaml sans boucle while

Impératif : C sans fonction

Commentaire 18.1 Remarque à moduler suivant le contexte de ce développement

★ Impératif vers fonctionnel

programme C \rightsquigarrow fonction : Etat de la mémoire \rightarrow Etat de la mémoire
 \uparrow
transforme un état de la mémoire en un autre

Exemple $x = 1$
 $y = x + 1 \rightsquigarrow (fun(x, y) \rightarrow (x * y, y)) \circ (fun(x, y) \rightarrow (x, x + 1)) \circ (fun(x, y) \rightarrow (1, y))$
 $x = x * y$

Tout se traduit naturellement. (Les tableaux vers des listes, les opérations arithmétiques tel quel, etc...). Il ne reste alors que les boucles while.

```
1 while(C){  
2   S;  
3 }
```

On suppose que C ne modifie pas l'état de la mémoire.

On procède alors par induction et que l'on a donc $C \rightsquigarrow f_C : \text{Etat de la mémoire} \rightarrow \text{bool}$
 $S \rightsquigarrow f_S : \text{Etat de la mémoire} \rightarrow \text{Etat de la mémoire}$

Alors

```
1 while(C)  
2 {  
3   S;  
4 }
```

 \rightsquigarrow

```
let rec while_S_C args =  
  if f_C args then while_S_C (f_S args)  
  else args
```

★ Fonctionnel vers Impératif

1. De OCaml vers C avec fonction

2. De C avec fonction vers C sans fonction

1. Comme on a le droit aux fonctions, c'est naturel, sauf :

- le polymorphisme : On suppose les fonctions non polymorphes
- l'ordre supérieur : c'est du sucre syntaxique.

Exemple :

<pre>(fun f x y -> if f x y then x else y) (fun x y -> x < y)</pre>	devient	<pre>fun x y -> if x < y then x else y</pre>
---	---------	--

- la curryfication : on décurryfie

2.

Commentaire 18.2 Dans cette partie là, on peut soit présenter la pile d'appels en parlant de saut, et en présentant comment elle fonctionne sans écrire le code. (Il faut alors adapter les annonces au dessus en fonction). Sinon on fait une version plus explicite (qui suit), mais qui n'est pas la vraie manière.

Pour simplifier, on suppose sans perte de généralité que tous les appels de fonctions sont de la forme

```
1 variable = f(liste variables sans appels de fonctions)
```

Idée de la transformation :

1. avoir une pile stockant dans l'ordre, les lignes de code devant être exécutées et la valeur des variables à ce moment là
2. tant que la pile est non vide, prendre le sommet, et exécuter la ligne indiquée avec l'environnement associé.
3. quand on exécute une ligne, calculer le nouvel environnement qu'elle produit, et ajouter sur la pile, les lignes qui doivent être alors exécutés, avec ce nouvel environnement.
4. on empile deux choses :
 - (a) le fait qu'il faudra reprendre le code là où on l'avait laissé
 - (b) la première ligne de la fonction avec comme environnement, ses paramètres avec des valeurs.
5. Une valeur spéciale pour gérer les valeurs de retour

Commentaire 18.3 Ici je mets ce qu'on doit dire ou écrire en parallèle de l'écriture de la liste au dessus.

<p>1.</p> <div style="border: 1px solid black; padding: 10px; display: inline-block;"> <pre>1 ... 33 ... 34 x = 1 + y; 35 ...</pre> </div>	deviendra	<pre>1 while(! est_vide(pile)){ 2 ligne, env = depiler(pile) 111 ... 112 if (ligne == 34){ 113 env[x] = 1 + env[y]; 114 empiler(pile, 35, env); 115 } 116 ... 301 }</pre>
--	-----------	---

2. On écrit ce que l'on vient d'expliquer.
- 3.

4. Pour un appel de fonctions, on a envie de dire qu'on va devoir exécuter deux lignes : celle de la fonction, puis après le reste du programme

```

1  ...
11 void f(){
12     ...
21 }
22 ...
53 f();
54 ...

```

deviendrait

```

1  while (! est_vide(pile)){
2      ligne , env = depiler(pile)
61  ...
62  if (ligne == 11){
63      // Executer le code de f
64  }
107  ...
108  if (ligne == 53){
109      empiler(pile , env , 54);
110      empiler(pile , env , 12);
111  }
112  ...
287 }

```

Les points 2 et 3 sont l'algorithme de transformation.

On associera à chaque variables un numéro. Un environnement devient alors seulement un tableau. On mettra également une valeur pour la valeur de retour d'une fonction.

Quand on fait un retour de fonction, on doit s'occuper de communiquer la valeur de retour à la variables d'après

Commentaire 18.4 Ce qui suit peut éventuellement être taper à l'ordi et projeter

```

1  int f(int x){
2      if(x==0)
3          return 1;
4      else{
5          int z = f(x-1);
6          return x*z;}
7  }
8  int y = f(4);

```

Tableau des variables (j'ai oublié le terme) :

ret	0
x	1
y	2
z	3

```

1  pile pile = pile_vide;
2  int env[5];
3  empiler(pile , env , 8);
4  while ! est_vide(pile):
5      env , ligne = depiler(pile)
6      if (ligne == 2){
7          if(env[1] == 0)
8              empiler(pile , env , 3);
9          else
10             empiler(pile , env , 5);
11     }
12     if(ligne == 3){
13         env2 , ligne2 = depiler(pile);
14         env2[0] = 1;
15         empiler(pile , env2 , ligne2);
16     }
17     if(ligne == 5){
18         empiler(pile , env , 5bis);
19         env[1] = env[1] - 1;
20         empiler(pile , env , 2);
21     }
22     if(ligne == 5bis){
23         env[3] = env[0];
24         empiler(pile , env , 6);
25     }
26     if(ligne == 6){
27         env2 , ligne2 = depiler(pile);
28         env2[0] = env[1]*env[3];
29         empiler(pile , env2 , ligne2);

```

```
30 }  
31 if(ligne == 8){  
32     empiler(pile , env , 8bis);  
33     env[1] = 4;  
34     empiler(pile , env , 2);  
35 }  
36 if(ligne==8bis){  
37     env[2] = env[0]  
38     empiler(pile , env , 9);  
39 }
```


Développement 19

Correction du tri fusion

Auteur·e·s: Emile Martinez

Références :

Ce développement a pour but de prouver la terminaison et la correction rigoureuse du tri fusion. Il va donc à priori dans les leçons sur les tris, sur diviser pour régner et sur la correction de programme. En début de leçon est rappelé le code de tri_fusion, mais qui a vocation à être dans le cours, et pas réécrit au tableau

Algorithme 19.1 : fusion(L_1, L_2)

```
res ← []
i, j ← 0
tant que  $i < |L_1|$  et  $j < |L_2|$  faire
    si  $L_1[i] < L_2[j]$  alors
        res.ajouter( $L_1[i]$ )
         $i \leftarrow i + 1$ 
    sinon
        res.ajouter( $L_2[j]$ )
         $j \leftarrow j + 1$ 
Ajouter le reste de  $L_1$  et de  $L_2$  à res
retourner res
```

Algorithme 19.2 : tri_fusion(L)

```
 $n \leftarrow |L|$ 
si  $n \leq 1$  alors
    retourner  $L$ 
 $L_1, L_2 \leftarrow \text{partitionner}(L)$ 
retourner fusion( tri_fusion( $L_1$ ), tri_fusion( $L_2$ ) )
```

Terminaison de fusion Prenons comme invariant $|L_1| - i + |L_2| - j$.

C'est bien un entier, positif (Car si $|L_1| > i$, $i + 1 \leq |L_1|$ donc $|L_1| - (i + 1) \geq 0$), qui décroît strictement à chaque itération : en effet

Soit $i \leftarrow i + 1$ donc $|L_1| - i \leftarrow |L_1| - i - 1$

Soit $j \leftarrow j + 1$ donc $|L_2| - j \leftarrow |L_2| - j - 1$

Donc fusion termine.

Terminaison de tri_fusion On prend comme variant $|L| \in \mathbb{N}$.

Les seuls appels récursifs à tri_fusion se font sur des listes de taille strictement inférieures.

Donc il n'y a qu'un nombre fini d'appels récursifs.

Or fusion termine, donc chaque appel recursif termine. Donc tri_fusion termine.

Correction partielle de fusion Spécification de fusion : Si L_1 et L_2 sont triés, alors $\text{fusion}(L_1, L_2)$ est $L_1 \sqcup L_2$ trié.

Prenons comme invariant \mathcal{P} : $\ll res = L_1[:i] \sqcup L_2[:j]$ trié et $L_1[i]$ et $L_2[j]$ sont plus grands que les éléments de res . \gg

★ Alors avant la boucle, on a bien $i = 0, j = 0$ donc $res = [] = [] \sqcup []$ trié

★ Supposons \mathcal{P} au début de la boucle. Alors

★★ Si $L_1[i] \leq L_2[j]$

Alors on ajoute à res $L_1[i]$. Donc par \mathcal{P} , res est trié

Et comme L_1 est trié, $L_1[i+1] \geq L_1[i]$ et $L_2[j] \geq L_1[i]$. Donc par \mathcal{P} , $L_1[i+1]$ et $L_2[j]$ sont plus grands que tous les éléments de res . Donc quand $i \leftarrow i+1$, on obtient bien le résultat.

★★ L'autre cas est symétrique.

Donc \mathcal{P} est bien un invariant.

Ainsi, à la fin \mathcal{P} est vrai et comme la condition d'arrêt est à faux, on a $i = |L_1|$ ou $j = |L_2|$. Donc par \mathcal{P} , une des deux listes est totalement dans res , et il ne manque à l'autre que ses plus grands éléments, tous plus grands que ceux de res (ça c'est par \mathcal{P} et car les listes d'entrées étaient triées).

A la fin de fusion, on a donc $res = L_1 \sqcup L_2$ trié.

Donc fusion est partiellement correcte

Correction partielle de tri_fusion Soit \mathcal{P} la propriété définie pour $n \in \mathbb{N}^*$ par $\mathcal{P}(n)$: $\ll \text{tri_fusion}(L)$ trie L pour toute liste de taille n . \gg

Soit $n \in \mathbb{N}^*$ tel que $\forall k \in \mathbb{N}, \mathcal{P}(k)$. Soit L une liste de taille n .

★ Si $n = 0$ ou $n = 1$ alors $\text{tri_fusion}(L) = L$ qui est donc L trié.

★ Sinon par $\mathcal{P}(\lfloor \frac{n}{2} \rfloor)$ et $\mathcal{P}(\lceil \frac{n}{2} \rceil)$, on a que $\text{tri_fusion}(L_1)$ (resp. $\text{tri_fusion}(L_2)$) vaut L_1 (resp. L_2) trié. Donc L_1 et L_2 vérifient les préconditions de fusion. Donc $\text{fusion}(\text{tri_fusion}(L_1), \text{tri_fusion}(L_2))$ vaut $L_1 \sqcup L_2$ trié. Or L_1 et L_2 partitionnent L (leur union disjointes contient donc les mêmes éléments que L). Donc $\text{tri_fusion}(L)$ renvoie L trié.

Ainsi par principe de récurrence, tri_fusion est partiellement correcte.

Conclusion tri_fusion est correcte.

Développement 20

Equivalence entre expression booléenne et fonction booléenne

Auteur·e·s: Emile Martinez

Références :

à compléter

Les expressions booléennes sont aussi puissantes que les fonctions. (\Leftarrow est immédiat).

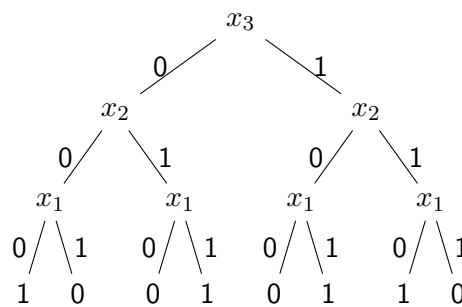
Notation On note $f \simeq e$ pour dire qu'une fonction $f : \{0, 1\}^n$ est équivalente à une expression booléenne e , i.e. $\forall \sigma : V \rightarrow \{0, 1\}, [e]_\sigma = f(\sigma(x_1), \sigma(x_2), \dots, \sigma(x_n))$

\Rightarrow On prend $f : \{0, 1\}^n \rightarrow \{0, 1\}$.

Idée On fait la table de vérité de f , et on fait une disjonction des cas sur la valeur de chaque argument.

Exemple $Pair : \{0, 1\}^3 \rightarrow \{0, 1\}$ qui vaut 1 si on a un nombre pair de 1

x_3	x_2	x_1	$Pair$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0



$$\begin{aligned} & \overline{x_3} \wedge ((\overline{x_2} \wedge \overline{x_1}) \vee (x_2 \wedge x_1)) \\ & \vee x_3 \wedge ((x_2 \wedge \overline{x_1}) \vee (\overline{x_2} \wedge x_1)) \end{aligned}$$

Commentaire 20.1 On peut ne pas tout étiqueter dans l'arbre, et quand on construit la formule, on explique que on ne garde que on ne veut garder que les branches qui finissent par 1. Que pour cela, soit on a x_3 qui est faux et on est à gauche dans l'arbre, soit x_3 est vrai est on est à droite dans l'arbre, etc...

Démonstration. Soit \mathcal{P} la propriété définie sur \mathbb{N}^* par $\mathcal{P}(n) : \ll \forall f : \{0, 1\}^n \rightarrow \{0, 1\}, \exists e \in EB : f \simeq e \gg$

* Soit $f : \{0, 1\} \rightarrow \{0, 1\}$.

On prend alors $e = \perp$

$$\wedge x_1 \text{ si } f(1) = 1$$

$$\wedge \overline{x_1} \text{ si } f(0) = 1$$

$$[e]_\sigma = 1 \Leftrightarrow \text{ou } \begin{matrix} \sigma(x_1) = 1 \text{ et } f(1) = 1 \\ \sigma(x_1) = 0 \text{ et } f(0) = 1 \end{matrix} \Leftrightarrow \text{ou } \begin{matrix} \sigma(x_1) = 1 \text{ et } f(\sigma(x_1)) = 1 \\ \sigma(x_1) = 0 \text{ et } f(\sigma(x_1)) = 1 \end{matrix} \Leftrightarrow f(\sigma(x_1)) = 1$$

Donc $\mathcal{P}(1)$

★ Soit $n \in \mathbb{N}^*$ tel que $\mathcal{P}(n)$. Soit $f : \{0, 1\}^{n+1} \rightarrow \{0, 1\}$

$$\text{Alors définissons } f_0 : \begin{matrix} \{0, 1\}^n & \rightarrow & \{0, 1\} \\ (b_1, \dots, b_n) & \mapsto & f(b_1, \dots, b_n, 0) \end{matrix} \quad \text{et } f_1 : \begin{matrix} \{0, 1\}^n & \rightarrow & \{0, 1\} \\ (b_1, \dots, b_n) & \mapsto & f(b_1, \dots, b_n, 1) \end{matrix}$$

On applique alors $\mathcal{P}(n)$ à f_0 et à f_1 pour obtenir $f_0 \simeq e_0$ et $f_1 \simeq e_1$.

On pose alors $e = (x_{n+1} \wedge e_1) \vee (\overline{x_{n+1}} \wedge e_0)$

$$[e]_\sigma = 1 \Leftrightarrow \text{ou } \begin{matrix} \sigma(x_{n+1}) = 1 \text{ et } [e_1]_\sigma = 1 \\ \sigma(x_{n+1}) = 0 \text{ et } [e_0]_\sigma = 1 \end{matrix}$$

$$\Leftrightarrow \text{ou } \begin{matrix} \sigma(x_{n+1}) = 1 \text{ et } f_1(\sigma(x_1), \sigma(x_2), \dots, \sigma(x_n)) = 1 \\ \sigma(x_{n+1}) = 0 \text{ et } f_0(\sigma(x_1), \sigma(x_2), \dots, \sigma(x_n)) = 1 \end{matrix}$$

$$\Leftrightarrow \text{ou } \begin{matrix} \sigma(x_{n+1}) = 1 \text{ et } f_1(\sigma(x_1), \sigma(x_2), \dots, \sigma(x_n), 1) = 1 \\ \sigma(x_{n+1}) = 0 \text{ et } f(\sigma(x_1), \sigma(x_2), \dots, \sigma(x_n), 0) = 1 \end{matrix}$$

$$\Leftrightarrow \text{ou } \begin{matrix} \sigma(x_{n+1}) = 1 \text{ et } f_1(\sigma(x_1), \sigma(x_2), \dots, \sigma(x_n), \sigma(x_{n+1})) = 1 \\ \sigma(x_{n+1}) = 0 \text{ et } f(\sigma(x_1), \sigma(x_2), \dots, \sigma(x_n), \sigma(x_{n+1})) = 1 \end{matrix}$$

$$\Leftrightarrow f(\sigma(x_1), \sigma(x_2), \dots, \sigma(x_n), \sigma(x_{n+1})) = 1$$

Ainsi, on obtient $\mathcal{P}(n+1)$

Par principe de récurrence, on obtient $\forall n \in \mathbb{N}^*, \mathcal{P}(n)$. □

Remarque Si on développe en utilisant la distributivité, on obtient la disjonction des conjonction qui mettent la formule à vrai. C'est ce que l'on appelle la forme normale disjonctive (FND) :

$$\bigvee_{(b_1, \dots, b_n) \in f^{-1}(\{1\})} \delta_{b_1}(x_1) \wedge \dots \wedge \delta_{b_n}(x_n) \text{ avec } \delta_1(y) = y \text{ et } \delta_0(y) = \overline{y}$$

Remarque Notre preuve est constructif (bien) mais exponentiel (mal) (la taille T_n vérifie $T_n = 2 + 2 \times T_{n-1}$). Souvent, cette explosion est inévitable. Parfois elle l'est.

Exemple Pour $\max : \{0, 1\}^n \rightarrow \{0, 1\}$

Commentaire 20.2 Commencer à faire l'arbre avec des pointillés en montrant que toutes les feuilles du sous arbre droit vaudront 1 et donc il faudra toutes les prendre, et y en a 2^{n-1} .

On a une expression exponentielle alors que l'on pourrait avoir $x_1 \vee \dots \vee w_n$.

Optimisation possible

En fusionnant des sous arbres Idée : Si les deux sous arbres sont les mêmes, c'est que le choix sur la variable ne sert à rien.

Commentaire 20.3 *Faire l'exemple sur \max_3 et montrer comment on arrive à fusionner les sous arbres*

En prenant la négation On peut également essayer de prendre la négation de la formule, si on a beaucoup de 1 et peu de zeros.

Développement 21

Explication de la pile d'appel

Auteur·e·s: Emile Martinez

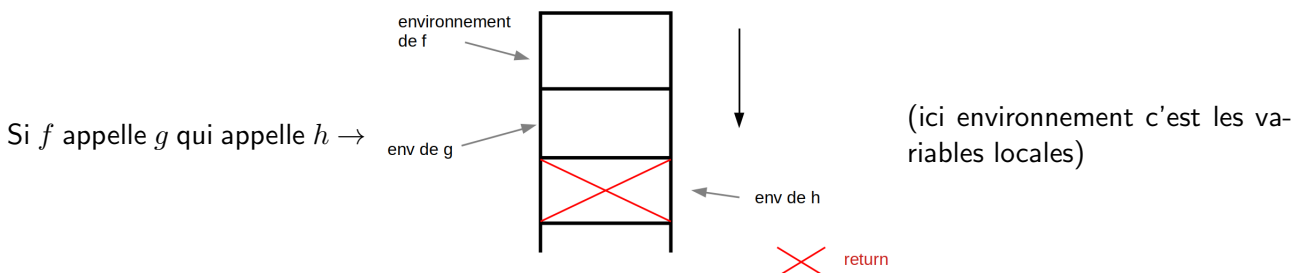
Références :

A compléter

Commentaire 21.1 Ici, suivant le temps, à voir si on met un rappel de à quoi sert la pile et à quoi sert le tas. Plutôt non et faire un schéma qui explique ce que fait le code assembleur à la fin.

But Stocker ce qui est local (donc pas les malloc).

Idée Avoir une pile.



Lors d'un appel de fonction :

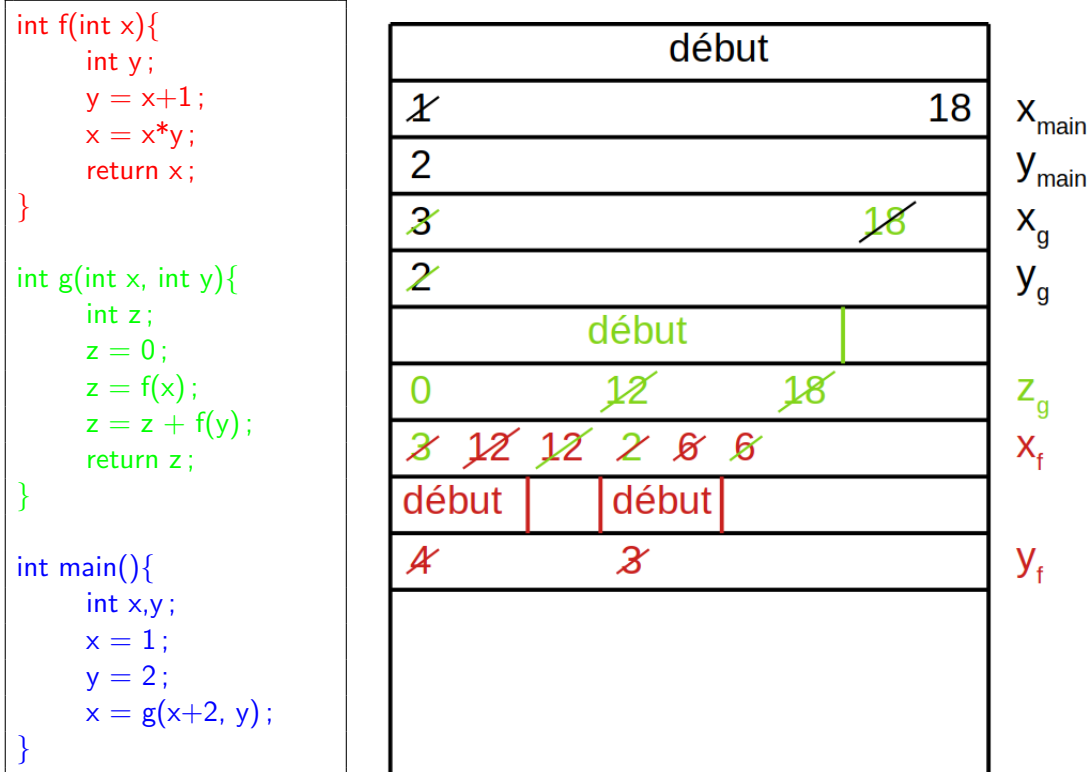
- ✧ On empile la valeur des arguments de la fonction
- ✧ On décale le début de la pile
 - \rightarrow Car on veut que l'appel à une fonction ne dépendent pas du niveau d'imbrication des appels
- ✧ On alloue sur le sommet de la pile de l'espace pour les variables définies localement.

Commentaire 21.2 Là, on peut commencer l'exemple et revenir écrire le retour de fonction après.

Lors d'un return :

- ✧ Calculer la valeur de retour
- ✧ Enlever de la pile toutes nos variables
- ✧ Remettre la pile à son début précédent
- ✧ Ecrire sur le sommet la valeur de retour
- ✧ Le code appelant lis la valeur sur le sommet

Exemple d'exécution de code



Comment faire ça en assembleur ? On se réserve deux registres, que le code ne touchera pas : b_p (pour base pointer, la base de la pile) et s_p (pour stack pointer, le sommet de la pile).

Idée b_p pointe vers «début» et «début» stocke le b_p précédent. s_p pointe vers le sommet de la pile.

$y = f(x_1, \dots, x_n)$

→

```

// mettre dans r1 la valeur de x1
store r1 [sp]
iadd sp sp 1
...
// mettre dans r1 la valeur de xn
store r1 [sp]
iadd sp sp 1

call f // va au code de f
//mettre dans r1 l'adresse de y
iadd sp sp -1
load r2 [sp]
store r2 [r1]

```

```
int f(x1, ..., xn){  
    int y1, ..., ym;  
    //code de f  
    return yi;  
}
```

→

```
store bp [sp]  
mv sp bp  
iadd sp sp 1  
  
iadd sp sp m;  
  
// code de f  
  
iadd r1 bp (i+1)  
load r2 [r1]  
iadd sp bp -n  
load bp [bp]  
store r2 [sp]  
ret //revient au code appelant
```


Développement 22

Passage d'une expression rationnelle à un automate

Auteur·e·s: Daphné Kany

Références :

Ce développement présente les constructions de Thompson pour passer d'une expression régulière à un NFA avec ϵ transition. On peut ensuite parler de détermination d'un automate, et de complexité. Il s'insère dans la leçon 30 et 34.

Théorème 22.1 Soit e une expression régulière. Il existe A un DFA tq $L(e) = L(A)$.


Démonstration. La preuve se fait en deux temps :

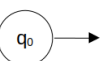
1. Il existe un NFA A_N tq $L(A_N) = L(e)$.
2. Pour tout NFA A_N , il existe un DFA A tq $L(A_N) = L(A)$.

Commentaire 22.1 Les deux preuves sont constructives et donnent donc un algorithme pour trouver A .

1. On raisonne par induction sur e :

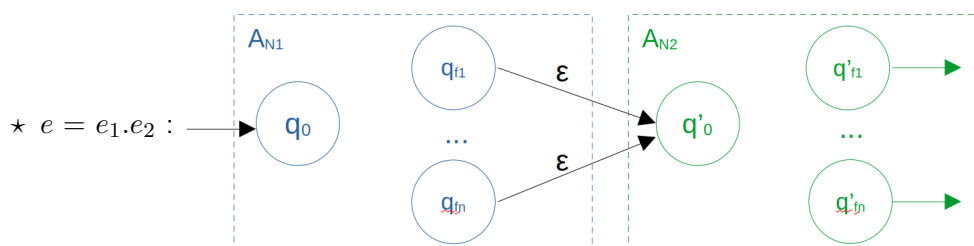
Cas de bases :

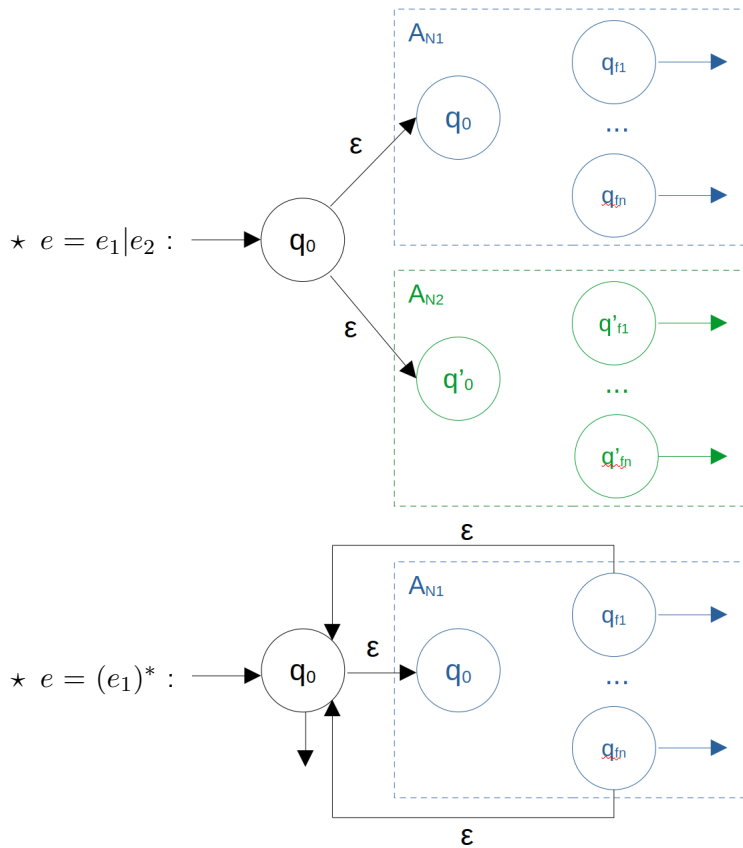
★ $e = \emptyset$. Alors A_N convient : 

★ $e = \epsilon$ Alors A_N convient : 

★ $e = a \in \Sigma$ Alors A_N convient : 

Soit e_1 et e_2 deux regexp, A_{N_1} et A_{N_2} deux NFA tq $L(A_{N_1}) = L(e_1)$ et $L(A_{N_2}) = L(e_2)$.

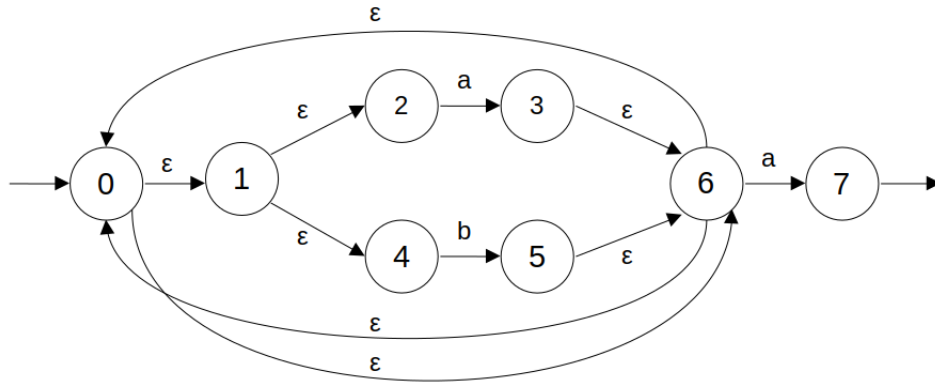




Commentaire 22.2 On peut faire à l'oral plus en détail les deux inclusions sur un des cas.

Exemple 22.1 $e = (a|b)^*.a$

Remarque : e est l'ensemble des mots finissant par a .



2. Pour tout NFA A_N , il existe un DFA A_D tq $L(A_N) = L(A_D)$.

Soit $A_N = (\Sigma, Q_N, q_{N0}, F_N, \delta_N : Q_N \times \Sigma \cup \{\epsilon\} \rightarrow P(Q_N))$.

On définit $A_D = (\Sigma, Q_D, q_{D0}, F_D, \delta_D : Q_D \times \Sigma \rightarrow Q_D)$ par :

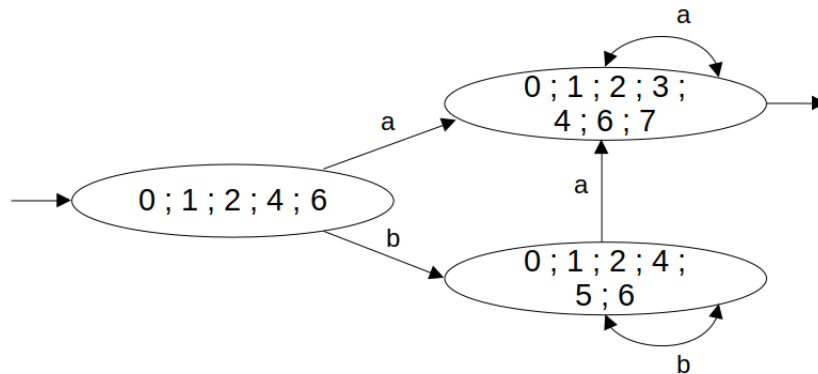
- ★ $Q_D = P(Q_N)$. Remarque : $|Q_D| = 2^{|Q_N|}$
- ★ $q_{D0} = E(\{q_{N0}\})$ où $E(q_D) = \{\text{états accessible depuis } q_D \text{ avec 0 ou plus } \epsilon\text{-transitions}\}$ (epsilon fermeture)
- ★ $F_D = \{q \in P(Q) | q \cap F_N \neq \emptyset\}$
- ★ $\delta_D(q_D, a) = E(\bigcup_{q_N \in q_D} \delta_N(q_N, a))$

Démonstration. On montre par récurrence

P_i : pour tout mot v de taille i , $\delta_D(q_{D0}, v) = \delta_N(q_{N0}, v)$

En particulier, v reconnu dans A_D ssi $\delta_D(q_{D_0}, v) \in F_D$ ssi $\delta_N(q_{N_0}, v) \cap F_N \neq \emptyset$ ssi v reconnu dans A_N . \square

Exemple 22.2 Sur le NFA précédent :



Commentaire 22.3 En construisant le DFA à la volée, on ne construit que les états accessibles qui peuvent être bien moins nombreux.

L'exemple montre également que l'automate obtenu n'est pas minimal (ici on aurait pu fusionner l'état initial et l'état du bas).

Enfin, on peut dire qu'il existe des NFA pour lesquels la taille d'un DFA minimal équivalent est exponentielle (langage dont la $n-1$ eme dernière lettre est a).

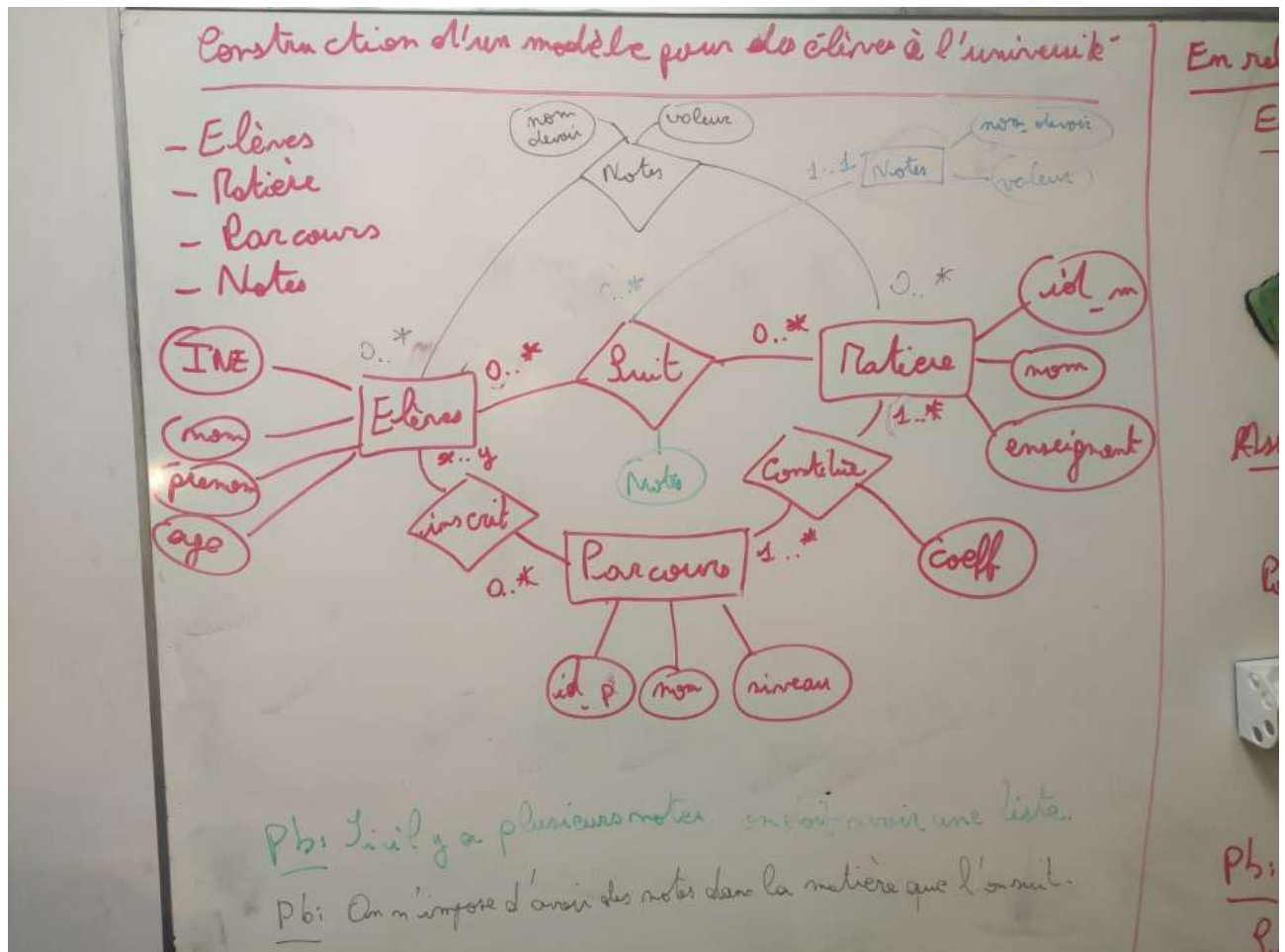
\square

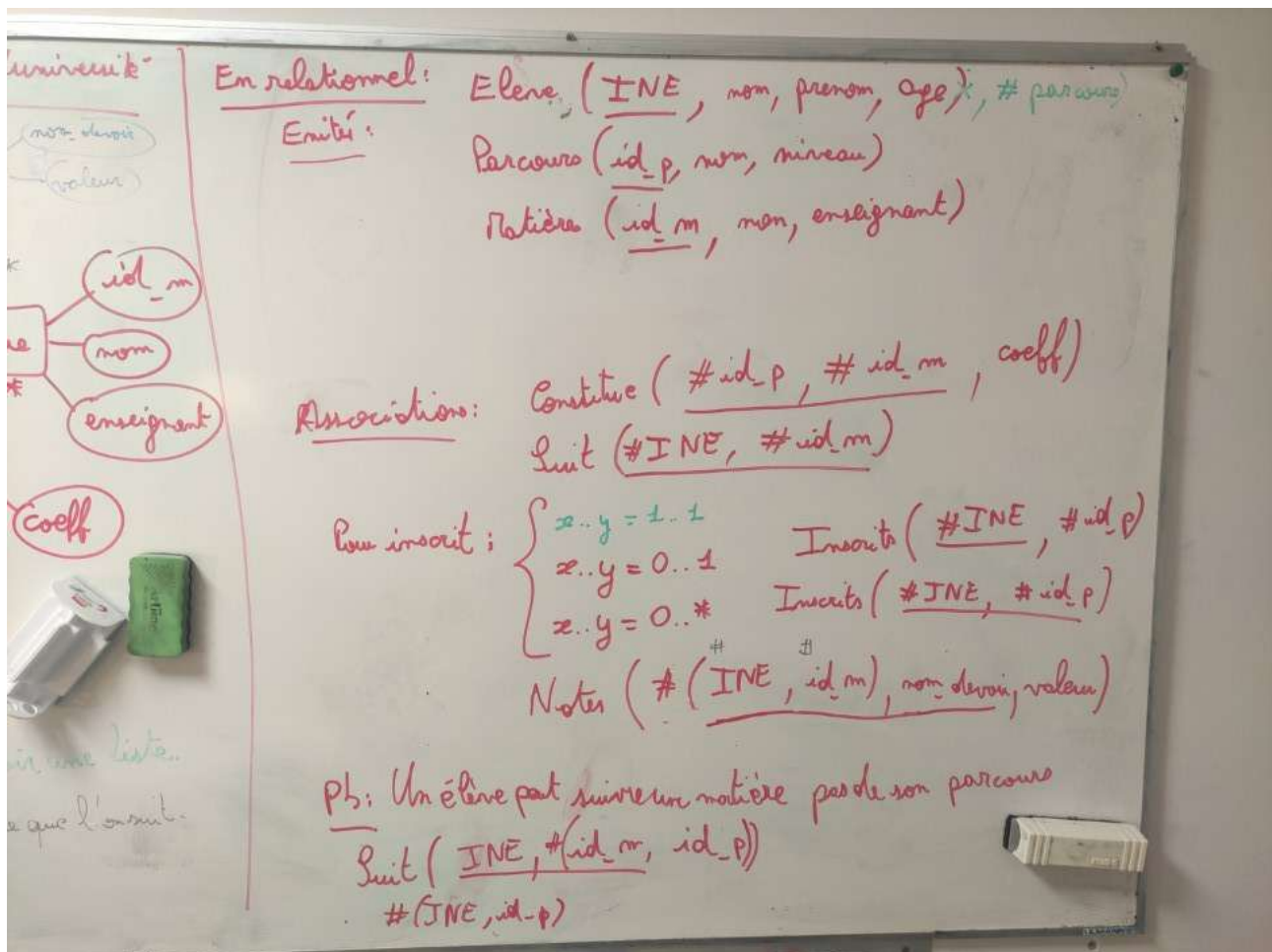
Développement 23

Construction d'une base de données relationnelles d'élèves à l'université

Auteur·e·s: Emile Martinez

Références : Balabonski MP11, exercice 200 et 202 pages 739-740, mais avec des trucs à rajouter





Développement 24

Présentation d'un algorithme d'analyse syntaxique descendant par retour sur trace

Auteur·e·s: Emile Martinez

Références : Legendre

Ce développement présente vérifiant l'appartenance d'un mot à un langage reconnu par une grammaire. Cet algorithme fonctionnant par retour sur trace (avec plein de subtilités du retour sur trace dedans), il s'insère donc complètement dans les leçons sur l'analyse lexicale et syntaxique, et dans la leçon sur glouton et retour sur trace.

Notation Notons $v \rightarrow_l^* u$ pour v se dérive en u en moins de l étapes.

Commentaire 24.1 D'abord l'exemple ou d'abord l'algorithme ? Ou dans quelle mesure on arrive à faire les deux en même temps.

Algorithme 24.1 : retour_sur_trace(G, u, v, l)

Données : grammaire G , u mot d'entrée, v mot en cours, l longueur restante de dérivation autorisée

Sorties : VRAI ssi $v \rightarrow_l^* u$

si $l < 0$ **alors**

retourner FAUX

si v est vide **alors**

retourner u est vide

$a, v_2 \leftarrow \text{depiler}(v)$

si $a \in \Sigma$ **alors** // a est un terminal

$b, u_2 \leftarrow \text{depiler}(u)$ // Si u est vide, retourner Faux

si $a \neq b$ **alors**

retourner FAUX

sinon

retourner retour_sur_trace(G, u_2, v_2, l)

sinon // a est un non terminal

pour chaque règle $a \rightarrow x_1 \dots x_n$ de G **faire**

$v_3 \leftarrow \text{empiler}(x_1 \dots x_n, v_2)$

si Backtrack($G, u, v_3, l - 1$) **alors**

retourner VRAI

retourner FAUX

Il suffit alors d'appeler `retour_sur_trace(\mathcal{G} , u , S , l_{max})` pour déterminer si $u \in \mathcal{L}(\mathcal{G})$ où S est l'axiome et l_{max} une borne supérieure sur la longueur minimale d'une dérivation de u si elle existe.

Illustration Soit \mathcal{G} :

$$\begin{array}{l} S \rightarrow aSb \mid aTb \\ T \rightarrow c \mid Tc \end{array}$$

On alors $\mathcal{L}(\mathcal{G}) = \{a^n c^m b^n \mid m, n > 0\}$

Comment faire pour reconnaître savoir si $u = aacbb \in \mathcal{L}(\mathcal{G})$:

u	v
$aacbb$	S
$\cancel{a}acbb$	$\cancel{a}Sb$
$\cancel{a}\cancel{c}bb$	$\cancel{a}S\cancel{b}b$
$\cancel{a}\cancel{c}\cancel{b}b$	$\cancel{a}S\cancel{b}\cancel{b}b$
$\cancel{a}\cancel{c}\cancel{b}\cancel{b}$	$\cancel{a}T\cancel{b}\cancel{b}b$
$\cancel{a}\cancel{c}bb$	$\cancel{a}Tbb$
$\cancel{a}\cancel{c}\cancel{b}b$	$\cancel{a}\cancel{c}\cancel{b}b$

Terminaison A-t-on la terminaison? Non si on inverse c et Tc dans \mathcal{G} . On ajoute alors une longueur maximale de dérivation.

Maintenant cela termine.

Variant : $|u| + l$

Correction On fait la correction par induction, en supposant la spécification de la fonction correcte sur tous les appels récursifs.

- ★ Pour les deux premières Si , on a bien le comportement attendu
- ★ Si a est un terminal, $\forall w \in (\Sigma \cup V)^*$, $v \rightarrow_l^* aw \Leftrightarrow v_2 \rightarrow_l^* w$ d'où le comportement correct.

Commentaire 24.2 Dire à l'oral que cela veut dire que toutes dérivation est indépendante de a , et le mot final commencera donc toujours par a .

- ★ Sinon, comme on devra transformer a en quelque chose avec une des règles, on a bien $v \rightarrow_l^* u \Leftrightarrow \exists a \rightarrow x_1 \dots x_n \in G : x_1 \dots x_n v_2 \rightarrow_{l-1}^* u$

Commentaire 24.3 Expliquer à l'oral cette formule.

Comment trouver l_{max} On peut tout d'abord essayer d'en deviner un en fonction de la grammaire. Par exemple, si chaque application de règle ajoute un non terminal (ou si plus largement, après n applications de règles, tout mot aura au moins n caractères), on peut prendre $|u|$ pour l_{max}

Sinon on a des résultats généraux mais moins précis sur les grammaires :

Théorème 24.1 Soit G une grammaire et $m \in \Sigma^*$. Les deux assertions suivantes sont équivalentes :

- $m \in L(G)$
- il existe une dérivation reconnaissant m dont la longueur est inférieure à $a^{|m| \times r}$

où a est le nombre maximum de symbole à droite d'une règle, et r le nombre de non-terminaux.

Commentaire 24.4 *A part si on est beaucoup trop en avance, donnée l'idée du théorème ne me paraît pas nécessaire.*

Commentaire 24.5 *Les remarques suivantes ne sont pas nécessaires et sont à mettre en fonction du temps qu'il reste.*

Langage de programmation Ici, l'implémentation de cet algorithme serait spécifiquement pratique en Ocaml, en utilisant pour les piles le type `symbole list`. L'immutabilité des données nous arrangeant beaucoup. En prenant des `int` pour non terminaux, \mathcal{G} deviendrait alors `symbole list array`. Néanmoins, nous n'avons pas la concaténation en $O(1)$ comme en C.

Complexité C'est n'importe quoi. On peut faire mieux en :

- changeant la grammaire pour que chaque application de règle ajoute des éléments.
- Mémoisant les résultats
- faisant de la programmation dynamique (si on arrive à donner à une grammaire la bonne forme)

Développement 25

Présentation des arbres k-dimensionnel

Auteur·e·s: Emile Martinez

Références : Balabonski MPI

Ce développement présente la création et la recherche dans des arbres K -dimensionnels. Il s'insère donc dans la leçon sur les arbres, sur les algorithmes d'apprentissage (pour k -PPV) et dans la leçon diviser pour régner en insistant plus sur les notions de ce paradigme de ce développement.

Objectif : Prétraiter n points de \mathbb{R}^K pour que l'on puisse rapidement trouver les k plus proches voisins d'un point $x \in \mathbb{R}^K$

Construction

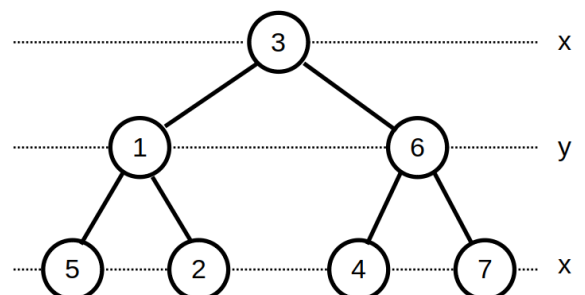
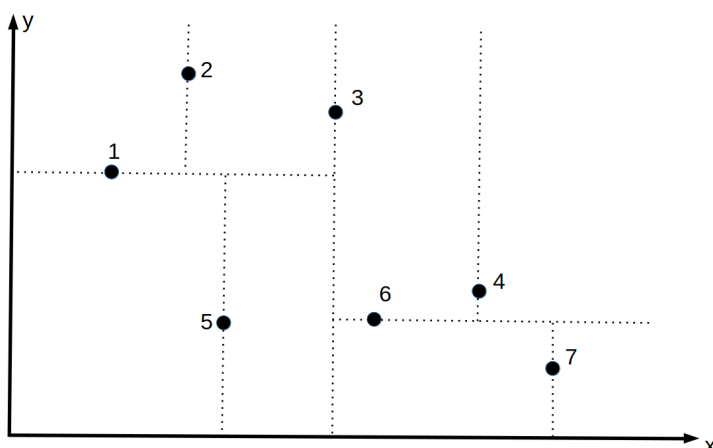
Idée : Un arbre K -dimensionnel de n points de \mathbb{R}^K est un arbre binaire de recherche sur K dimension :

- ★ chaque noeud contient un point
- ★ pour un noeud à la profondeur i contenant le point x ,
 - pour y contenu dans le sous arbre gauche de x , $y_{i \% K} \leq x_{i \% K}$
 - pour y contenu dans le sous arbre droit de x , $y_{i \% K} \geq x_{i \% K}$
- ★ les sous arbres gauche et droit d'un noeud ont à peu près la même taille.

construction : Si on est à la profondeur i :

- On prend la médiane selon la $i \% K$ -ème coordonnée
- On construit l'arbre enraciné à la profondeur $i + 1$ des points de $i \% K$ -ième coordonnée inférieur (resp. supérieur) à la médiane qui deviendra le sous arbre gauche (resp. droit) de notre médiane

Exemple en deux dimensions :



Recherche de k plus proches voisins

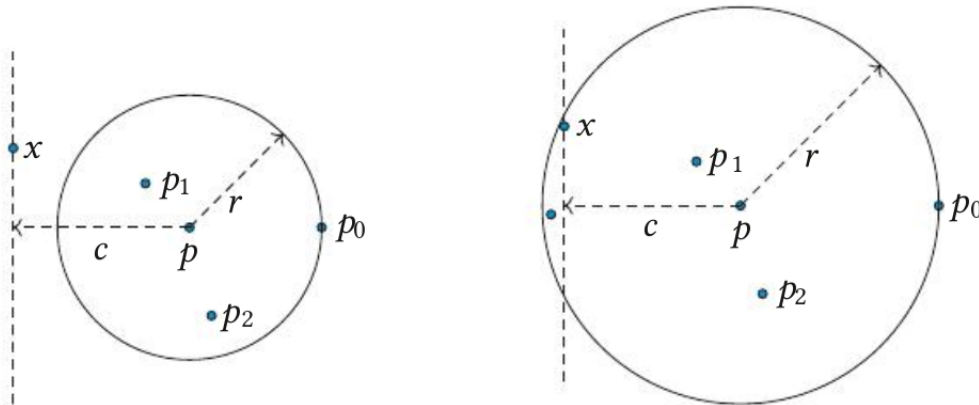
Idée : On cherche de préférence là où on aurait mis notre noeud

- ★ On cherche les k plus proches voisins dans le sous arbre où notre noeud aurait été inséré
- ★ Si les k que l'on a trouvé sont suffisamment loin pour que les plus proches puissent être dans un autre sous arbres, on remonte d'un cran et on cherche dans l'autre sous arbre, etc...

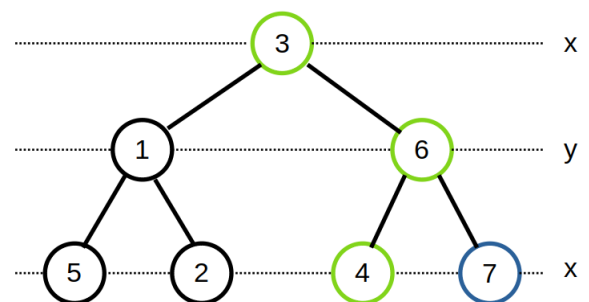
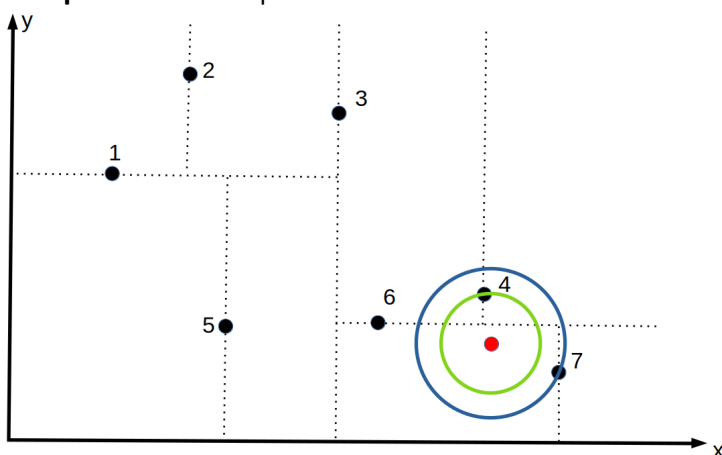
Algorithme Quand on cherche les voisins de p à la hauteur i

1. Si on a moins de K noeuds dans notre sous arbre, on les renvoie et on complète avec des noeuds à distance ∞
2. Sinon, on compare $p_i \% K$ à $x_i \% K$ où x est le point de notre noeud. Si c'est inférieur, on cherche dans le sous arbre gauche, sinon dans le droit
3. Quand on a récupéré p_1, \dots, p_k nos k voisins, on regarde si x est plus proche. Si oui, on l'ajoute
4. Notons r la distance au point le plus éloigné. Si $r > c = |p_i \% K - x_i \% K|$, on cherche dans l'autre sous arbre et on garde les k plus proches voisins des deux sous arbres.
5. On renvoie les voisins que l'on a.

Illustration Pour $K = 2$, un exemple du test de l'étape 4.



Exemple Sur l'exemple de la construction :



Complexité

Construction de l'arbre : $C(n) = C(\lceil (n-1)/2 \rceil) + C(\lfloor (n-1)/2 \rfloor) + \text{Mediane}(n)$

Donc si la médiane est fait en temps linéaire : $C(n) = O(n \log(n))$

Pour la recherche :

- dans le pire des cas : $O(kn)$
- dans le meilleur des cas : $O(k \log n)$
- dans le cas moyen : $O(k \log n)$ avec un adversaire

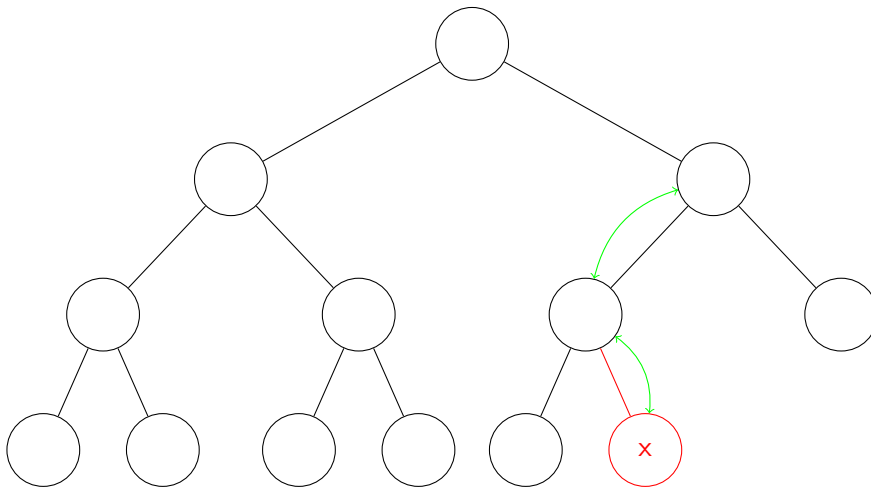
Développement 26

Correction de l'insertion dans un tas min

Auteur·e·s: Emile Martinez

Références :

Insertion d'un élément dans un tas min



Notation Pour s un sommet, on note $p(s)$ le père de s et \mathcal{A}_s l'arbre enraciné en s . Notons \mathcal{A} l'arbre global.

Algorithme 26.1 : Insertion de dans un tas min

Mettre x au seul endroit qui préserve la structure de tas

tant que $x < p(x)$ **faire**

 Echanger x et $p(x)$

Correction

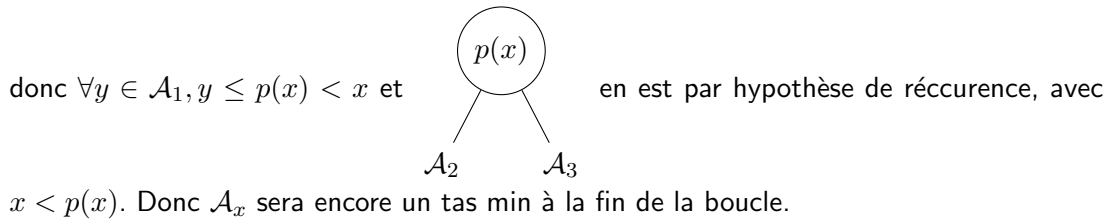
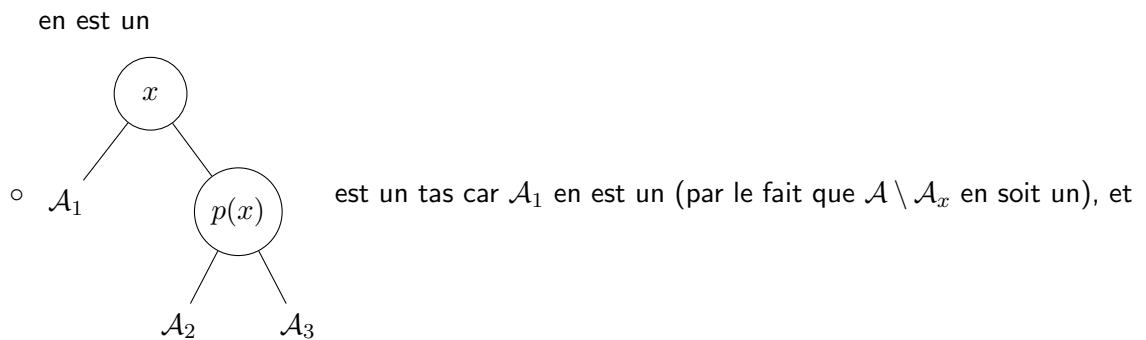
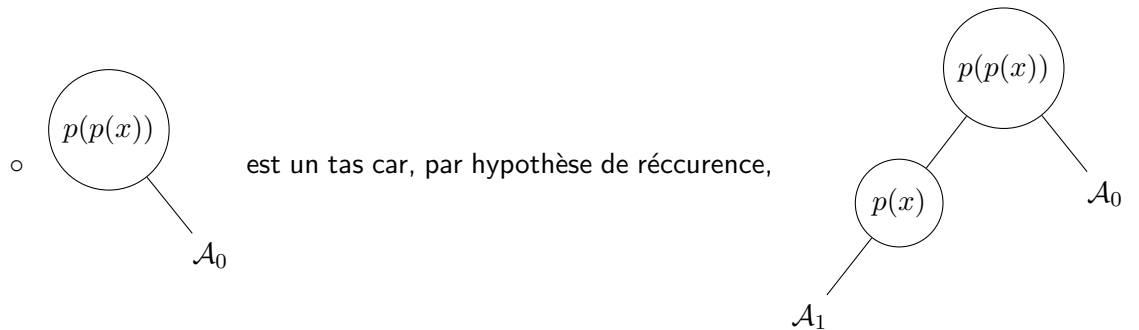
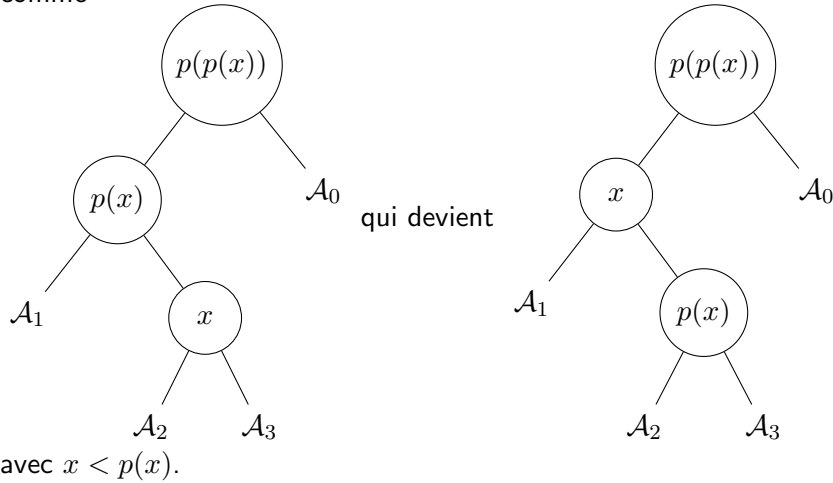
- La structure d'arbre presque complet est préservée par l'insertion et par les inversions
- Intéressons nous maintenant à la structure de tas.

On a alors l'invariant de boucle suivant :

- ◀ ◦ $\mathcal{A} \setminus \mathcal{A}_x$ est un tas
- \mathcal{A}_x est un tas
- $\mathcal{A}_x[x \leftarrow p(x)]$ est un tas. ▶

★ Avant la boucle l'invariant est vérifié, car \mathcal{A}_x ne contient qu'un élément et $\mathcal{A} \setminus \mathcal{A}_x$ est le tas min initial dans lequel on insère

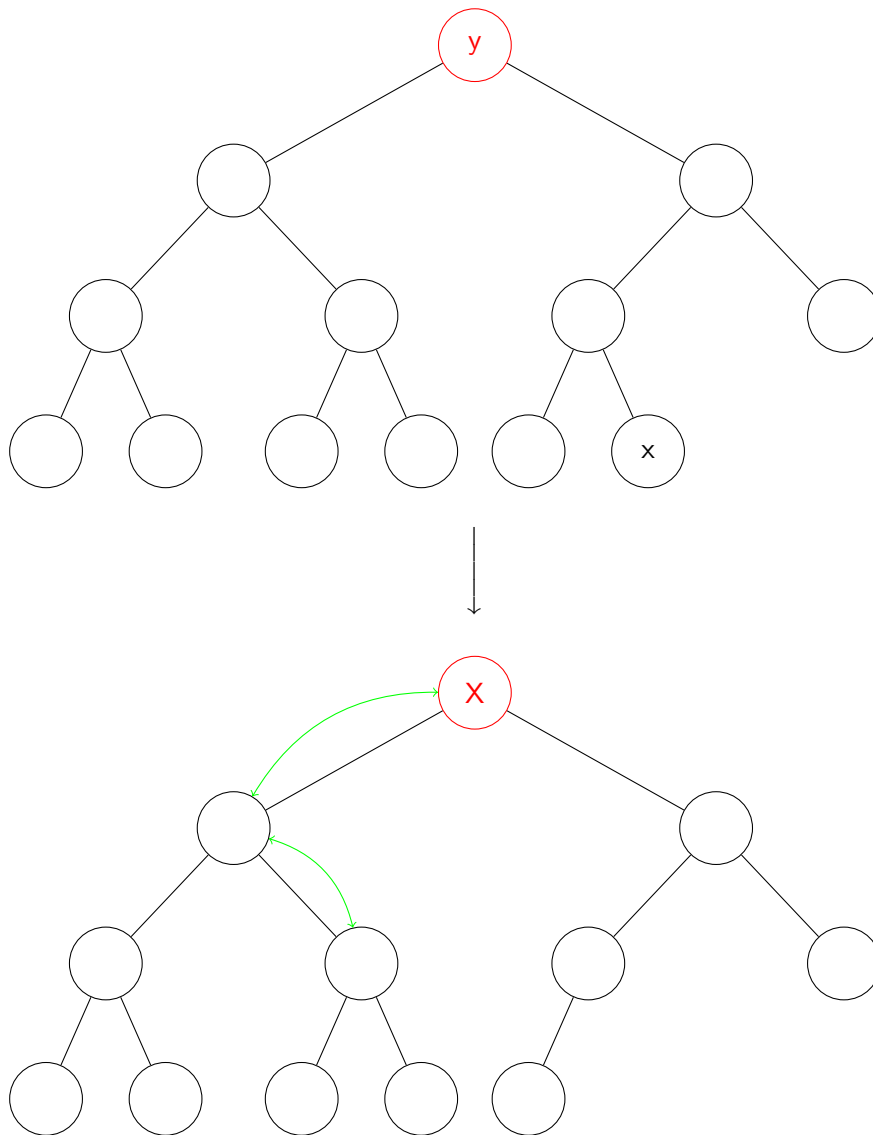
- ★ Supposons l'invariant vrai en début de boucle. Alors, on suppose que l'on a quelque chose comme



- $p(p(x)) < p(x)$ donc $\forall y \in \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3 \cup \{p(x)\}, y \leq p(x) \leq p(p(x))$. Donc, à la fin de la boucle, on aura encore $\mathcal{A}_x[x \leftarrow p(x)]$ qui sera un tas.

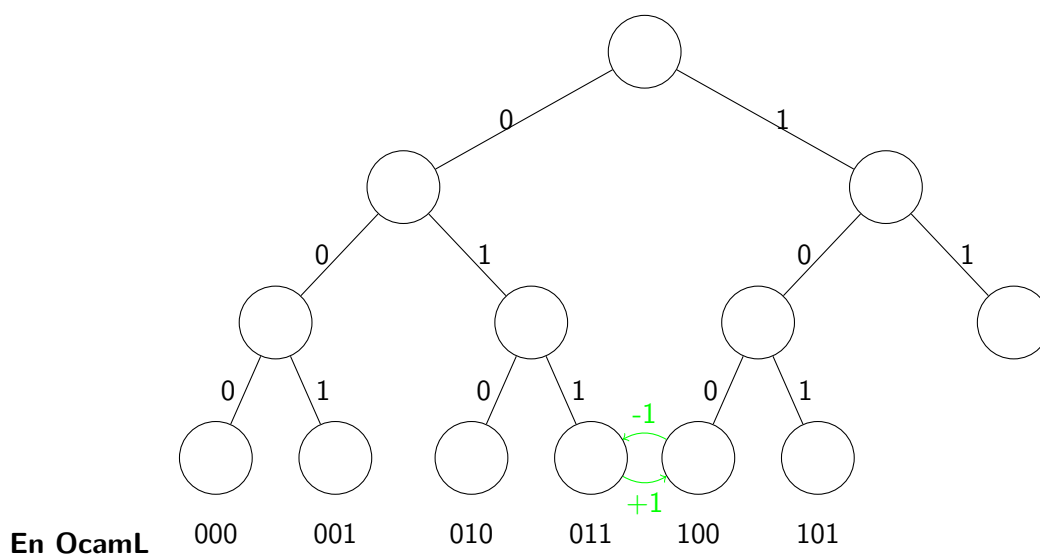
L'invariant de boucle est donc vérifié. Or, à la fin de l'algorithme, on a que $x \geq p(x)$, et on a que \mathcal{A}_x et $\mathcal{A} \setminus \mathcal{A}_x$ sont des tas. Donc \mathcal{A} est un tas.

Extraction de l'élément minimum



Commentaire 26.1 *La on parle pas de complexité. On peut en toucher un mot suivant le temps que cela prend*

Implémentation

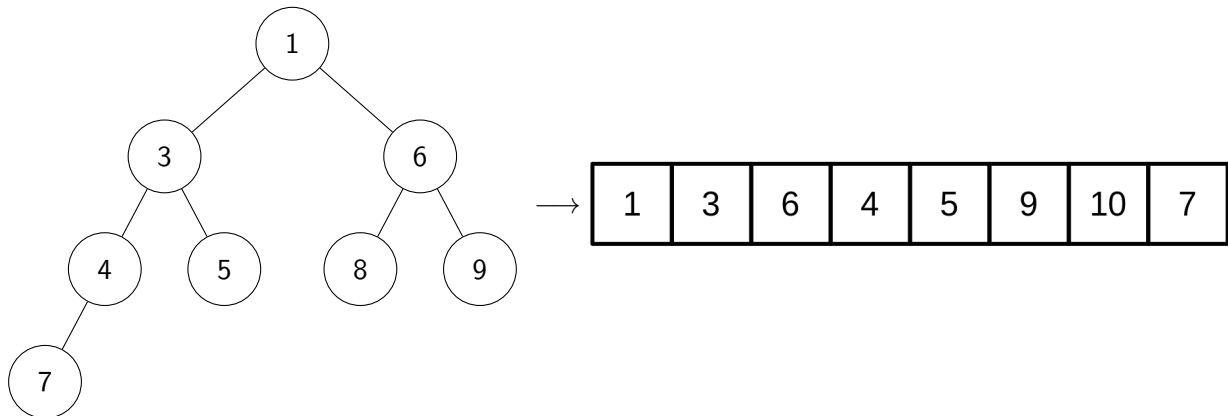


Idée : Une liste de 0 et de 1 représente un chemin dans l'arbre (avec 0 pour gauche, 1 pour droite). Pour passer à l'élément «suivant», il suffit de faire un $+1$ en considérant la liste comme un entier (Δ il faut traiter à part quand on ajoute passe à la ligne suivante)

```
type 'a bin = E | N of 'a * 'a bin * 'a bin;;  
type 'a file = {arbre : 'a bin; chemin : bool list };;
```

En C Idée : On stocke tout notre arbre dans un tableau. Alors le noeud à la case i a

- ses fils aux cases $2i+1$ et $2i+2$
- son père à la case $(i-1)/2$



Développement 27

2-SAT est résoluble en temps linéaire

Auteur·e·s: Emile Martinez, Malory Marin

Références :

Définition 27.1 (2-SAT)

Données :

- un ensemble de variables propositionnelles $X = \{x_1, \dots, x_n\}$
- une formule φ sous forme normale conjonctive où chaque clause est composée de 2 littéraux (un littéral étant une variable ou sa négation).

Problème : existe-t-il une valuation $v : \{x_1, \dots, x_n\} \rightarrow \{0, 1\}$ telle que $v(\varphi) = 1$?

Remarque 27.1 $y_1 \vee y_2 \equiv \overline{y_1} \rightarrow y_2 \equiv \overline{y_2} \rightarrow y_1$ en identifiant $\overline{\overline{x}}$ à x

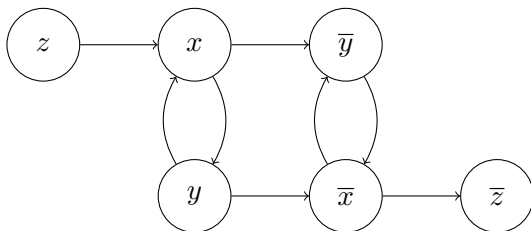
Associions à φ le graphe $G = (S, A)$

$$S = \{x_1, \dots, x_n, \overline{x_1}, \dots, \overline{x_n}\}$$

$$A = \{(y_1, y_2) \mid y_1, y_2 \in S, \varphi \text{ contient une clause équivalente à } y_1 \rightarrow y_2\}$$

Remarque 27.2 On remarque que le graphe est de taille linéaire en la taille de la formule.

Exemple 27.1 $\varphi = (x \vee \overline{y}) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y}) \wedge (x \vee \overline{z})$



Proposition 27.1 φ est satisfiable \Leftrightarrow pour tout $x \in \{x_1, \dots, x_n\}$, x et \overline{x} ne sont pas dans la même composante connexe.

Démonstration.

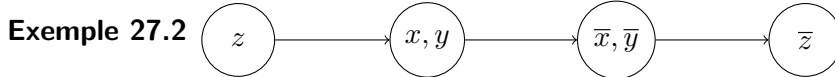
\Rightarrow Exercice

Commentaire 27.1 On ne le fait pas par manque de temps, mais on pourrait virer l'exemple par exemple et faire quand même ce sens.

⇐ Supposons que pour tout $x \in \{x_1, \dots, x_n\}$, x et \bar{x} ne sont pas dans la même composante connexe (i.e. la droite de \Leftrightarrow)

Idée On va construire une valuation pour φ .

Considérons le graphe G_c des composantes fortement connexes de G qui est orienté acyclique (DAG)



Lemme 27.1 Si C est un noeud de G_c , il existe \bar{C} un noeud de G_c tel que $\bar{C} = \{\bar{x}/x \in C\}$

Démonstration. Soit $u, v \in C$. Alors \bar{u}, \bar{v} dans la même composante connexe (car $u \rightarrow y \rightarrow \dots \rightarrow v \implies \bar{v} \rightarrow \dots \rightarrow \bar{y} \rightarrow \bar{u}$). Donc $\exists \bar{C} \in G_c : \{\bar{u}/u \in C\} \subset \bar{C}$ d'où le résultat par symétrie. \square

On procède alors par induction sur le nombre de sommets de G (sur le fait qu'il existe une valuation respectant G , donc φ)

- ★ Pour 0 une telle valuation (c'est une fonction de $\emptyset \rightarrow \{0, 1\}$)
- ★ G_c est un DAG donc $\exists P \in V_c : d_+(P) = 0$. P est donc un puit et \bar{P} une source. Par induction, on prend alors une valuation v pour $G \setminus P \cup \bar{P}$.
On étend alors v par pour $y \in P, v(y) = 1$
→ ceci est légale car pour $y \in P$ et $z \in P$, on a pas $\bar{y} = z$ (par hypothèse)

On a alors pour $y \in \bar{P}, v(y) = 0$

On a alors 5 types d'arêtes dans G :

1. les arêtes de \bar{P} dans \bar{P} qui sont respectées par v (car $0 \rightarrow 0$)
2. les arêtes de \bar{P} dans $S \setminus P \cup \bar{P}$ qui sont respectées (car faux implique tout)
3. les arêtes de $S \setminus P \cup \bar{P}$ dans $S \setminus P \cup \bar{P}$ qui sont respectées (par hypothèse d'induction)
4. les arêtes de $S \setminus P \cup \bar{P}$ dans P qui sont respectées car tout implique vrai.
5. les arêtes de P dans P qui sont respectées (car $1 \rightarrow 1$)

Donc v est une valuation respectant G . \square

Ainsi, pour résoudre 2-SAT il suffit de calculer les composantes fortement connexes du graphe associé à la formule grâce à l'algorithme de Kosaraju. On fait ensuite un tableau où on met l'identifiant de la composante connexe de chaque noeud et de son complémentaire, et on vérifie qu'ils n'ont pas la même.

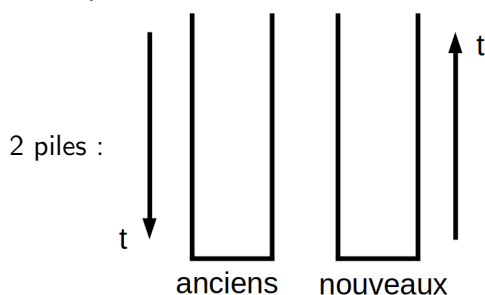
Développement 28

Implementation d'une file PAPS avec des piles

Auteur·e·s: Emile Martinez

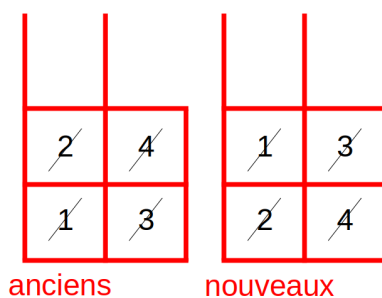
Références :

A compléter



Exemple

Sur l'exécution suivante :
Enfiler : 1 2 3 4
Défiler : 1 2 3 4



En Ocaml :

```
type 'a file = {anc : 'a list; nouv : 'a list};;  
  
let vide = {anc = []; nouv = []};;  
  
let enfile f x = {anc = f.anc; nouv = x :: f.nouv};;  
  
let defile f = match f.anc with  
| [] -> let t::q = List.Rev f.nouv in t, {anc = q; nouv = []}  
| t::q -> t, {anc = q; nouv = f.nouv}
```

Complexité

- ★ enfile est en $O(1)$
- ★ defile est en $O(|\text{nouveaux}|)$ ce qui dans le pire des cas est en $O(n)$
- ★ Mais, en regardant sur tous les défilages on peut obtenir mieux.

Démonstration. Notons (t_i) les instants où l'on inverse nouveaux (noté $\text{nouveaux}[t_i]$).

Notons C_d la complexité de tous les appels à defile

$$C_d = \underbrace{n \times O(1)}_{\text{tous sauf le renversement}} + \sum_i O(|\text{nouveaux}[t_i]|)$$

$$= O(n) + O\left(\sum_i |\text{nouveaux}[t_i]|\right) \quad \leftarrow \begin{array}{l} \triangle \text{ A priori c'est illégal. On ne peut que} \\ \text{quand le nombre de terme de la somme est constant} \end{array}$$

Or pour $t_i < t_j$, $\text{nouveaux}[t_i] \cap \text{nouveaux}[t_j] = \emptyset$.

Donc $C_d = O(n) + O\left(\left|\bigcup_i \text{nouveaux}[t_i]\right|\right) = O(n)$.

En n opérations, on a donc un coût en $O(n)$. Donc si on lisse sur chaque opération, on obtient ce qu'on appelle le coût amorti, qui vaut ici $O(1)$. \square

Retour sur \triangle En effet, $k \underset{n \rightarrow +\infty}{=} O(1)$ donc $\sum_{k=1}^n k = \sum_{k=1}^n O(1)$, pourtant $\sum_{k=1}^n k = \Theta(n^2)$. On peut inverser car la constante derrière est la même.

$$\sum_i C(\text{nouveaux}[t_i]) \leq \sum_i K \times |\text{nouveaux}[t_i]| \leq K \times \sum_i |\text{nouveaux}[t_i]| = O\left(\sum_i |\text{nouveaux}[t_i]|\right)$$

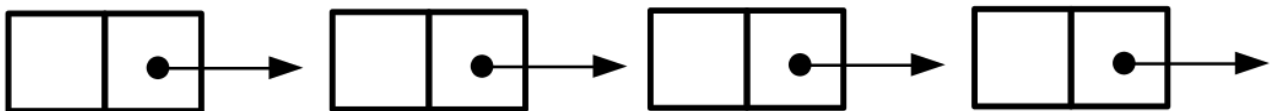
Commentaire 28.1 On peut insister ici que le K est le même pour tous.

Commentaire 28.2 A la place de la section suivante, ou en plus si jamais on est trop en avance, on peut également profiter du contexte pour illustrer les différentes manières de faire de la complexité amortie (banquier et potentiel).

Peut on faire mieux ?

Une liste doublement chaînée peut tout faire.

En CamL, les listes sont simplement chaînées :



Liste doublement chaînée :



Implémentation en C :

```
typedef struct noeud noeud;;
struct list{
    noeud *prec , *suiv;
    void *valeur;
};
```

```
typedef struct {  
    noeud *debut, *fin;  
} liste;
```

On peut alors faire plus d'opérations, toujours en $O(1)$ (pas amortie) mais plus technique dans la manipulation de pointeur et la gestion de la mémoire.

Si on veut supprimer au début :

```
void enleve_tete(struct list *l){  
    if (l->fin == NULL)  
        exit(1);  
    noeud tmp = l->fin;  
    l->fin = tmp->prec;  
    l->fin->suiv = NULL;  
    free(tmp);  
}
```

Développement 29

Zoom sur le protocole TCP

Auteur·e·s: Daphné Kany

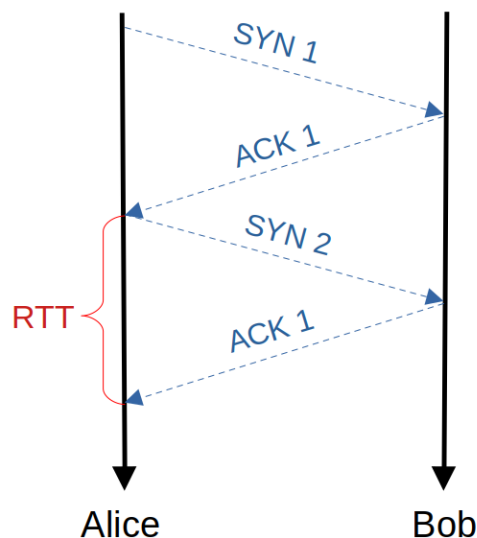
Références :

On présente l'utilité de la fenêtre glissante dans le protocole TCP : d'une part pour augmenter le taux d'envoi, et de l'autre pour réguler le trafic et éviter la congestion.

Commentaire 29.1 On rappelle rapidement à l'orale l'existence du three way handshake précisé dans le plan. On se place ensuite directement au moment de l'échange de données.

Rappel

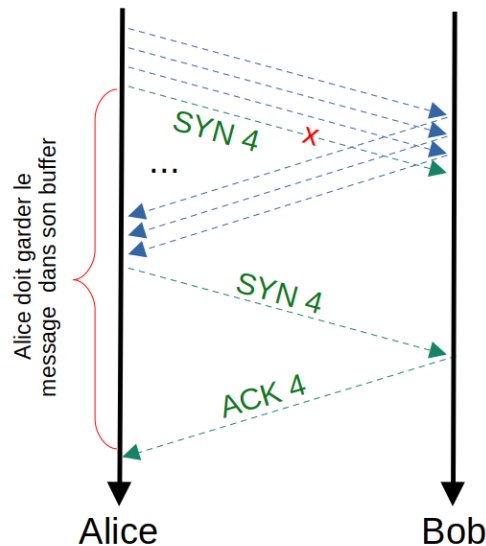
Alice et Bob communiquent en utilisant le protocole TCP. On suppose ici leur communication asymétrique : Alice envoie les données, Bob se contente de retourner les ACK.



Quel est le taux d'envoi ? $\frac{1}{RTT}$. On voudrait améliorer ce taux.

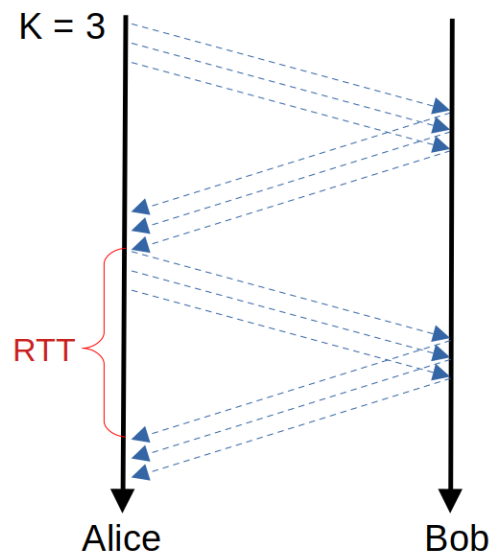
Solution 1 : Envoi en continu

Alice envoie tous les messages en continu. Problème : en cas d'erreur, Alice doit renvoyer le message et donc le garder en mémoire jusqu'à réception du ACK correspondant. Son buffer est limité.



Solution 2 : La fenêtre glissante

On considère qu'Alice ne garde que k messages dans son buffer. Lorsqu'il est plein, elle cesse d'envoyer. Dès qu'elle reçoit un ACK, elle supprime le message et en envoie un nouveau.



Taux d'envoi : $\frac{k}{RTT}$ Comment déterminer la valeur de k ?

Compromis entre le taux d'envoi et la taille du buffer d'Alice, et de ceux des routeurs sur le réseau pour éviter la congestion.

Remarque 29.1 Alice estime ici qu'il y a congestion si elle ne reçoit pas les ACK à temps. C'est en pratique assez vérifié pour les connexions fiables.

Algorithme 29.1 : Taille de la fenêtre glissante

$k \leftarrow 1$

tant que il n'y a pas d'erreur **faire**

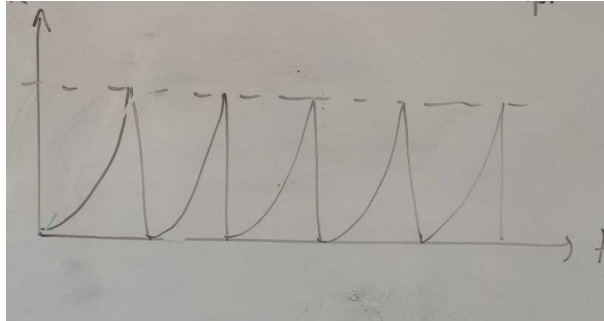
 Augmenter k

Recommencer

A quelle vitesse augmenter k ?

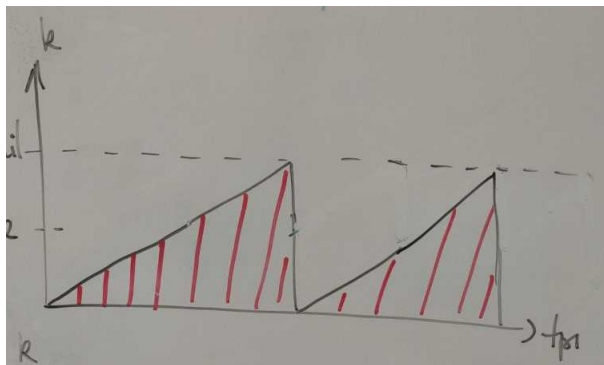
On l'augmente vite (exponentiellement)

Commentaire 29.2 Après tout c'est logique : on veut atteindre une grande valeur vite. Mais en fait, on n'en profite pas longtemps.



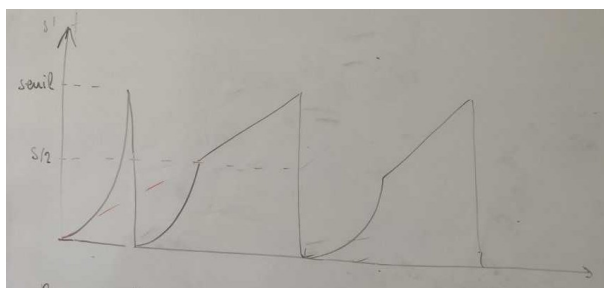
On l'augmente pas vite (linéairement)

Commentaire 29.3 C'est mieux. Mais on met du temps à atteindre le seuil. Combinons ces deux propositions :)



Combinons :)

Commentaire 29.4 Et beh quelle bonne idée.



Et la descente ?

On peut aussi jouer sur la descente. On ne repart plus de $k=1$.

Développement 30

Indécidabilité de la terminaison et de la correction partielle

Auteur·e·s: Emile Martinez

Références :

Ce développement montre l'indécidabilité de la terminaison et de la correction partielle. Il se place à bas niveau, (début prépa) puisque on repasse sur toutes les difficultés de formalisation. Il est donc à priori plutôt adapter à des leçons comme sur la correction des programmes (qui aurait lieu tôt) que à des leçons qui auraient lieu tard (comme indécidabilité)

Théorème 30.1 *Il n'existe pas d'algorithme \mathcal{A} décidant pour tout algorithme \mathcal{B} , si \mathcal{B} termine en temps fini sur une entrée w .*

Algorithme : Un programme \mathcal{C} avec une mémoire infini qui a pour seule entrée une chaîne de caractères. Cela représente du pseudo code.

décider : Renvoie Vrai ou Faux en temps fini *Démonstration*. Par l'absurde. Supposons qu'il existe un tel \mathcal{A} .

Construisons \mathcal{D} qui sur l'entrée $\langle \mathcal{C} \rangle$:

Exécute \mathcal{A} sur $\langle \mathcal{C}, \langle \mathcal{C} \rangle \rangle$

Si \mathcal{A} renvoie Vrai

Boucle

Sinon

Termine

On note alors $\langle a \rangle$ la chaîne de caractère décrivant l'objet a .

Exemple 30.1 $\langle \text{"patate"}, \text{"carottes"} \rangle = \text{"patate\#carottes"}$.

Question Est ce que \mathcal{D} est un algorithme.

- ★ On peut vérifier si l'entrée est le code d'un programme \mathcal{C} . (C'est la première étape que fait un compilateur)
- ★ On peut «exécuter \mathcal{A} », car \mathcal{A} existant (par hypothèse) on peut écrire un code.
- ★ On peut écrire $\langle \mathcal{C}, \langle \mathcal{C} \rangle \rangle$, car \mathcal{C} existe donc son code aussi.
- ★ Boucler ou terminer, on peut le faire en \mathcal{C} .

Réponse \mathcal{D} est un algorithme.

Regardons le comportement de \mathcal{D} sur $\langle \mathcal{D} \rangle$.

A-t-on le droit ?

Oui, car on définit \mathcal{D} avant, donc $\langle \mathcal{D} \rangle$ existe et \mathcal{D} prend n'importe quel code d'algorithme.

Commentaire 30.1 *On a donc pas de définition récursive. On ne définit pas $\mathcal{D}_{\mathcal{D}}$*

Si \mathcal{D} termine sur $\langle \mathcal{D} \rangle$ termine :

Alors \mathcal{A} sur $\langle \mathcal{D}, \langle \mathcal{D} \rangle \rangle$ renvoie Vrai, donc \mathcal{D} sur $\langle \mathcal{D} \rangle$ boucle.

Sinon \mathcal{D} sur $\langle \mathcal{D} \rangle$ boucle :

Alors \mathcal{A} sur $\langle \mathcal{D}, \langle \mathcal{D} \rangle \rangle$ renvoie Faux, donc \mathcal{D} sur $\langle \mathcal{D} \rangle$ termine.

Ce qui est absurde.

□

Théorème 30.2 *La correction partielle est indécidable.*

Plus rigoureusement, pour toute propriété \mathcal{P} non triviale liant les entrées aux sorties, il n'existe pas d'algorithme \mathcal{A} tel que sur l'entrée $\langle \mathcal{B} \rangle$, \mathcal{A} décide si, quand \mathcal{B} termine, son entrée et sa sortie vérifie \mathcal{P} .

Démonstration. Supposons \mathcal{P} : «La sortie vaut 2 fois l'entrée si l'entrée est un entier.»

Par l'absurde, supposons \mathcal{A} décidant la correction partielle de \mathcal{P} .

On crée alors \mathcal{B} qui sur l'entrée $\langle \mathcal{C}, w \rangle$:

Crée le programme \mathcal{D} qui sur l'entrée $\langle n \rangle$

Si $n = 2$

Exécute \mathcal{C} sur w

Renvoie 5

Sinon

Renvoie $2n$

Exécute \mathcal{A} sur \mathcal{D} .

\mathcal{B} décide alors le problème de l'arrêt.

□

Commentaire 30.2 *Suivant le temps qu'il reste, on peut ou non parler de est-ce que c'est un algorithme.*

On peut alors expliquer que ça en est un sans parler de machine universelle. En effet, pour créer le programme \mathcal{D} on peut concaténer les instructions au programme \mathcal{C} . De même, on en a pas besoin pour exécuter \mathcal{A} . Il suffit d'écrire son code dans le programme.

Développement 31

LZW

Auteur·e·s: Emile Martinez

Références : Balabonski MP2I (notamment les quantifications à la fin)

Idée Compresser les répétitions de sous-chaines.

→ Cette algorithme est online

Commentaire 31.1 Mentionner la différence avec Huffman qui est pas online et qui fonctionne pas sur les sous-chaines mais directement sur les caractères. Et que si on faisait huffman sur les mots, on aurait un très gros truc à rajouter au début, et en plus le caractère ' ' serait arbitraire.

1. On associe un entier à chaque lettre de notre alphabet → dictionnaire
2. On parcourt le mot pas encore dans le dictionnaire :
 - On met dans le dictionnaire le plus grand préfixe possible
 - On ajoute au dictionnaire une nouvelle clé : préfixe + lettre suivante (nouveau numéro)

Exemple "ALLALALA", $\Sigma = \{A, B\}$

ALLALALA	Sortie	Dictionnaire
		A → 0
		L → 1
<u>A</u> L	0	AL → 2
<u>L</u> L	1	LL → 3
<u>L</u> A	1	LA → 4
<u>AL</u> A	2	ALA → 5
<u>ALA</u>	5	

Commentaire 31.2 Essayons de faire un algorithme de décompression

Entrée	Sortie	Dictionnaire
		0 → A 1 → L
0	A	
1	L	2 → AL
1	L	3 → LL
2	L	4 → LA

Commentaire 31.3 Ça a l'air de fonctionner. Essayons d'écrire un code pour cela.
(Quand on le fait, laisser de la place au milieu pour pouvoir faire les modifications à venir dessus)

```

d ← dictionnaire initial
vieux_i ← None
indice ← |d|
tant que il y a une entrée faire
    i ← input()
    print(d[i])
    si vieux_i ≠ None alors
        d[indice] ← d[vieux_i] + d[i][0]
    vieux_i ← i
    indice ++

```

Commentaire 31.4 Reprenons l'exécution sur l'exemple

Problème On a pas 5 dans le dictionnaire. On a réutiliser tout de suite le facteur que l'on a ajouter. **Il faut traiter à part ce cas là.**

Commentaire 31.5 Bien expliquer sur la compression, que c'est parce que on a rajouter 5 puis utiliser 5. Que on a le problème parce que on ajoute dans le dico avec un de décalage à la décompression. Que par conséquent le problème vient du fait que on reconnu le motif que l'on vient d'ajouter, donc on connaît la lettre suivante, c'est celle du mot d'avant.

```

d ← dictionnaire initial
vieux_i ← None
indice ← |d|
tant que il y a une entrée faire
    i ← input()
    si i ∈ d alors
        print(d[i])
        si vieux_i ≠ None alors
            d[indice] ← d[vieux_i] + d[i][0]
    sinon
        d[indice] ← d[vieux_i] + d[vieux_i][0]
        print(d[indice])
    vieux_i ← i
    indice ++

```

Implementation

- Représentation des entiers
 - Nombre limité de bits (ex. 12) mais dico limité.
 - On arrête de le compléter
 - On vide et on recommence
 - On augmente la taille des représentations au cours de l'algo
- Choix de l'alphabet
 - {0, 1}
 - les 256 char possibles

Commentaire 31.6 Sur les tour du monde en 80 jours, avec l'alphabet $\{0,1\}$, 29%, avec l'alphabet de tous les char 57%, en fixant la taille du dictionnaire à 16 (moins, on détecte moins de répétition, plus on a trop de bits pour trop peu de facteurs). Avec Huffman, on faisait 44%.

Question Gagne-t-on toujours ?

Compression : $LZW\Sigma^* \rightarrow \Sigma^*$. A-t-on alors, $\forall w \in \Sigma^*$, $|LZW(w)| \leq |w|$.

$fLZW$ est injectif donc non (en effet, $|LZW(\Sigma^n)| = |\Sigma^n| > |\Sigma^{n-1}|$).

Donc LZW, des fois, on y perd. 0011 on aura perdu, car on devra stocker nos facteurs sur 2 bits, mais en ne codant que des lettres (7 bits en tout au lieu de 4).

Remarque On peut choisir de ne perdre au plus que 1 bit. $f : \Sigma^* \rightarrow \Sigma^*$

$$w \mapsto \begin{cases} 0.LZW(w) & \text{si } |LZW(w)| < |w| \\ 1.w & \text{sinon} \end{cases}$$

Développement 32

Vérification du produit de matrice

Auteur·e·s: Marin Malory (légèrement modifié par Emile Martinez)

Références : Mitzenmacher

Ce développement présente un exemple d'utilisation d'aléatoire afin de vérifier si un produit de matrice est correct de manière efficace. Il s'intègre ainsi dans la leçon sur les tests de programmes et de part sa nature probabiliste, il illustre la leçon sur les algorithmes probabilistes.

Contexte On suppose que l'on a un algorithme produit qui calcule le produit de matrice efficacement. On veut vérifier qu'il est correct sur de grandes entrées. Comment faire ?

On pourrait calculer le produit, mais si les matrices sont grandes cela serait très long. On peut alors utiliser une méthode probabiliste.

Notre problème est donc de vérifier, étant donné A , B et C si $A \times B = C$. On se concentrera ici sur des matrices d'entiers modulo 2.

Rappel $M_1 = M_2 \Leftrightarrow \forall r \in \{0, 1\}^n, M_1 r = M_2 r$.

Idée Prendre des vecteurs au hasard, et vérifier que cela fonctionne.

Intérêts Il se trouve que pour calculer ABr , on peut ne pas calculer AB , en utilisant l'associativité des matrices $\rightarrow ABr = A \times (B \times r) \rightarrow O(n^2)$ (car multiplier une matrice et un vecteur est $O(n^2)$).

Théorème 32.1 Soient $A, B, C \in \{0, 1\}^{n \times n}$. Si $AB \neq C$, et si r est choisit uniformément dans $\{0, 1\}^n$, alors

$$\mathbb{P}(ABr = Cr) \leq \frac{1}{2}$$

Démonstration . L'événement considéré est « $ABr = Cr$ ». Soit $D = AB - C \neq 0$. Ainsi, $ABr = Cr$ implique $Dr = 0$. Puisque $D \neq 0$, il existe au moins un coefficient non nul : on prend $d_{11} \neq 0$ sans perdre de généralité.

Puisque $Dr = 0$, on a :

$$\sum_{j=1}^n d_{1j} r_j = 0$$

et de manière équivalente :

$$r_1 = - \frac{\sum_{j=2}^n d_{1j} r_j}{d_{11}} \quad (32.1)$$

On peut supposer que l'on choisit (r_2, \dots, r_n) uniformément dans $\{0, 1\}^{n-1}$ et r_1 uniformément dans $\{0, 1\}$. Ces deux tirages sont réalisés de manière indépendante.

D'après la formule des probabilités totales :

$$\begin{aligned}
 \mathbb{P}(ABr = Cr) &= \sum_{(x_2, \dots, x_n) \in \{0, 1\}^{n-1}} \mathbb{P}(ABr = Cr \cap (r_2, \dots, r_n) = (x_2, \dots, x_n)) \\
 &\leq \sum_{(x_2, \dots, x_n) \in \{0, 1\}^{n-1}} \mathbb{P}\left(r_1 = -\frac{\sum_{j=2}^n d_{1j} r_j}{d_{11}} \cap (r_2, \dots, r_n) = (x_2, \dots, x_n)\right) \\
 &= \sum_{(x_2, \dots, x_n) \in \{0, 1\}^{n-1}} \mathbb{P}\left(r_1 = -\frac{\sum_{j=2}^n d_{1j} x_j}{d_{11}} \cap (r_2, \dots, r_n) = (x_2, \dots, x_n)\right) \\
 &= \sum_{(x_2, \dots, x_n) \in \{0, 1\}^{n-1}} \mathbb{P}\left(r_1 = -\frac{\sum_{j=2}^n d_{1j} x_j}{d_{11}}\right) \cdot \mathbb{P}((r_2, \dots, r_n) = (x_2, \dots, x_n)) \\
 &= \sum_{(x_2, \dots, x_n) \in \{0, 1\}^{n-1}} \frac{1}{2} \cdot \mathbb{P}((r_2, \dots, r_n) = (x_2, \dots, x_n)) \\
 &= \frac{1}{2}
 \end{aligned}$$

□

On amplifie alors le résultat.

Algorithme 32.1 : $\text{verification}(A, B, C, k)$

```

pour i allant de 1 à k faire
    Choisir uniformément r
    si  $A \times (B \times r) \neq C \times r$  alors
        retourner Faux
retourner Vrai

```

Efficacité Si $AB = C$, l'algorithme renvoie la bonne réponse, sinon, les choix des r étant indépendants, la probabilité de ne pas détecter une erreur est $\leq 2^{-k}$. Donc notre algorithme réussit avec probabilité au moins $1 - 2^{-k}$, en $(Ok \times n^2)$

En vrai Ainsi, pour $k = 100$, on échoue avec probabilité 2^{-100} soit environ une chance sur 1 quintillion (donc jamais). Ainsi, on a un algorithme en $O(100 \times n^2)$ qui tant en théorie que en vrai, peut être considéré en $O(n^2)$ qui vérifie en gros sans erreur le produit de matrice.

Commentaire 32.1 Et si jamais c'est faux, on va probablement finir beaucoup plus vite que ça, car il y a aura souvent plusieurs coefficient faux.

Commentaire 32.2 Pour éventuellement gagner un peu de temps

Question Qu'en est il si on ne considère pas des entiers modulus 2 ? Et bien la probabilité de se tromper est encore plus faible, car on a plus de valeurs possibles.

Commentaire 32.3 Et au pire on peut également parler des flottants si vraiment on est trop en avance.

Développement 33

Algorithme de Peterson

Ce développement présente l'algorithme de Peterson qui permet de résoudre le problème de l'exclusion mutuelle pour deux processus, le but étant de montrer la correction de l'algorithme. Il s'insère naturellement dans les leçons qui abordent la synchronisation et la gestion de ressources, comme les leçons ??, ?? et ??. Enfin, il peut illustrer la leçon ?? si le paradigme de programmation concurrente est abordé.

Introduction. Lorsqu'il y a une condition de concurrence, on veut que nos différents processus soient en exclusion mutuelle, c'est-à-dire qu'aucun des processus ne rentre dans leur section critique en même temps. Après une proposition non satisfaisante de Dekker (présentée par Dijkstra), Peterson a proposé une méthode élégante pour résoudre ce problème avec deux processus.

```
#define FALSE 0
#define TRUE 1
#define N 2 / * nombre de processus * /
int turn;
int interested[N]; / * à qui le tour? * /

/ * on initialise les valeurs à 0 (FALSE) * /

void enter_region(int process) / * le processus est 0 ou 1 * /
{
    int other; / * nombre de l'autre processus */
    other = 1 - process; / * l'opposé du processus */

    interested[process] = TRUE; / * on est intéressé */
    turn = other; / * on initialise le drapeau */

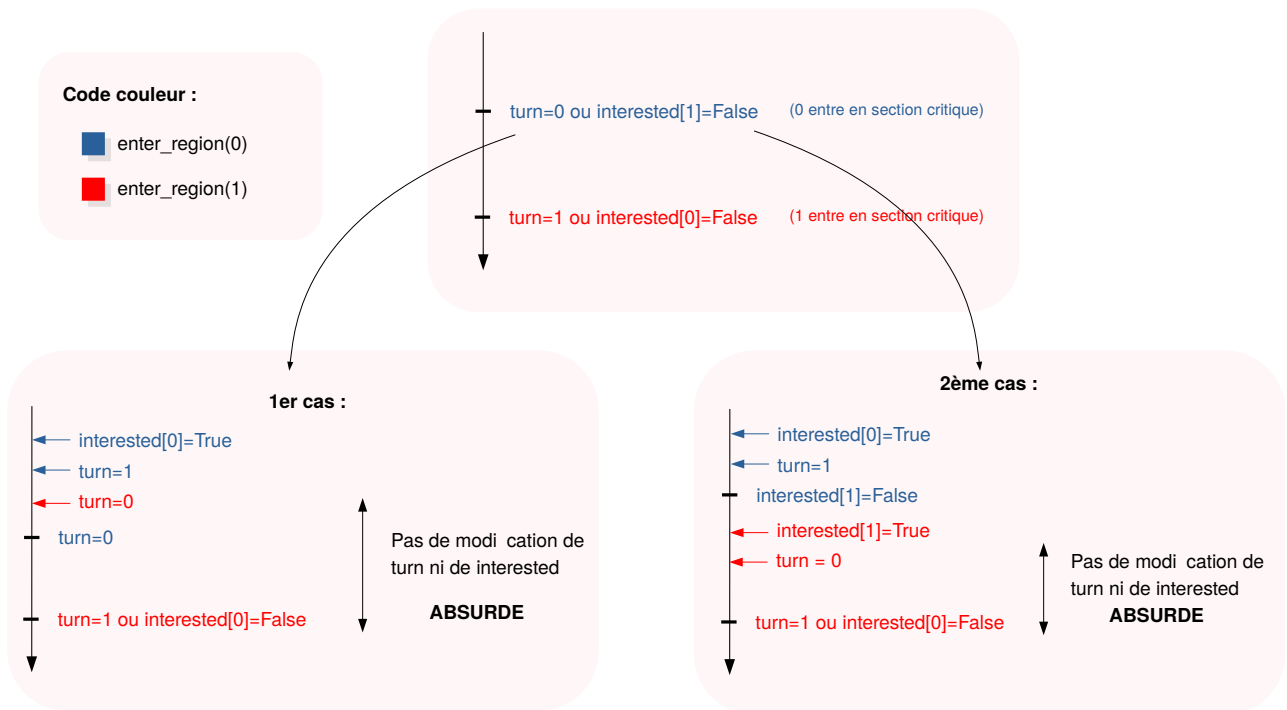
    / * attente */
    while (turn == other && interested[other] == TRUE) ;
}

void leave_region(int process)
{
    / * sortie de la région critique */
    interested[process] = FALSE;
}
```

Exclusion mutuelle.

Proposition 33.1 *L'algorithme de Peterson vérifie la propriété d'exclusion mutuelle, c'est-à-dire deux processus ne rentrent jamais dans leurs sections critiques en même temps.*

Démonstration. On raisonne par l'absurde. Supposons que le processus 0 soit dans sa section critique, et le processus 1 accède à la sienne. Ainsi, `enter_region(0)` a terminé (le processus 0 a réussi à rentrer dans sa section critique) et seulement après l'appel `enter_region(1)` a terminé.



□

Exclusion mutuelle.

Proposition 33.2 *L'algorithme de Peterson ne provoque jamais d'interblocage, c'est-à-dire que si deux processus essayent d'accéder à leur section critique, au moins l'un des deux y arrive.*

Démonstration. On raisonne par l'absurde, si les deux appels sont bloqués dans la boucle `while`, on a à la fois `turn == 1` et `turn == 0`. □

Absence de famine.

Proposition 33.3 *L'algorithme de Peterson assure l'absence de famine, c'est-à-dire que tout processus qui demande à rentrer en section critique y accède en temps fini, à condition que chaque processus reste un temps fini dans sa section critique.*

Démonstration. Par l'absurde, si le processus 0 lance son appel `enter_region(0)` et reste bloqué indéfiniment dans sa boucle `while` à partir de l'instant t_0 . Donc pour tout temps $t \geq t_0$, on a `turn == 0` et `interested[1]`

`== TRUE`. Par la deuxième contrainte, on sait que le processus 1 a appelé `enter_region(1)`. De plus, le processus 1 ne reste pas bloqué, puisque sinon il y aurait interblocage. Ainsi, 1 entre en section critique et en sort en temps fini, il appelle alors `leave_region(1)`. À la fin de cet appel, on a `interested[1] == FALSE`. Il y a alors deux cas :

- si le processus 0 reprend la main, alors on a une absurdité puisque `interested[1] == FALSE` ;
- si le processus 1 garde et appelle à nouveau `enter_region(1)`, alors il reste cette fois bloqué dans le `while` puisqu'il est le dernier à avoir modifié `turn`. Ainsi, lorsque le processus 0 reprend la main, il est débloquent, ce qui est absurde aussi.

□

Temps d'attente borné.

Proposition 33.4 *L'algorithme de Peterson assure une attente bornée, c'est-à-dire qu'un nombre borné de processeur peuvent passer avant lui lorsqu'il demande à entrer dans sa section critique.*

Démonstration. Il suffit de reprendre la preuve précédente.

□

Développement 34

Construction d'un additionneur à retenue anticipée

Auteur·e·s: Sorci Émile, Marin Malory

Références : [?]

Ce développement présente la construction d'un additionneur rapide : l'additionneur à retenue anticipée (carry-lookahead adder). Ce développement est parfait pour présenter un circuit combinatoire complexe, illustrant la leçon ???. De plus, ce circuit utilise un paradigme diviser-pour-régner, illustrant ainsi la leçon ??? en présentant une implémentation matérielle d'un algorithme. Enfin, ce circuit illustre la manière dont les opérations arithmétiques sont réalisées en machine, et illustre ainsi certains principes de fonctionnement des ordinateurs, illustrant la leçon ???.

Introduction. Un additionneur n bits est un circuit combinatoire : il peut être écrit sous la forme d'une formule logique. L'additionneur classique (*full-adder*) présente un problème majeur : pour calculer le i -ème bit du résultat, on doit attendre la retenue sortante c_i . Dans ce circuit, le chemin critique est alors de taille $\mathcal{O}(n)$.

On pourra ici dessiner un full-adder et surligner le chemin critique de longueur n .

$$s_i = a_i \overline{b_i} \overline{c_i} + \overline{a_i} b_i \overline{c_i} + \overline{a_i} \overline{b_i} c_i + a_i b_i c_i \quad (34.1)$$

$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i \quad (34.2)$$

On peut alors réécrire l'équation précédente :

$$c_{i+1} = g_i + p_i c_i, \quad g_i = a_i b_i, \quad p_i = a_i + b_i \quad (34.3)$$

- si g_i est vraie, alors c_{i+1} est vraie, et donc une retenue est **générée** ;
- si p_i est vraie, alors si c_i est vraie, alors on a c_{i+1} qui est vraie, la retenue a été **propagée**.

En itérant, on a alors :

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} + p_i p_{i-1} g_{i-2} + \dots + p_i p_{i-1} \dots p_1 p_0 g_0 \quad (34.4)$$

CLA. L'additionneur qui calcule les retenues en utilisant l'équation 34.4 s'appelle **additionneur à retenue anticipée**, ou CLA pour *carry-Lookahead adder*. Malheureusement, cette formule n'est pas utilisable directement telle quelle (il fait faire un OR à n entrées et un AND à n entrées, et des fils très longs).

On va donc travailler par bloc :

- P_{ij} : une retenue est propagée de i à j .
- G_{ij} : une retenue est générée entre i et j ;

On peut les définir par récurrence de la manière suivante :

$$P_{i,j+1} = P_{i,j}p_{j+1} \quad (34.5)$$

$$G_{i,j+1} = g_{j+1} + p_{j+1}G_{i,j} \quad (34.6)$$

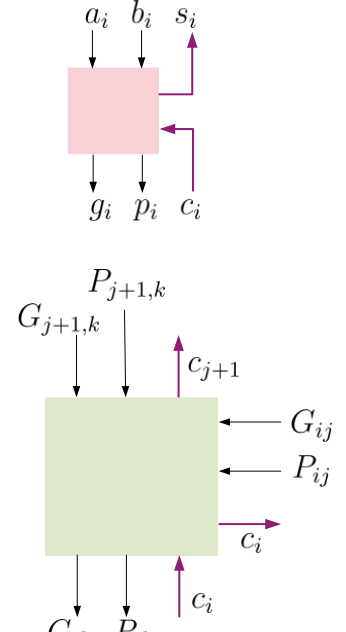
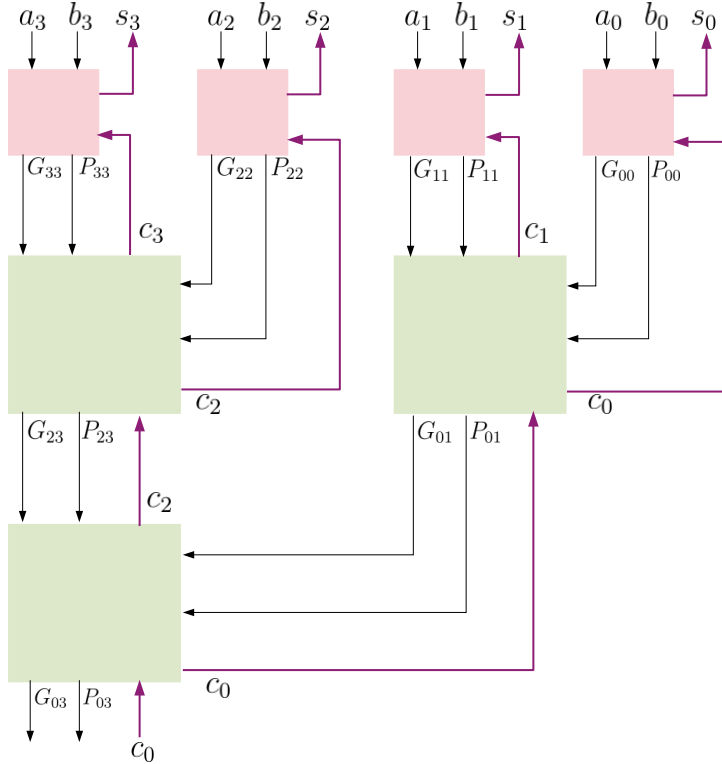
On a alors pour tout $i \leq j \leq k-1$:

$$G_{ik} = G_{j+1,k} + P_{j+1,k}G_{ij} \quad (34.7)$$

$$P_{ik} = P_{ij}P_{j+1,k} \quad (34.8)$$

$$c_{j+1} = G_{ij} + P_{ij}c_i \quad (34.9)$$

On peut alors dessiner un additionneur à retenue anticipée sur 4 bits.



En notant c_v et c_r les longueurs des chemins critiques dans les blocs vert et rouge, on remarque que pour n bits, la longueur du chemin critique $l_{CLA}(n)$ vérifie l'inéquation :

$$l_{CLA}(n) = c_v + l_{CLH}(n/2)$$

et $l_{CLA}(1) = c_r$. Ainsi, on a, on supposant que n est une puissance de 2 :

$$l_{CLA} = \log_2(n)c_v + c_r = \mathcal{O}(\log_2(n))$$

Développement 35

Distance d'édition

Auteur·e·s: Emile Martiez

Références :

Définition 35.1

$$\text{lev}(w_1, w_2) = \min \{k \in \mathbb{N} / \exists f_1, \dots, f_k \in \{\text{ins}_{a,i}, \text{sub}_{a,i}, \text{sup}_i / a \in \Sigma, i \in \mathbb{N}\} : w_2 = f_k \circ \dots \circ f_1(w_1)\}$$

Proposition 35.1

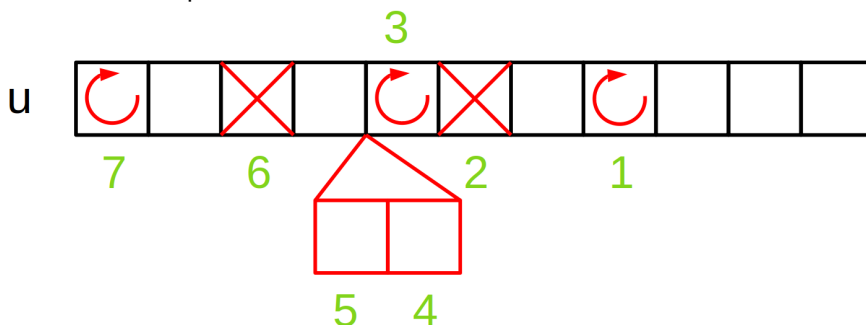
$$\begin{aligned} \text{lev}(u, \epsilon) &= |u| \\ \text{lev}(\epsilon, v) &= |v| \\ \text{lev}(u.a, v.b) &= \min \begin{cases} \text{lev}(u, v) + 1 & \text{sub} \\ \text{lev}(u.a, v) + 1 & \text{ins} \\ \text{lev}(u, v.b) + 1 & \text{sup} \\ \text{lev}(u, v) & \text{si } a = b \end{cases} \end{aligned}$$

Idee On peut commencer par la dernière lettre, et alors appliquer une des transformations possibles. Mettons cette idée en preuve.

Notation Pour $f \in \{\text{ins}_{a,i}, \text{sub}_{a,i}, \text{sup}_i\}$, notons $i = \alpha(f)$.

Lemme 35.1 Pour $u, v \in \Sigma^*$, avec $k = \text{lev}(u, v)$, il existe une transformation minimale $f_k \circ \dots \circ f_1$ avec $\alpha(f_j) \leq \alpha(f_{j-1})$ avec égalité seulement si $f_j = \text{ins}$

Intuition On peut traiter les lettres dans l'ordre, et une fois chacune.



Démonstration. Intuition de la preuve : On fait des inversions comme pour un tri à bulle.

Tant qu'il reste j ne vérifiant pas la condition, alors on échange f_j et f_{j-1} . Par exemple : si $f_j = ins_{b,i_1}$ et $f_{j-1} = sup_{i_2}$ avec donc $i_1 > i_2$ (on est dans le cas d'égalité sinon), alors $f_j \circ f_{j-1} = sup_{i_2} \circ ins_{b,i_1+1}$. Les seuls cas où l'on ne peut pas inverser sont ceux où la même lettre est concerné (ex : $sup_i \circ ins_i$) mais qui sont interdits par minimalité.

De plus, nos transformations termineront car les indices ne concernent que des lettres valides et ne font que croître. \square

Commentaire 35.1 Ne pas hésiter à montrer les inversions et les différents cas sur le dessin.

Retour sur la propriété Montrons maintenant par induction que nous calculons la bonne distance.

Démonstration.

- ★ Si $u = \epsilon$ ou si $v = \epsilon$, alors on ne peut en effet pas faire mieux que $|v|$ insertions ou $|u|$ suppressions.
- ★ On prend une transformation minimale $f_k \circ \dots \circ f_1$ du lemme. Alors
 - Soit $\alpha(f_1) < |u|$ et alors, comme on ne pourra jamais toucher de lettre plus loin que f_1 , $a = b$, et la transformation restante est minimale pour u, v (sinon on pourrait en faire une plus petite pour $u.a, v.b$), d'où le résultat par hypothèse d'induction
 - Soit $\alpha(f_1) = |u|$ et alors il fait soit une suppression (ce que l'on gère), soit une substitution (resp. une insertion). Il ne peut alors que mettre la lettre b (car on ne changera rien après b). Ainsi on a bien la mise de b , puis une transformation de u en v (resp. de $u.a$ en v) (car on ne retouchera aucune lettre après b). Or cette transformation est minimale car sinon on aurait pu faire mieux. D'où le résultat par hypothèse d'induction.

Commentaire 35.2 On ne montre ici que l'inégalité difficile, mais on obtient la facile en disant que ce qu'on fait sinon c'est quand même des transformations, donc elle ne font pas mieux que le minimum.

\square

Bon là du coup on montrait le tableau qu'on prend, la complexité, et tout le tin toutin. A voir ce qu'on récupère de Malloy pour le mettre ici

Développement 36

Automate des motifs

Auteur·e·s : Emile Martinez, Malory Marin, Daphné Kany

Références : 131 développements

Ce développement présente un algorithme permettant de résoudre le problème de recherche de motif dans un texte. Pour cela, on construit un automate minimal en pré-traitement, permettant ensuite de résoudre le problème linéairement en la taille du texte. Ainsi, il s'intègre aussi bien dans la leçon ?? que dans la leçon ?. Enfin, il peut illustrer la leçon ?? si la programmation orienté automate est abordée.

Introduction. On souhaite détecter si M est un sous-mot de T . On prend alors $M \in \Sigma^*$, $|M| = k$ et on cherche à construire un automate \mathcal{A} tel que $\mathcal{L}(\mathcal{A}) = \Sigma^*M$

Notation

- Si $u, v \in \Sigma^*$, on note $u \sqsubset v$ si u est suffixe de v .

Exemple : $ab \sqsubset abbab$

- On note M_i le i -ème préfixe de M .

Exemple : Si $M = abbab$, $M_0 = \varepsilon$, $M_1 = a$, $M_2 = ab$, $M_3 = abb$, etc.

- On note $\sigma(u) = \max\{i / M_i \sqsubset u\}$. C'est la taille du plus grand préfixe de M qui est également suffixe de u .

Exemple : $M = abaa$, $u = aaba$, $\sigma(u) = \max\{|a|, |aba|\} = 3$

Construction d'un automate. Soit $A = (Q, \Sigma, I, F, \delta)$ l'automate déterministe complet défini par :

- $Q = \{0, \dots, k\}$;
- $I = \{0\}$;
- $F = \{k\}$;
- pour tout $q \in Q$ et $a \in \Sigma$, $\delta(q, a) = \sigma(M_q a)$.

Exemple 36.1 $\Sigma = \{a, b\}$, $M = ab$

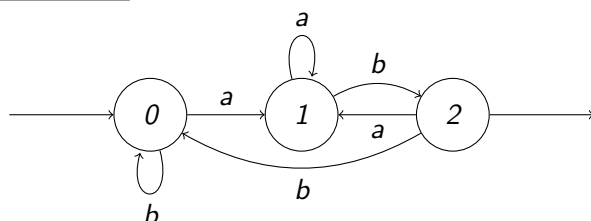
$\delta :$

	a	b
0	1	0
1	1	2
2	1	0

$$\delta(1, a) = \sigma(aa) = 1$$

$$\delta(1, b) = \sigma(ab) = 2$$

$$\delta(2, a) = \sigma(aba) = 1$$



Commentaire 36.1 Expliquer ici pourquoi en effet, notre automate reconnaît bien ce qu'on veut. On peut à cette occasion déjà mentionner le fait que l'état i correspond à au quel point au max, on a déjà lu M

Correction $\mathcal{L}(\mathcal{A}) = \Sigma^* M$

Proposition 36.1 Pour tout mot $u \in \Sigma^*$, on a $\delta^*(0, u) = \sigma(u)$

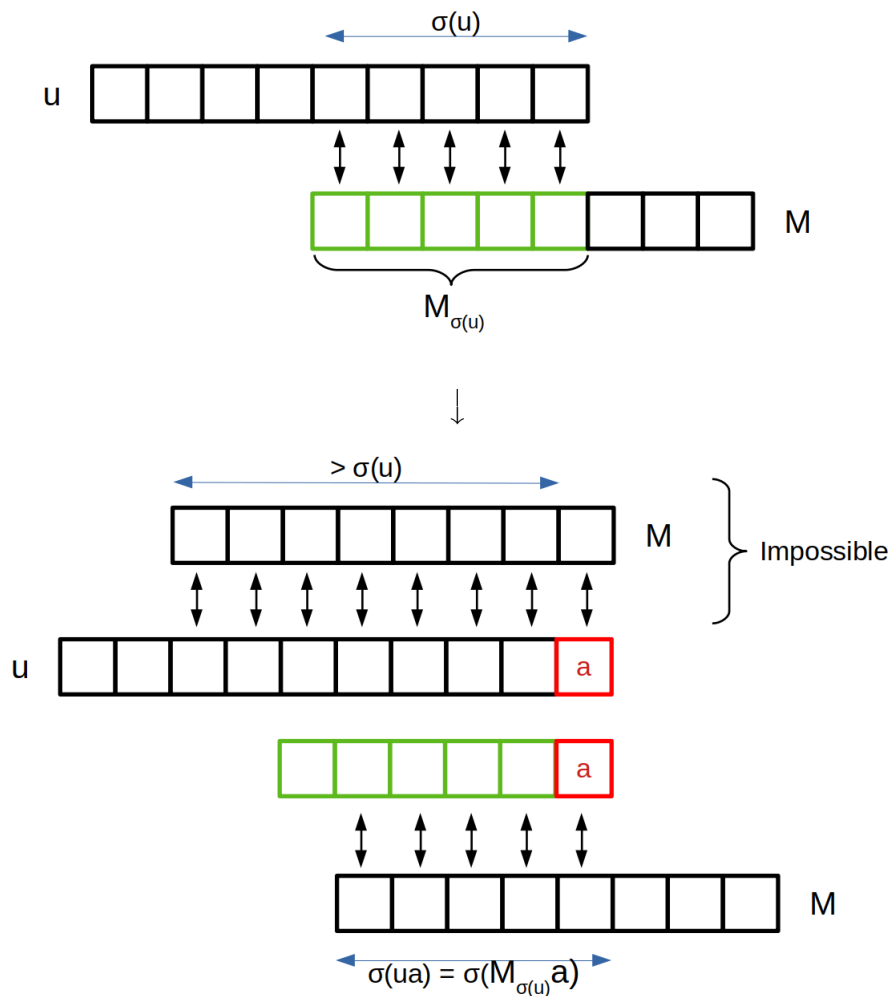
Démonstration. On procède par récurrence sur la longueur l de u .

- Si $l = 0$, alors $u = \epsilon$ et $\delta^*(0, \epsilon) = 0 = \sigma(\epsilon)$.
- Supposons que $l > 0$ et que tout mot $v \in \Sigma^{l-1}$ vérifie $\delta^*(0, v) = \sigma(v)$. Soit $u \in \Sigma^l$, qu'on écrit $u = va$ avec $|v| = l - 1$ et $a \in \Sigma$. En appliquant l'hypothèse de récurrence :

$$\delta^*(0, u) = \delta(\delta^*(0, v), a) = \delta(\sigma(v), a) = \sigma(M_{\sigma(v)}a) = \sigma(va) = \sigma(u)$$

Lemme 36.1 $\sigma(va) = \sigma(M_{\sigma(v)}a)$

Démonstration. En effet, on ne peut pour $\sigma(va)$ considérer que les $|\sigma(v)| + 1$ dernières lettres, qui sont les mêmes que celles de $M_{\sigma(v)}a$.



□

Cela conclut la récurrence.

□

Conclusion Par définition de σ , $\sigma(u) = k$ si et seulement si $M \sqsubset u$. Ainsi,

$$u \in \mathcal{L}(\mathcal{A}) \Leftrightarrow \delta^*(0, u) = k \Leftrightarrow \sigma(u) = k \Leftrightarrow M \sqsubset u \Leftrightarrow u \in \Sigma^* M$$

et donc $\mathcal{L}(\mathcal{A}) = \Sigma^* M$.

Proposition 36.2 *Aucun état de \mathcal{A} n'est inutile.*

Démonstration.

- Accessibilité : $\delta^*(0, M_i) = i$ donc tous les états sont accessibles
- Séparabilité : Soit $0 \leq i < j \leq k$. Montrons $\exists u : \begin{cases} \delta^*(i, u) \notin F \\ \delta^*(j, u) \in F \end{cases}$
 On note $N_i = m_{i+1} \dots m_k$ le suffixe de M de taille $k - i$. Prenons alors $u = N_j$.
 $\delta^*(j, N_j) = k$
 et $\delta^*(i, N_j) < k$ car $|m_1 \dots m_i m_{j+1} \dots m_k| < k$

□

Algorithme. La construction de A (et plus particulièrement de sa fonction de transition δ) n'utilise que le motif M et non le texte T . On peut représenter cette fonction via un tableau bidimensionnelle de taille $(k + 1) \times |\Sigma|$. On peut alors remplir cette table avec les différentes valeurs de σ , ce qui se fait en temps polynomial en k . Enfin, on lit le texte T lettre par lettre et si on atteint l'état k , on a trouvé une occurrence de du motif M . Si on atteint la fin de T sans jamais atteindre k , alors M n'apparaît pas dans T .

Développement 37

3-SAT est NP complet

Auteur·e·s: Emile Martinez

Références : livre de bob, cormen (pour la question à la fin)

Définition 37.1 3-SAT : $\varphi = \bigwedge_{i=1}^p C_i$ avec $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$ ($l_{i,j} \in \{x_1, \dots, x_n, \overline{x_1}, \dots, \overline{x_n}\}$)
est-elle satisfiable ?

Commentaire 37.1 l c'est pour littéral

1. 3-SAT \in NP
→ On prend comme certificat la valuation qui met φ à vrai (bien polynomial)
2. Reduction de SAT-FNC

Définition 37.2 SAT-FNC : $\varphi = \bigwedge_{i=1}^p C_i$ avec $C_i = \bigvee_{j=1}^{k_i} l_{i,j}$ ($l_{i,j} \in \{x_1, \dots, x_n, \overline{x_1}, \dots, \overline{x_n}\}$)
est-elle satisfiable ?

SAT-FNC est bien NP-complet.

3. Soit φ une instance de SAT-FNC.
Transformons la en une instance de 3-SAT.

Soit $i \in \llbracket 1, p \rrbracket$

- ★ Si $k_i = 1$, on pose $C'_i = l_{i,1} \vee l_{i,1} \vee l_{i,1}$
- ★ Si $k_i = 2$, on pose $C'_i = l_{i,1} \vee l_{i,2} \vee l_{i,2}$
- ★ Si $k_i = 3$, on pose $C'_i = C_i$
- ★ Si $k_i \geq 4$, on prend alors de nouvelles variables $y_{i,j}$ On pose alors

$$\begin{aligned} C'_i &= l_{i,1} \vee l_{i,2} \vee y_{i,2} \\ &\wedge \bigwedge_{j=3}^{k_i-2} \overline{y_{i,j-1}} \vee l_{i,j} \vee y_{i,j} \\ &\wedge \overline{y_{i,k_i-2}} \vee l_{i,k_i-1} \vee l_{i,k_i} \end{aligned}$$

Commentaire 37.2 Expliquer ici que quand un $l_{i,j}$ sera a vrai, on pourra vérifier toutes celles au dessus avec des y à 1, et toutes celles en dessous avec des y à 0. Peut être que ca serait plus clair en écrivant la formule avec des pointillées, mais c'est plus long et moins formel. On pourrait faire les deux, mais a voir le temps

On prend alors $\varphi' = \bigwedge_{i=1}^p C'_i$ qui est une instance de 3-SAT.

Remarque 37.1 C'_i ici sont des conjonctions de clauses de taille 3.

Pour chaque clause, on a rajouter au plus k_i variables et chaque nouvelle variable apparaît au plus 2 fois, chaque ancienne apparaît au plus 3 fois plus (pour $k_i = 1$). Donc notre transformation $\varphi \rightsquigarrow \varphi'$ est polynomiale.

4. Montrons que $\varphi \in \text{SAT-FNC} \Leftrightarrow \varphi' \in \text{3-SAT}$

Commentaire 37.3 Différent de $\varphi \leftrightarrow \varphi'$

\Rightarrow Supposons $\varphi \in \text{SAT-FNC}$.

Alors $\exists \sigma : V \rightarrow \{0, 1\} : [\varphi]_\sigma = 1$.

Pour chaque C_i , on choisit $n_i \in \llbracket 1, k_i \rrbracket$ tel que $[l_{i,n_i}]_\sigma = 1$.

On pose alors $\sigma'(x) = \begin{cases} 1 & \text{si } x = y_{i,j} \text{ avec } j < m_i \\ 0 & \text{si } x = y_{i,j} \text{ avec } j \geq m_i \\ \sigma(x) & \text{sinon} \end{cases}$

Montrons alors que $[C'_i]_\sigma = 1$:

★ Si $k_i \leq 3$, c'est immédiat

★ Sinon,

- $\forall j < n_i, [\overline{y_{i,j-1}} \vee l_{i,j} \vee y_{i,j}]_{\sigma'} = 1$ car $\sigma'(y_{i,j}) = 1$
- $[\overline{y_{i,n_i-1}} \vee l_{i,n_i} \vee y_{i,n_i}]_{\sigma'} = 1$ car $[l_{i,n_i}]_{\sigma'} = 1$ (choix de n_i et définition de σ')
- $\forall j > n_i, [\overline{y_{i,j-1}} \vee l_{i,j} \vee y_{i,j}]_{\sigma'} = 1$ car $\sigma'(y_{i,j-1}) = 0$
- Pareil pour les clauses extrémales.

\Leftarrow Supposons $\varphi' \in \text{3-SAT}$.

$\exists \sigma : V \rightarrow \{0, 1\} : [\varphi']_\sigma = 1$.

Alors $[\varphi]_\sigma = 1$.

Idée de la preuve : si ce n'est pas le cas, on a $[c_i]_\sigma = 0$.

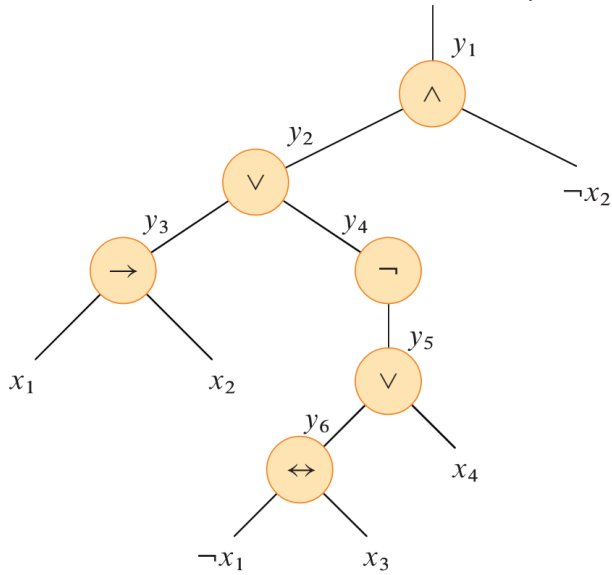
Alors comme $[C'_i]_\sigma = 1$, on a $\sigma(y_{i,2}) = 1$ donc $\sigma(y_{i,3}) = 1$, etc.

Tous les $y_{i,j}$ sont donc à vrai, donc $[\overline{y_{i,k_i-2}} \vee l_{i,k_i-1} \vee l_{i,k_i}]_\sigma = 0$ donc $[\varphi']_\sigma = 0$

Ainsi, 3-SAT est NP-complet.

Commentaire 37.4 Ce qui suit n'est à faire que suivant le temps

Question Peut-on réduire directement depuis *SAT* ?



Cette arbre représente la formule

$$\varphi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

On met alors une variable par sommet, et on transforme en la formule suivante :

$$\begin{aligned} \varphi' = & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge (y_4 \leftrightarrow \neg y_5) \\ & \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) \end{aligned}$$