Option Informatique

Classes de MP et MP*

Auteur : Marc Lorenzi
Mis à jour le 11 avril 2023

Lycée Camille Guérin

Année 2016-2017

Table des matières

Chapitre 1

Étude des algorithmes

I Notion de complexité

I.1 Qu'est-ce qu'un algorithme ?

Soient A et B deux ensembles. Soit $f : A \longrightarrow B$ une fonction. Étant donné $x \in A$, existe-t-il une méthode *effective* pour calculer $f(x)$? Pour donner un sens à cette question, il faudrait définir ce que l'on entend par caluler. Différentes définitions du concept de *fonction calculable* ont été données au début du vingtième siècle par Alan Turing (machines de Turing), Alonzo Church (lambda-calcul), Kurt Gödel (fonctions récursives) entre-autres. Toutes ces définitions, a priori très différentes, se sont avérées équivalentes. Ces modèles de calcul conduisent aux mêmes ensembles de fonctions. La *thèse de Church* dit qu'une fonction est calculable si et seulement si elle calculable par une machine de Turing, ou exprimable à l'aide d'un terme du lambda-calcul. Nous nous contenterons dans ce polycopié de la « définition » suivante :

Un **Algorithme** est une **Fonction Caml**.

I.2 Correction et Terminaison

Supposons écrite une fonction `f : 'a -> 'b`. Par terminaison et correction de `f`, il faut comprendre qu'il s'agit de savoir si pour tout `x` de type `'a`,

- L'appel `f x` termine.
- La valeur renvoyée par `f x` est bien ce que l'on attend.

Pour des fonctions sans effet de bord, cette preuve s'effectue en général par des récurrences ou des inductions structurelles. Lorsqu'il y a des effets de bords comme par exemple des références ou des tableaux modifiés dans des boucles, les démonstrations sont souvent plus compliquées. On s'appuie souvent sur ce que l'on appelle des *invariants de boucle*. Nous verrons des exemples dans la suite.

I.3 Complexité

La complexité d'un algorithme est sa propension à utiliser des ressources physiques. Des ressources typiques sont le temps et l'espace, mais cela pourrait être tout autre chose comme par exemple de l'énergie. On parle ainsi de complexité en temps ou en espace.

On peut définir proprement le concept de complexité à partir de la notion de machine de Turing, ce que nous ne ferons pas. Dans la suite, nous nous concentrerons sur la complexité en temps. Pour nous :

Complexité en temps = Nombre d'opérations

Encore faut-il préciser ce que l'on appelle opération. On se contentera de dénombrer des opérations précisées à l'avance. Quelques exemples, non exhaustifs :

- Des additions et des multiplications pour un algorithme numérique.
- Des comparaisons pour un algorithme de tri.
- Des appels à `::` pour un algorithme sur les listes.
- Le nombre d'appels récursifs pour une fonctions récursive.

Notation. On notera $C(f, x)$ la complexité de l'algorithme f appliqué à l'objet x. Lorsque l'algorithme étudié est clair, on note tout simplement $C(x)$.

Remarque 1. Bien entendu, l'algorithme f peut avoir plusieurs paramètres x, y, etc. Cela ne change rien à la discussion puisque on peut tout aussi bien considérer qu'au lieu de 2 paramètres x et y, il n'y en a qu'un : le couple $z = (x, y)$.

I.4 Complexité en moyenne, en pire cas

La complexité d'un algorithme est fonction des paramètres passés à la fonction qui le décrit. Ces paramètres possèdent la plupart du temps une *taille caractéristique*, qui est liée à la place prise par le paramètre en mémoire, commme par exemple :

- La longueur d'une liste.
- Le nombre de noeuds ou la hauteur d'un arbre.
- Le nombre d'arêtes ou de sommets d'un graphe.
- Le logarithme d'un entier n.

Notation. On note très fréquemment $|x|$ la taille de l'objet x. Si plusieurs tailles sont en jeu, on doit utiliser des notations plus précises (comme $|t|$ et $h(t)$ pour le nombre de noeuds et la hauteur de l'arbre t, ou encore $S(G)$ et $A(G)$ pour le nombre de sommets et le nombre d'arêtes du graphe G).

Bien entendu, la complexité $C(f, x)$ d'un algorithme f ne dépend pas que de la taille n de l'objet x passé en paramètre, mais de l'objet x lui-même. On peut à partir de là s'intéresser à la complexité en pire cas

$$C_n(f) = \max\{C(f, x), |x| = n\}$$

ou à la complexité en moyenne

$$C_n(f) = E(C(f, x)) = \sum_{|x|=n} p_x C(f, x)$$

où E est une espérance mathématique. Pour calculer cette espérance, il faut posséder une probabilité $\mathbb{P}$ sur l'ensemble des objets de taille n. À chaque objet x de taille n est alors associée la probabilité $p_x = \mathbb{P}(\{x\})$. Nous verrons un exemple de complexité en moyenne dans la suite. Lorsque la probabilité $\mathbb{P}$ est la probabilité uniforme, on a

$$C_n(f) = \frac{1}{N} \sum_{|x|=n} C(f, x)$$

où N est le nombre d'objets de taille n.

II Deux exemples simples

II.1 Concaténation de deux listes

On considère la fonction `join : 'a list -> 'a list -> 'a list` ci-dessous.

```
let rec join xs ys =
  match xs with
  | [] -> ys
  | x :: xs' -> x :: join xs' ys
```

Proposition 1. *Pour toutes listes xs et ys, l'appel* `join xs ys` *termine et renvoie la concaténation de xs et ys en $|xs|$ appels à* `::`.

Démonstration. Montrons par induction structurelle sur xs que `join` termine et calculons en même temps sa complexité $C(xs, ys)$, qui sera le nombre d'appels à `::`.

C'est évident si $xs = []$: dans ce cas c'est le premier motif du filtrage qui réussit et il n'y a aucun appel à `::`. On a donc $C([], ys) = 0 = |[]|$.

Soit maintenant xs une liste telle que tout appel `join xs ys` réussit et renvoie la concaténation des listes `xs` et `ys` en $|xs|$ appels à `::`. Soit x un objet. Un appel à `join (x :: xs) ys` fait réussir le second motif du filtrage. L'appel récursif à `join xs ys`, par l'hypothèse d'induction, termine en $|xs|$ itérations et renvoie la concaténation de xs et ys. De plus,

$$C(x :: xs, ys) = 1 + C(xs, ys) = 1 + |xs| = |x :: xs|$$

Soit $xs = [x_1; \ldots; x_n]$. Soit $ys = [y_0; \ldots; y_m]$. Un appel à `join (x::xs) ys` induit un appel récursif à `join xs ys` qui par l'hypothèse de récurrence renvoie la concaténation $[x_1; \ldots; x_n; y_0; \ldots; y_m]$. Puis `x :: join xs ys` renvoie $[x_0; x_1; \ldots; x_n; y_0; \ldots; y_m]$ qui est bien la concaténation de xs et ys. □

II.2 Renversement d'une liste

On considère la fonction `reverse : 'a list -> 'a list` ci-dessous.

```
let rec reverse xs =
  match xs with
  | [] -> []
  | x :: xs' -> join (reverse xs') [x]
```

Montrons la terminaison et la correction de `reverse` par récurrence sur la longueur de `xs`. Si `xs` est vide, alors `reverse xs = []`, qui est bien la valeur attendue. Supposons maintenant que `reverse` renvoie la valeur correcte pour toutes les listes de taille n. Soit $xs = [x_0; \ldots; x_n]$ une liste de taille $n + 1$. L'appel `reverse xs` unifie $x = x_0$ et $xs' = [x_1; \ldots; x_n]$. d'après l'hypothèse de récurrence, *reverse* $xs' = [x_n; \ldots; x_1]$ puis l'appel à `join` renvoie $[x_n; \ldots; x_1; x_0]$ qui est bien la valeur attendue.

Intéressons-nous maintenant à la complexité de `reverse`, en fonction du nombre d'appels à `::`. On a $C(\texttt{reverse}, []) = 0$ et

$$\begin{aligned} C(\texttt{reverse}, x :: xs') &= C(\texttt{reverse}, xs') + C(\texttt{join}, \texttt{reverse}\ xs', [x]) \\ &= C(\texttt{reverse}, xs') + |\texttt{reverse}\ xs'| \\ &= C(\texttt{reverse}, xs') + |xs'| \end{aligned}$$

Une récurrence sur $n = |xs|$ montre que pour les listes xs de taille n,

$$C(\texttt{reverse}, xs) = n(n-1)/2$$

On voit que cette complexité n'est pas bonne. Pour renverser efficacement une liste, il faut adopter une autre approche.

```
let rec rev' xs ys =
  match xs with
  | [] -> ys
  | x :: xs' -> rev' xs' (x :: ys)

let reverse2 xs = rev' xs []
```

Par une récurrence sur la longueur des listes, on montre que `rev' xs ys = join (reverse xs) ys`. Donc, `reverse2 xs = join (reverse xs) [] = reverse xs`. Maintenant,

$$\mathcal{C}(\texttt{rev}', [], ys) = 0$$

et

$$\mathcal{C}(\texttt{rev}', x :: xs', ys) = 1 + \mathcal{C}(\texttt{rev}', xs', ys)$$

Par une récurrence immédiate sur la longueur des listes, on obtient que

$$\mathcal{C}(\texttt{rev}', xs, ys) = |xs|$$

Ainsi,

$$\mathcal{C}(\texttt{reverse2}, xs) = |xs|$$

III Exponentiation rapide

III.1 Algorithme fonctionnel

On considère la fonction `power : int -> int -> int` ci-dessous.

```
let rec power x n =
    if n = 0 then 1
    else
        let y = power (x * x) (n / 2) in
        if n mod 2 = 0 then y
        else x * y;;
```

III.2 Terminaison et preuve

un appel à `power x n` ne renvoie pas x^n pour tous les entiers x et n : en effet, le type `int` est un sous ensemble de l'ensemble des entiers relatifs, et des débordements se produisent pour des valeurs trop grandes de n. Nous ne nous occuperons pas de ce problème et nous faisons dans la suite comme si aucun débordement ne se produisait.

Proposition 2. *Pour tous x, n de type* `int`*, l'appel* `power` x n *renvoie* x^n .

Démonstration. On fait une récurrence forte sur n. □

III.3 Complexité

Notons $\mathcal{C}(x, n)$ le nombre de multiplications effectuées lors du calcul de `power x n`. On a $\mathcal{C}(x, 0) = 0$. Pour tout $p \geq 0$, on a $\mathcal{C}(x, 2p + 1) = 2 + \mathcal{C}(x, p)$ et pour tout $p \geq 1$, $\mathcal{C}(x, 2p) = 1 + \mathcal{C}(x, p)$. La quantité $\mathcal{C}(x, n)$ ne dépend « clairement » pas de x (récurrence !), nous la noterons $\mathcal{C}(n)$ dans la suite.

Proposition 3. *Il existe deux réels c et d tels que $\forall n \geq 1, \mathcal{C}(n) \leq c \lg n + d$, où* lg *désigne le logarithme en base 2.*

Démonstration. On a $C(1) = 2$. Donc tout réel c et tout réel $d \geq 2$ conviennent pour $n = 1$. Soit maintenant un entier $n \geq 1$. Supposons trouvés c et d tels que $C(k) \leq c \lg k + d$ pour tout entier $k \leq n$. Trouvons une condition suffisante sur c et d pour que l'inégalité soit encore vérifiée pour $n + 1$. Supposons $n = 2p + 1$ avec $p \geq 0$. Alors

$$C(n + 1) = C(2p + 2) = 1 + C(p + 1) \leq 1 + c \lg(p + 1) + d = c \lg(n + 1) + d - c + 1$$

Condition suffisante : $d - c + 1 \leq d$, c'est à dire $c \geq 1$. Supposons maintenant $n = 2p$ avec $p \geq 1$. Alors

$$C(n+1) = C(2p+1) = 2+C(p+1) \leq 2+c \lg(p)+d \leq c \lg(2p)+d-c+2 \leq c \lg(n+1)+d-c+2$$

Condition suffisante : $d-c+2 \leq d$, c'est à dire $c \geq 2$. Conclusion, si l'on prend $c = d = 2$ l'inégalité désirée est vraie pour tout entier $n \geq 1$. En conclusion,

$$\forall n \geq 1, C(n) \leq 2 \lg n + 2$$

□

Exercice. Montrer que pour tout entier $n \geq 1$, on a $\mathcal{C}(n) = \mu(n) + \gamma(n)$ où $\mu(n) = \lfloor \lg n \rfloor + 1$ est le nombre de chiffres de l'écriture de n en base 2 et $\gamma(n)$ est la somme des chiffres de cette même écriture. Faire une récurrence forte.

III.4 Algorithme impératif

Voici une fonction d'exponentiation rapide écrite dans un style impératif.

```
let pow x n =
    let z = ref 1
    and m = ref n
    and y = ref x in
    while !m <> 0 do
        if !m mod 2 = 1 then z := !z * !y ;
```

```
        m := !m / 2 ;
        y := !y * !y
    done;
    !z;;
```

III.5 Terminaison, complexité, correction

Supposons $n \geq 1$. Il existe un unique entier N tel que $2^N \leq n < 2^{N+1}$. Cet entier est $N = \lfloor \lg n \rfloor$. On obtient par une récurrence sur k qu'après k itérations, $0 \leq k \leq N$, $2^{N-k} \leq m < 2^{N+1-k}$. Pour $k = N$, on obtient alors $1 \leq m < 2$, c'est-à-dire $m = 1$. Encore une itération et $m = 0$. Ainsi, le nombre $I(n)$ d'itérations de la boucle while vaut $I(n) = 1 + \lfloor \lg n \rfloor$. L'algorithme termine en $I(n)$ itérations.

Soit $C(n)$ le nombre de multiplications effectuées lors de l'appel à `pow x n`. On a à chaque itération une ou deux multiplications, selon la parité de m. Ainsi, comme il y a $1 + \lfloor \lg n \rfloor$ itérations, on a

$$1 + \lfloor \lg n \rfloor \leq C(n) \leq 2 + 2\lfloor \lg n \rfloor$$

On retrouve le même majorant que pour la fonction récursive.

Il reste à prouver la correction. On va montrer que la quantité zy^m est un *invariant de boucle*. Soient z, y, m les valeurs des variables au début d'une itération, et z', y', m' leurs valeurs à la fin de cette même itération. Deux cas se présentent :

- $m = 2p$ est pair, non nul. Alors $z' = z$, $y' = y^2$ et $m' = p$. On en tire $z'y'^{m'} = z(y^2)^p = zy^m$.
- $m = 2p + 1$ est impair. Alors $z' = zy$, $y' = y^2$ et $m' = p$. On en tire $z'y'^{m'} = zy(y^2)^p = zy^{2p+1} = zy^m$.

En entrée de boucle, on a $zy^m = 1.x^n = x^n$. En sortie de boucle, on a $zy^m = zy^0 = z$. Par l'invariance, il vient en sortie de boucle $z = x^n$.

IV Tri rapide

IV.1 Algorithme

```
let rec partition pivot xs =
    match xs with
    | []    -> ([],[])
    | x::xs' ->
        let (ys, zs) = partition pivot xs' in
        if x <= pivot then (x::ys, zs)
                      else (ys, x::zs);;

let rec quicksort xs =
    match xs with
    | []    -> []
    | x::xs' ->
        let (ys, zs) = partition x xs' in
        quicksort ys @ x :: quicksort zs;;
```

IV.2 Analyse de la fonction de partition

Nous admettons le résultat suivant, qui se prouve par récurrence forte sur la longueur de la liste xs : `partition pivot s` termine après $|xs|$ appels récursifs et $|xs|$ comparaisons d'éléments de xs. La valeur renvoyée est un couple (xs_1, xs_2) où xs_1 contient les éléments de xs inférieurs ou égaux à `pivot`, xs_2 contient les éléments de xs strictement supérieurs à `pivot`, les éléments de xs_1 et xs_2 apparaissant dans le même ordre que leur ordre d'apparition dans xs.

IV.3 Analyse en moyenne du tri rapide

Le tri rapide d'une liste de taille n commence par partitionner la liste en prenant comme pivot sa tête, puis trie récursivement deux sous-listes. Les tailles respectives de ces sous-listes ont pour valeurs possibles $(0, n-1)$, $(1, n-2)$,... ,$(n-2, 1)$, $(n-1, 0)$. En faisant l'hypothèse que ces tailles sont équiprobables, on en déduit que le nombre moyen de comparaisons C_n effectuées par le tri rapide pour trier une liste de taille $n \geq 1$ est :

$$C_n = n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}(C_k + C_{n-1-k}) = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} C_k$$

avec de plus $C_0 = C_1 = 0$. Il reste à résoudre cette récurrence.

Pour $n \geq 2$ on a

$$nC_n - (n-1)C_{n-1} = 2(n-1) + 2C_{n-1}$$

d'où

$$nC_n = 2(n-1) + (n+1)C_{n-1}.$$

Posons pour $n \geq 0$ $D_n = \frac{C_n}{n+1}$. Il vient pour $n \geq 2$,

$$D_n = 2\frac{n-1}{n(n+1)} + D_{n-1}$$

Remarquons que cette égalité est vraie pour $n = 1$. D'où, pour tout $n \geq 1$:

$$D_n = D_0 + 2\sum_{k=1}^{n}\frac{k-1}{k(k+1)}.$$

On a

$$\frac{k-1}{k(k+1)} = -\frac{1}{k} + \frac{2}{k+1}.$$

On en tire, puisque $D_0 = 0$, que

$$D_n = \frac{2}{n+1} + \sum_{k=1}^{n}\frac{1}{k} - 2.$$

Cette quantité étant équivalente à $\ln n$, on a donc

$$C_n \sim n \ln n.$$

IV.4 Analyse en pire cas du tri rapide

Notons T_n la complexité du tri rapide en pire cas pour une liste de longueur n.

- Notons C_n le nombre de comparaisons nécessaires au tri d'une liste déjà triée. Prenons une liste `s=x::xs` triée. Supposons pour simplifier que les éléments de la liste sont tous distincts. La fonction de partition appelée sur `xs` avec pour pivot `x` renvoie le couple `([],xs)` . La partition effectue $n-1$ comparaisons. Puis, le tri rapide est appelé sur `[]` (0 comparaison), et `xs`, de taille $n-1$. Ainsi, $C_n = n-1+C_{n-1}$. D'où :

  $$C_n = \frac{n(n-1)}{2} \sim \frac{1}{2}n^2$$

  On en déduit que $T_n \geq \frac{n(n-1)}{2}$.
- Montrons maintenant que, pour toute liste xs de longueur n, on a $C(xs) \leq cn^2$ pour une constante c convenable. On a $C(xs) = n-1+C(xs_1)+C(xs_2)$ où xs_1 et xs_2, résultats de la partition de xs, sont de tailles respectives k et $n-1-k$ avec $0 \leq k \leq n-1$. Ainsi,

  $$C(xs) \leq n-1+T_k+T_{n-1-k} \leq \max_{0\leq k\leq n-1}(T_k+T_{n-1-k})$$

  Ceci étant vrai pour toute liste xs de taille n, on en déduit pour tout $n \geq 1$

  $$T_n \leq n-1+\max_{0\leq k\leq n-1}(T_k+T_{n-1-k})$$

  Pour $n=0$, n'importe quel réel c convient. Supposons trouvé un réel positif c convenant pour tout entier k strictement inférieur à l'entier $n>0$. On a alors

  $$T_n \leq n-1+c\max_{0\leq k\leq n-1}(k^2+(n-1-k)^2)$$

  On vérifie facilement que la fonction $\varphi : [0, n-1] \to R$ définie par $\varphi(x) = x^2+(n-1-x)^2$ atteint son maximum aux bornes de son intervalle de définition, ce maximum étant égal à $(n-1)^2$. On en déduit que

  $$T_n \leq n-1+c(n-1)^2$$

  On vérifie alors que, pourvu que $c \geq \frac{1}{2}$, on aura bien $T(n) \leq cn^2$. Ainsi, $T(n) \leq \frac{1}{2}n^2$.

En conclusion, le tri rapide est rapide en moyenne, mais a une mauvaise complexité dans certains cas. On peut montrer que ces cas sont rares d'un point de vue probabiliste (l'écart type du nombre de comparaisons est négligeable devant sa moyenne). De plus, il existe des méthodes simples pour éviter les cas pathologiques, comme par exemple d'échanger le premier élément de la liste à trier (le pivot) avec un élément au hasard.

V Multiplication rapide de matrices

Prenons deux matrices carrées $n \times n$ A et B. Le calcul naïf du produit $C = AB$ nécessite celui des n^2 coefficients de la matrice C. Pour chacun d'entre eux, il faut effectuer n multiplications et $n-1$ additions, ce qui donne en fin de commpte un algorithme en $O(n^3)$. Il est possible de faire mieux. Nous allons examiner un algorithme plus efficace permettant de multiplier deux matrices carrées dont la taille est une puissance de 2 (dans le cas général, l'algorithme qui suit peut être adapté).

- Soient, pour commencer, deux matrices carrées 2×2 A et B définies par

$$A = \begin{pmatrix} a & c \\ b & d \end{pmatrix} ; \; B = \begin{pmatrix} a' & c' \\ b' & d' \end{pmatrix}$$

On calcule les quantités suivantes :

$$\begin{cases} m_1 = (b + d - a)(d' - c' + a') \\ m_2 = aa' \\ m_3 = cb' \\ m_4 = (a - b)(d' - c') \\ m_5 = (b + d)(c' - a') \\ m_6 = (c - b + a - d)d' \\ m_7 = d(a' + d' - b' - c') \end{cases}$$

On a alors

$$AB = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}$$

Il est ainsi possible de multiplier deux matrices 2×2 en utilisant 7 multiplications et 24 additions.

Exercice : Montrer que 15 additions sont suffisantes.

- Voyons en quoi ceci est une amélioration de l'algorithme naïf. Le calcul ci-dessus peut s'appliquer à des multiplications par blocs. Pour multiplier deux matrices $2n \times 2n$, on les coupe chacune en 4 blocs $n \times n$ et on utilise les formules ci-dessus, récursivement bien sûr. Il est enfin clair que deux matrices 1×1 se multiplient avec 1 multiplication scalaire et 0 addition scalaire. Notons $M(n)$ et $A(n)$ respecivement le nombre de multiplications et d'additions scalaires requises pour multiplier deux matrices carrées de taille 2^n. On a $M(0) = 1$ et $A(0) = 0$. De plus, on a les relations de récurrence

$$M(n) = 7M(n-1)$$

$$A(n) = 7A(n-1) + 15.2^{2(n-1)}$$

La relation sur M est claire. En ce qui concerne $A(n)$, On a d'une part les additions cachées dans l'appel récursif, et d'autre part les sommes de matrices de taille $2^n/2 = 2^{n-1}$ qui sont effectuées classiquement et demandent donc pour chacune $(2^{n-1})^2$ additions scalaires.
On obtient pour tout n positif $M(n) = 7^n$. Un calcul « en cascade » fournit

$$A(n) = 5 \times 7^n - 5 \times 4^n$$

N'oublions pas que ces calculs concernent des matrices de taille 2^n. Pour un produit de matrices de taille n, on obtient, en remplaçant n par $\lg n$ dans les expressions ci-dessus, un nombre de multiplications scalaires égal à $n^{\lg 7}$ et un nombre d'additions scalaires égal à $5n^{\lg 7} - 5n^2$. Sachant que $\lg 7 \simeq 2.8$, nous avons donc un algorithme dont le nombre d'opérations est négligeable par rapport à l'algorithme naïf.

Revenons un instant sur le titre du paragraphe. La multiplication rapide de matrices est-elle rapide ? Nous allons comparer brièvement le nombre total d'opérations (addition, multiplication) effectuées d'une part par l'algorithme naïf, d'autre part par l'algorithme que nous venons d'étudier. On a :

- $6n^{\lg 7} - 5n^2$ opérations pour l'algorithme rapide.

- $2n^3 - n^2$ opérations pour l'algorithme naïf.

On cherche donc à savoir lorsque $6n^{\lg 7} - 5n^2 \leq 2n^3 - n^2$. Ceci est le cas pour $n \geq 290$. L'algorithme rapide n'a donc d'intérêt que pour des grosses matrices. Signalons en plus les multiples appels récursifs, consommateurs de temps et de mémoire, qui n'amélioreront sûrement pas le « vrai » temps d'exécution (c'est à dire celui pendant lequel on attend devant sa machine que le calcul veuille bien s'effectuer).

On pourrait reprocher au calcul précédent d'attribuer un même temps d'exécution aux additions et aux multiplications. Le reproche est parfois justifié.

Exercice. Refaire le calcul en en attribuant un poids 2 aux multiplications. On obtiendra que la multiplication rapide devient intéressante en nombre d'opérations pour une taille de matrice supérieure à 29. C'est mieux. Dans la pratique, on met en oeuvre un algorithme mixte, où l'algorithme naïf prendrait le relais lorsque la taille des matrices devient inférieure à une certaine taille.

VI Une récurrence classique

Nous étudions ici la récurrence

$$T_n = aT_{\frac{n}{2}} + O(n^\alpha)$$

où $a > 0, \alpha \geq 0$. Elle intervient dans de nombreux algorithmes du type *diviser pour régner* : exponentiation rapide, tri par fusion, algorithme de Karatsuba pour le produit de deux polynômes, multiplication rapide de matrices, etc. On se contente de regarder ce qui se passe lorsque n est une puissance de 2 et $O(n^\alpha) = bn^\alpha$ où $b > 0$.

Remarque 2. Dans le cas général où n n'est pas une puissance de 2, ce n'est pas $\frac{n}{2}$ qui intervient dans la relation de récurrence mais $\lfloor \frac{n}{2} \rfloor$ ou $\lceil \frac{n}{2} \rceil$, voire ces deux quantités simultanément. Par exemple, la relation récurrence pour la complexité du tri par fusion est $T_n = T_{\lfloor \frac{n}{2} \rfloor} + T_{\lceil \frac{n}{2} \rceil} + O(n)$, qui se réduit à $T_n = 2T_{\frac{n}{2}} + O(n)$ lorsque n est une puissance de 2. Nous admettrons que les ordres de grandeur de la complexité sont conservés dans le cas général. Nous posons donc $u_n = T_{2^n}$, de sorte que $T_n = u_{\lg n}$ et $u_n = au_{n-1} + b(2^\alpha)^n$. L'écriture d'égalités en cascade et leur combinaison fournit

$$u_n = a^n(u_0 - b) + b\sum_{k=0}^{n} a^k(2^\alpha)^{n-k}$$

VI.1 Le cas $a = 2^\alpha$

On a ici

$$u_n = a^n(u_0 - b) + b(n+1)(2^\alpha)^n = (2^n)^\alpha(u_0 + nb) \sim nb(2^n)^\alpha$$

$$T_n \sim bn^\alpha \lg n$$

VI.2 Le cas $a \neq 2^\alpha$

On a ici

$$u_n = a^n(u_0 - b) + b\frac{a^{n+1} - (2^\alpha)^{n+1}}{a - 2^\alpha} = Aa^n + B(2^\alpha)^n$$

où

$$A = u_0 + \frac{b2^\alpha}{a - 2^\alpha}$$

$$B = \frac{b}{2^\alpha - a}$$

On peut encore écrire

$$u_n = A(2^n)^{\lg a} + B(2^n)^\alpha$$

Ainsi,

- Si $a > 2^\alpha$, $u_n \sim A(2^n)^{\lg a}$ et $T_n \sim An^{\lg a}$
- Si $a < 2^\alpha$, $u_n \sim B(2^n)^\alpha$ et $T_n \sim Bn^\alpha$.

VI.3 Exemples

- L'exponentiation rapide : $a = 1$, $\alpha = 0$. On a $T_n = O(\lg n)$.
- Le tri par fusion : $a = 2$, $\alpha = 1$. On a $T_n = O(n \lg n)$.
- Produit rapide de matrices : $a = 7$, $\alpha = 2$. On a $T_n = O(n^{\lg 7})$.
- Algorithme de Karatsuba : $a = 3$, $\alpha = 1$. On a $T_n = O(n^{\lg 3})$.

Chapitre 2

Arbres

I De quoi allons-nous parler ?

De façon informelle, un arbre est une structure comportant des *noeuds* et des *feuilles*. Chaque noeud de l'arbre possède un ou plusieurs fils, qui sont eux-mêmes des noeuds ou des feuilles. Les feuilles de l'arbre sont les noeuds qui n'ont pas de fils. Un noeud particulier, la *racine*, n'est le fils de personne.

Les arbres sont présents, de façon implicite ou explicite, dans un grand nombre de problèmes : arbres de recherche, permettant d'implémenter la structure de dictionnaire, tas, reliés à la notion de file de priorité, arbres couvrants dans les graphes, etc.

Nous nous consacrerons presque exclusivement dans ce chapitre aux arbres binaires, c'est à dire aux arbres dont chaque noeud possède exactement deux fils.

Définissons en Caml un type arbre dont les noeuds sont de type `'a` et les feuilles de type `'b` et demandons-nous ce que nous venons de faire :

```
type ('a, 'b) arbre =
|  Feuille of 'b
|  Noeud of ('a, 'b) arbre * 'a * ('a, 'b) arbre
```

Qu'entendons-nous exactement par « arbre » ?

- Si f est un objet de type b, alors `Feuille f` est un arbre.
- Si x est un objet de type a et t_1 et t_2 sont deux arbres, alors `Noeud (t1, x, t2)` est un arbre.

Mais aussi : si t est un arbre, alors

- Ou bien il existe un unique objet f de type b, tel que `t = Feuille f`.
- Ou bien il existe un unique objet x de type a, un unique arbre t_1 et un unique arbre t_2 tels que `t = Noeud (t1, x, t2)`.

Il semble qu'il manque quelque chose : par exemple, rien de ce que nous avons dit jusqu'à présent n'interdit d'avoir des branches de longueur infinie ! Nous allons imposer à notre concept d'arbre une dernière propriété :

L'ensemble des arbres devrait être le plus petit ensemble vérifiant tout ce que nous venons de dire.

II Généralités

II.1 Notion d'arbre

Définition 1. On suppose donnés un ensemble $\mathcal{F}$ dont les éléments sont appelés *feuilles* et un ensemble $\mathcal{N}$ dont les éléments sont appelés *noeuds*. On définit une suite croissante d'ensembles en posant

- $\mathcal{T}_0 = \mathcal{F}$.

- Pour tout entier $n \in \mathbb{N}$, $\mathcal{T}_{n+1} = \mathcal{T}_n \cup (\mathcal{T}_n \times \mathcal{N} \times \mathcal{T}_n)$.

On pose enfin

$$\mathcal{T} = \bigcup_{n \geq 0} T_n$$

et on appelle arbre (binaire, à noeuds dans $\mathcal{N}$ et à feuilles dans $\mathcal{F}$) tout élément de $\mathcal{T}$.

Informellement, un arbre est soit une feuille, soit un noeud avec deux fils, un gauche et un droit, qui sont eux-mêmes des arbres. L'ensemble des arbres est en fait le plus petit ensemble d'objets vérifiant cela :

Proposition 1. *L'ensemble $\mathcal{T}$ est le plus petit ensemble vérifiant :*

1. $\mathcal{F} \subseteq \mathcal{T}$.
2. *Pour tous $t_1, t_2 \in \mathcal{T}$ et tout $x \in \mathcal{N}$, on a $(t_1, x, t_2) \in \mathcal{T}$.*

Démonstration. L'ensemble $\mathcal{T}$ vérifie la propriété 1. Montrons qu'il vérifie 2 : soient t_1 et t_2 deux arbres. Il existe deux entiers m et n tels que $t_1 \in \mathcal{T}_m$ et $t_2 \in \mathcal{T}_n$. Prenons pour fixer les idées $m \leq n$. On a alors $t_1, t_2 \in \mathcal{T}_n$ donc $t \in \mathcal{T}_{n+1} \subseteq \mathcal{T}$.

Montrons maintenant que l'ensemble $\mathcal{T}$ est inclus dans tous les ensembles vérifiant 1 et 2. Soit donc $\mathcal{U}$ un ensemble vérifiant 1 et 2. Montrons par récurrence sur n que pour tout entier n, $\mathcal{T}_n \subseteq \mathcal{U}$. C'est clair pour $n = 0$ d'après le point 1. Soit maintenant $n \in \mathbb{N}$. Supposons que $\mathcal{T}_n \subseteq \mathcal{U}$. Soit $t \in \mathcal{T}_{n+1}$. Étant donné que $\mathcal{T}_{n+1} = \mathcal{T}_n \cup (\mathcal{T}_n \times \mathcal{N} \times \mathcal{T}_n)$, on a soit $t \in \mathcal{T}_n$, et donc $t \in \mathcal{U}$, soit $t \in \mathcal{T}_n \times \mathcal{N} \times \mathcal{T}_n$ et donc $t = (t_1, x, t_2)$ où $t_1, t_2 \in \mathcal{T}_n$ et $x \in \mathcal{N}$. Mais par l'hypothèse de récurrence, $t_1, t_2 \in \mathcal{U}$ et comme $\mathcal{U}$ vérifie le point 2, on a bien $t \in \mathcal{U}$.

L'ensemble $\mathcal{T}$ étant la réunion des $\mathcal{T}_n$ qui sont tous inclus dans $\mathcal{U}$, on a donc finalement $\mathcal{T} \subseteq \mathcal{U}$. □

Remarque 1. Le théorème précédent nous dit entre autre que si on combine des arbres et des noeuds d'une certaine façon, on obtient encore des arbres. La question de la réciproque se pose : tout arbre est il soit une feuille, soit une combinaison d'arbres « plus simples » ? Oui, à condition de préciser le concept de simplicité, c'est l'objet du paragraphe suivant.

II.2 Mesures de la taille d'un arbre

Notation. Pour tout arbre t, on note $|t|$ le nombre de noeuds de t, et $||t||$ le nombre de feuilles de t.

Les deux nombres ci-dessus sont deux mesures possibles de la simplicité d'un arbre : moins un arbre a de noeuds ou de feuilles, moins il est compliqué. Les arbres les plus simples sont ceux n'ayant aucun noeud, et une unique feuille. Un autre indicateur essentiel de la complexité d'un arbre est sa *hauteur*.

Définition 2. Soi $t \in \mathcal{T}$. On appelle hauteur de t l'entier $h(t) = \min\{h \in \mathbb{N}, t \in \mathcal{T}_h\}$.

Exemple. Un arbre est de hauteur 0 si et seulement si c'est une feuille. L'arbre $(((f, 3, f), 5, f), 9, (f, 1, f))$ est de hauteur 3 (enfin, il est dans $\mathcal{T}_3$ en on n'arrive pas à voir comment il pourrait être dans $\mathcal{T}_2$).

II.3 Lecture unique

Théorème 2. *[Lecture unique] Soit $t \in \mathcal{T}$. On est dans un, et un seul, des deux cas suivants :*

1. *$t \in \mathcal{F}$.*
2. *Il existe un unique $t_1 \in \mathcal{T}$, un unique $t_2 \in \mathcal{T}$ et un unique $x \in \mathcal{N}$ tels que $t = (t_1, x, t_2)$.*

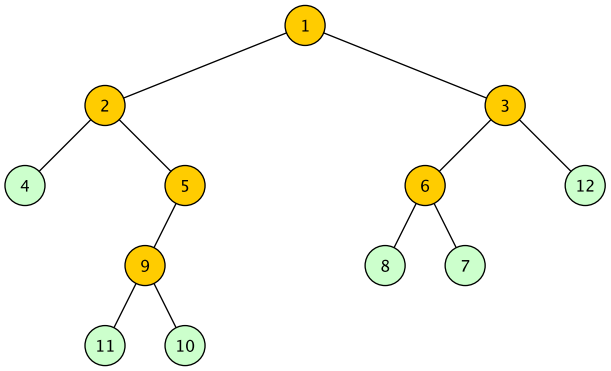
Démonstration. Les deux cas s'excluent évidemment. Un arbre de hauteur 0 est une feuille, et vérifie donc le point 1. Soit maintenant un arbre t de hauteur $h > 0$. On a $t \in \mathcal{T}_h \setminus \mathcal{T}_{h-1}$. C'est à dire que t est dans $\mathcal{T}_{h-1} \times \mathcal{N} \times \mathcal{T}_{h-1}$. Il existe donc deux arbres $t_1, t_2 \in \mathcal{T}_{h-1}$ et $x \in \mathcal{N}$ tels que $t = (t_1, x, t_2)$. L'unicité est quant à elle évidente puisque deux triplets sont égaux si et seulement si ils ont les mêmes composantes. □

Définition 3.

- Pour tout arbre $t = (t_1, x, t_2)$ qui n'est pas une feuille, t_1 est le fils gauche de t, t_2 est le fils droit de t et x est la racine de t.
- Une feuille n'a ni fils gauche, ni fils droit. Selon les conventions adoptées et selon le contexte, on pourra envisager qu'elle a une unique racine, elle-même, ou pas de racine.

II.4 Représentation graphique

Maintenant que nous avons existence et unicité de tout, cela ne pose plus de problème de représenter graphiquement les arbres. La racine est placée en haut, on tire un trait oblique vers le fils gauche et le fils droit. Les noeuds et les feuilles sont représentés par leur valeur, éventuellement entourés et coloriés pour faire joli. Ci-dessous, les feuilles sont vertes et les noeuds sont jaunes.



II.5 Retour sur la hauteur

Proposition 3. *Soit t un arbre.*

- *Si t est une feuille alors $h(t) = 0$.*

- *Si $t = (t_1, x, t_2)$ est un arbre de hauteur strictement positive, alors*

$$h(t) = 1 + \max(h(t_1), h(t_2))$$

Démonstration. Seul le second point mérite que l'on s'y attarde. Soient m et n les hauteurs de t_1 et t_2, et h la hauteur de t. Supposons par exemple $m \leq n$. On a $t_1, t_2 \in \mathcal{T}_n$, donc $t \in \mathcal{T}_{n+1}$ et donc $h(t) \leq n + 1 = 1 + max(m, n)$.

Inversement, on a $t \in \mathcal{T}_h \backslash \mathcal{T}_{h-1}$ donc $t = (t_1, x, t_2) \in \mathcal{T}_{h-1} \times \mathcal{N} \times \mathcal{T}_{h-1}$. On en déduit que $t_1, t_2 \in \mathcal{T}_{h-1}$ donc $m, n \leq h - 1$ et ainsi $\max(m, n) \leq h - 1$ d'où l'autre inégalité. □

II.6 Fonctions Caml élémentaires

```
let racine t =
  match t with
  | Feuille _ -> failwith "racine"
  | Noeud (_, x, _) -> x
```

Exercice. Écrire les fonctions fils gauche et fils droit.

```
let rec nombre_noeuds t =
  match t with
  | Feuille _ -> 0
  | Noeud (t1, _, t2) -> 1 + nombre_noeuds t1 + nombre_noeuds t2

let rec nombre_feuilles t =
  match t with
  | Feuille _ -> 1
  | Noeud (t1, _, t2) -> nombre_feuilles t1 + nombre_feuilles t2

let rec hauteur t =
  match t with
  | Feuille _ -> 0
  | Noeud (t1, _, t2) -> 1 + max (nombre_noeuds t1) (nombre_noeuds t2)
```

Remarque 2. Les trois dernières fonctions se ressemblent étrangement. Voir plus loin pourquoi et comment.

II.7 Démontrer des propriétés des arbres

Pour démontrer une propriété relative aux arbres, on peut bien entendu faire des récurrences sur le nombre de noeuds ou sur la hauteur. Il existe une méthode plus élégante, qui ressemble à une récurrence, et qui repose sur la structure même des arbres : c'est la démonstration par *induction structurelle*.

Théorème 4. *[Induction structurelle] Soit $\mathcal{P}(t)$ une propriété relative aux arbres. On suppose que*

- *Pour toute feuille f, on a $\mathcal{P}(f)$.*

- *Pour tous arbres t_1 et t_2, pour tout noeud x, si on a $\mathcal{P}(t_1)$ et $\mathcal{P}(t_2)$, alors on a $\mathcal{P}((t_1, x, t_2))$.*

Alors on a $\mathcal{P}(t)$ pour tous les arbres t.

Démonstration. On fait une démonstration par récurrence sur la hauteur des arbres. Montrons précisément par récurrence forte sur n que pour tout entier $n \in \mathbb{N}$, pour tout arbre t de hauteur n, on a $\mathcal{P}(t)$.

Pour $n = 0$ c'est clair : c'est l'hypothèse 1 du théorème. Soit maintenant $n \geq 1$. Supposons la propriété vraie pour tous les arbres de hauteur strictement inférieure à n. Soit $t = (t_1, x, t_2)$ un arbre de hauteur n. t_1 et t_2 sont deux arbres de hauteur inférieure ou égale à $n - 1$, ils vérifient donc $\mathcal{P}$. D'après l'hypothès 2 du théorème, on a donc $\mathcal{P}(t)$. □

Regardons quelques exemples sur lequels nous mettrons en oeuvre l'induction structurelle.

Proposition 5. *Pour tout arbre t, on a $||t|| = 1 + |t|$*

Démonstration. On rappelle que $|t|$ est le nombre de noeuds de t et $||t||$ est le nombre de feuilles de t. Pour ceux qui se demandaient pourquoi je n'avais pas parlé de récurrence sur le nombre de feuilles, la réponse est maintenant évidente : autant récurrer sur le nombre de noeuds, ce qui ne change rien. Bref, démontrons l'galité par induction structurelle.

Si t est une feuille, $||t|| = 1 = 1 + 0 = 1 + |t|$.

Soient t_1 et t_2 tels que $||t_i|| = 1 + |t_i|$ pour $i = 1, 2$. Soit x un noeud. Soit $t = (t_1, x, t_2)$. On a $||t|| = ||t_1|| + ||t_2||$ et $|t| = 1 + |t_1| + |t_2|$ d'où le résultat cherché. □

Définition 4. Étant donnés un arbre t et un noeud ou une feuille x de t, on appelle profondeur de x la distance de x à la racine de t. Ainsi, la racine a pour profondeur 0, les fils de la racine ont pour profondeur 1, etc. On note $p(x, t)$ la profondeur de x dans t.

Proposition 6. *[Égalité de Kraft] On a, pour tout arbre t,*

$$\sum_{f \in \mathcal{F}(t)} 2^{-p(f,t)} = 1$$

où $\mathcal{F}(t)$ désigne l'ensemble des feuilles de t.

Démonstration. Notons, pour tout arbre t, $\varphi(t) = \sum_{f \in \mathcal{F}(t)} 2^{-p(f,t)}$. Montrons par induction structurelle que pour tout arbre t on a $\varphi(t) = 1$. C'est évident si t est une feuille f_0 :$\varphi(t) = 2^{-p(f_0,f_0)} = 2^0 = 1$. Soient maintenant deux arbres t_1 et t_2. Soit $x \in \mathcal{N}$. Supposons $\varphi(t_1)$ et $\varphi(t_2) = 1$ et montrons que pour $t = (t_1, x, t_2)$, on a $\varphi(t) = 1$.

Remarquons tout d'abord que pour toute feuille f de t_1, f est une feuille de t et $p(f, t) = p(f, t_1) + 1$. Même égalité pour les feuilles de t_2. Remarquons enfin que $\mathcal{F}(t)$ est la réunion disjointe des ensembles $\mathcal{F}(t_1)$ et $\mathcal{F}(t_2)$. On a alors $\varphi(t) = \sum_{f \in \mathcal{F}(t_1)} 2^{-p(f,t)} + \sum_{f \in \mathcal{F}(t_2)} 2^{-p(f,t)} =$

$\sum_{f\in\mathcal{F}(t_1)} 2^{-p(f,t_1)+1} + \sum_{f\in\mathcal{F}(t_2)} 2^{-p(f,t_2)+1} = \frac{1}{2}\varphi(t_1) + \frac{1}{2}\varphi(t_2) = 1$. Par le théorème d'induction structurelle on a donc $\varphi(t) = 1$ pour tout arbre t. □

II.8 Définir des fonctions sur l'ensemble des arbres

Tout comme on peut démontrer des propriétés par induction structurelle, on peut aussi définir des fonctions par induction structurelle.

Théorème 7. *Soit E un ensemble non vide. Soit $f : \mathcal{F} \to E$. Soit $g : E \times \mathcal{N} \times E \to E$. Il existe une unique fonction $F : \mathcal{T} \to E$ telle que*

1. *Pour toute feuille t, $F(t) = f(t)$.*
2. *Pour tous arbres t_1, t_2 et tout noeud x, $F((t_1, x, t_2)) = g(F(t_1), x, F(t_2))$.*

Démonstration.

L'unicité, facile, est laissée en exercice. C'est une induction structurelle. Nous allons montrer l'existence et là, ce ne sera pas aussi facile. Étant donnée une fonction $F : \mathcal{T} \to E$ et un entier naturel n, nous dirons que F est une approximation à l'ordre n (de la solution de notre problème) lorsque

1. Pour toute feuille t, $F(t) = f(t)$.
2. Pour tous arbres t_1, t_2 *de hauteur strictement inférieure à* n et tout noeud x,

$$F((t_1, x, t_2)) = g(F(t_1), x, F(t_2))$$

Remarquons tout d'abord que pour tous entiers $m \leq n$, pour toutes fonctions $F, G : \mathcal{T} \to E$, si F est une approximation à l'ordre m et G est une approximation à l'ordre n, alors F et G coïncident sur $\mathcal{T}_n$. Il suffit de faire une récurrence sur la hauteur des arbres. En particulier, cela implique que s'il existe une approximation à l'ordre n, il en existe une seule (prendre $m = n$). Appelons cette propriété la *coïncidence des restrictions*.

Montrons maintenant par récurrence sur n que pour tout entier $n \in \mathbb{N}$ il existe une approximation à l'ordre n. Soit $F_0 : \mathcal{T} \to E$ définie par $\forall t \in \mathcal{F}_0(t) = f(t)$ et pour tout arbre $t \notin \mathcal{F}$, $F_0(t) = e$, où e est un élément quelconque fixé de E. Alors F_0 est clairement une approximation à l'ordre 0.

Soit maintenant $n \in \mathbb{N}$. Supposons qu'il existe une approximation F_n à l'ordre n. Soit $F_{n+1} : \mathcal{T} \to E$ définie par $\forall t \in \mathcal{T}_n, F_{n+1}(t) = F_n(t)$, $\forall t = (t_1, x, t_2) \in \mathcal{T}_{n+1} \setminus \mathcal{T}_n, F_{n+1}(t) = g(F_n(t_1), x, F_n(t_2))$ et pour tout arbre t de hauteur strictement supérieure à $n+1$, $F_{n+1}(t) = e$, ce fameux élément de E fixé quelconque. On vérifie facilement par récurrence (finie) sur la hauteur que F_{n+1} est une approximation à l'ordre $n+1$.

Nous avons donc démontré l'existence, pour tout entier n, d'une unique approximation F_n à l'ordre n. Soit finalement $F : \mathcal{T} \to E$ définie par $F(t) = F_{h(t)}(t)$. Montrons que F est solution du problème.

Soit t une feuille. On a $F(t) = F_0(t) = f(t)$ par définition de F_0.

Soit maintenant $t = (t_1, x, t_2)$ un arbre qui n'est pas une feuille. Soit $h > 0$ la hauteur de t. On a $F(t) = F_h(t) = F_h((t_1, x, t_2)) = g(F_{h-1}(t_1), x, F_{h-1}(t_2))$. Or, t_1 et t_2 sont de hauteur inférieure ou égale à $h-1$. On a donc $F_{h-1}(t_1) = F_{h(t_1)}(t_1)$ $F_{h-1}(t_2) = F_{h(t_2)}(t_2)$, d'après la propriété de coïncidence des restrictions. Ainsi, $F(t) = g(F_{h(t_1)}(t_1), x, F_{h(t_2)}(t_2)) = g(F(t_1), x, F(t_2))$, ce qu'il fallait démontrer.

□

Remarque 3. On peut écrire en Caml une fonction ultra-générale qui renvoie des fonctions définies par induction structurelle. Ce que j'entends par là, c'est que la fonction F du théorème ci-dessus est calculable, et voici d'ailleurs un algorithme pour la calculer :

```
let rec ind_struct f g = function
  | Feuille a -> f a
  | Noeud (t1, x, t2) -> g(ind_struct f g t1, x, ind_struct f g t2)
```

On peut alors écrire de façon extrêmement compacte les fonctions calculant la hauteur, le nombre de noeuds et le nombre de feuilles d'un arbre.

```
let hauteur t = ind_struct (fun _ -> 0) (fun m _ n -> 1 + max m n) t
let nombre_noeuds t = ind_struct (fun _ -> 0) (fun m _ n -> 1 + m + n) t
let nombre_feuilles t = ind_struct (fun _ -> 1) (fun m _ n -> m + n) t
```

Franchement, c'est très très beau, mais c'est tellement compacté que c'est devenu incompréhensible. Je déconseille d'abuser des fonctions de fonctions de fonctions renvoyant des fonctions de fonctions...

II.9 Parcourir les arbres

Un grand nombre d'algorithmes nécessitent le parcours d'un arbre afin d'effectuer des opérations sur chacun des noeuds de l'arbre. On peut distinguer plusieurs types de parcours, parmi lesquels le parcours préfixe, le parcours infixe et le parcours postfixe. Voici, en pseudo-Caml, à quoi ressemble un parcours préfixe.

```
let rec parcours_prefixe t =
  match t with
  | Feuille f -> faire quelque chose avec f
  | Noeud (t1, x, t2) ->
      faire quelque chose avec x; (* 1 *)
      parcours_prefixe t1;        (* 2 *)
      parcours_prefixe t2         (* 3 *)
```

Pour le parcours infixe, on choisit l'odre 2-1-3 et pour le parcours suffixe on choisit l'ordre 2-3-1.

Exemple. Supposons que les feuilles de nos arbres soient des chaînes de caractères. Écrivons une fonction qui effectue un parcours de l'arbre et renvoie la concaténation de ses feuilles. Ici, infixe préfixe ou suffixe ne change rien, vu que l'on n'utilise pas les valeurs des noeuds.

```
let rec concat_feuilles t =
  match t with
  | Feuille f -> f
  | Noeud (t1, _, t2) -> concat_feuilles t1 ^ concat_feuilles t2
```

Exemple. Pour donner un autre exemple, voici une fonction qui prend un arbre en paramètre et renvoie la liste de ses noeuds. Choisissons un parcours infixe : lorsque nous parlerons d'arbres binaires de recherche, nous comprendrons pourquoi.

```
let rec liste_noeuds t =
  match t with
  | Feuille f -> []
  | Noeud (t1, x, t2) ->
      let s1 liste_noeuds t1
```

```
    and s2 = liste_noeuds t2 in
    s1 @ [x] @ s2
```

Exercice. Écrire une fonction `noeuds_feuilles : ('a, 'b) arbre -> 'a list * 'b list` prenant un arbre t en paramètre et renvoyant le couple (liste des noeuds de t, liste des feuilles de t). Choisir un parcours préfixe.

III Arbres binaires de recherche

III.1 Structure d'ABR

On suppose dans cette section que l'ensemble $\mathcal{N}$ des noeuds est totalement ordonné par une relation $\leq$. Pour simplifier, on suppose que l'ensemble des feuilles est un singleton $\mathcal{F} = \{f\}$, la valeur de l'objet f étant sans intérêt. L'unique arbre f réduit à une feuille est appelé l'arbre vide. Nous prendrons donc comme type

```
type 'a arbre =
| Feuille
| Noeud of 'a arbre * 'a * 'a arbre
```

Définition 5. Soit t un arbre. On dit que t est un arbre binaire de recherche (en abrégé ABR) lorsque

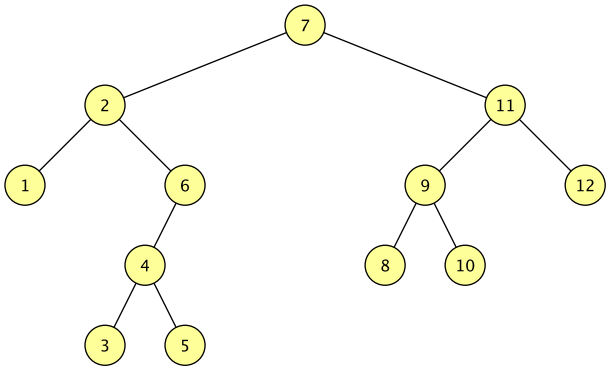
1. t est vide, ou bien
2. $t = (t_1, x, t_2)$, t_1 et t_2 sont eux-mêmes des ABR, les noeuds de t_1 sont inférieurs ou égaux à x, et les noeuds de t_2 sont strictement supérieurs à x.

Voici une fonction prenant un arbre en paramètre et vérifiant si cet arbre est un ABR.

```
let rec est_ABR t =
  match t with
  | Feuille           -> true
  | Noeud (t1, x, t2) -> est_abr t1 && est_abr t2 &&
                         maximum t1 <= x && minimum t2 > x
```

Voir un peu plus loin pour le calcul du minimum (algorithme analogue pour le maximum) d'un ABR.

Remarque 4. On remarque un dissymétrie entre la condition sur le fils gauche et celle sur le fils droit. Dans la pratique, nous supposerons les noeuds distincts, ce qui rend notre définition symétrique. Voici un exemple d'ABR dont les valeurs des noeuds sont des entiers (on ne représente pas les feuilles) :



III.2 Recherche, insertion

Voici le code de la fonction de recherche : `rechercher x t` renvoie `true` si x est un noeud de l'arbre t, et `false` sinon.

```
let rec rechercher x t =
  match t with
  | Feuille                    -> false
  | Noeud (_, y, _) when x = y  -> true
  | Noeud (t1, y, _) when x < y -> rechercher x t1
  | Noeud (_, _, t2)            -> rechercher x t2
```

La fonction d'insertion est tout à fait analogue : `inserer x t` renvoie un ABR résultant de l'insertion du noeud x dans l'arbre t.

```
let rec inserer x t =
  match t with
  | Feuille                     -> Noeud (Feuille, x, Feuille)
  | Noeud (_, y, _) when x = y   -> t
  | Noeud (t1, y, t2) when x < y -> Noeud (inserer x t1, y, t2)
  | Noeud (t1, y, t2)            -> Noeud (t1, y, inserer x t2)
```

Dans tout ce qui va suivre, nous nous intéresserons à la complexité des fonctions que nous écrirons. Nous appelerons complexité de la fonction `f : 'a arbre -> 'b` le nombre $C(f,t)$ de comparaisons de noeuds effectuées par la fonction lors de l'appel ft.

Proposition 8. *Les complexités en pire cas de la fonction de recherche et de la fonction d'insertion sont en $O(h(t))$.*

Démonstration. Nous le faisons pour la fonction de recherche. C'est une récurrence forte sur la hauteur des arbres. Si t est vide, alors $C(t) = 0$ (aucune comparaison). Sinon, $t = (t_1, x, t_2)$ ..... □

III.3 Minimum, suppression

Le minimum d'un ABR est le noeud « le plus à gauche » dans l'arbre.

```
let rec minimum t =
  match t with
  | Feuille               -> failwith "minimum"
  | Noeud (Feuille, x, _) -> x
  | Noeud (t1, _, _)      -> minimum t1
```

Encore une fois, nous avons une complexité en $O(h(t))$.

Voici la fonction de suppression.

```
let rec supprimer x t =
      match t with
      | Feuille -> Feuille
      | Noeud (t1, y, t2) when x < y -> Noeud (supprimer x t1, y, t2)
      | Noeud (t1, y, t2) when x > y -> Noeud (t1, y, supprimer x t2)
      | Noeud (t1, y, Feuille)      -> t1
      | Noeud (t1, y, t2)            ->
          let m = minimum t2 in
          Noeud (t1, m, supprimer m t2)
```

Seuls les deux derniers cas du filtrage méritent une explication. Dans ces deux cas, on désire supprimer la racine de l'arbre. Si cette racine n'a pas de fils droit (c'est à dire plutôt que son fils droit est l'arbre vide), on renvoie le fils gauche. Si, en revanche, cette racine a un fils droit, on récupère le minimum m du fils droit, on supprime m de ce fils, et on remplace la racine par m.

Proposition 9. *Soit t un ABR. Pour tout noeud x, l'arbre renvoyé par* `supprimer x t` *est un arbre binaire de recherche dont les noeuds sont exactement les noeuds de t, excepté x.*

Lemme 10. *Soit t un ABR. Soit x un noeud de t dont le fils droit n'est pas une feuille. Alors le succeseur de x dans l'ABR t est le minimum du fils droit de x.*

Démonstration. Par successeur de x on veut dire un noeud $m > x$ tel que les noeuds de t sont soit plus petits que x soit plus grands que m. Tout noeud de t différent du noeud maximal possède évidemment un successeur.

Soit donc x un noeud ayant un fils droit qui n'est pas une feuille, et soit m le minimum de ce fils droit. On suppose pour la démonstration que tous les noeuds de t sont distincts. Soit $y \neq x$ un noeud quelconque de t et soit z le plus proche ancêtre commun de x et y. Nous avons exactement 4 cas :

1. $z = x$. Si y est dans le fils gauche de x alors $y < x$. Si y est dans le fils droit de x alors $m \leq y$ puisque m est le minimum du fils droit de x.
2. $z = y$. Si x est dans le fils gauche de y, alors m aussi et donc $m < y$. Si x est dans le fils droit de y, alors $y < x$.
3. y est dans le fils gauche de z et x est dans le fils droit de z. Alors $y < z < x < m$.
4. y est dans le fils droit de z et x est dans le fils gauche de z. Alors m est aussi dans le fils gauche de z, et donc $m < z < y$.

□

Démonstration. On rappelle que l'on ne considère que des arbres dont les noeuds sont distincts. Notons, pour tout arbre t, par $\mathcal{N}(t)$, l'ensemble des noeuds de t. Induction structurelle, bien entendu. Si t est vide, c'est clair. Soient t_1 et t_2 deux ABR. Supposons que ces deux arbres vérifient la propriété. Soit y un noeud, soit $t = (t_1, y, t_2)$. Soit x un noeud. Trois cas sont à considérer.

- $x < y$. Par l'hypothèse d'induction, `t'=supprimer x t1` est un ABR et $\mathcal{N}(t') = \mathcal{N}(t_1) \setminus \{x\}$. On a ainsi $\mathcal{N}((t', y, t2)) = \mathcal{N}(t') \cup \{y\} \cup \mathcal{N}(t_2) = (\mathcal{N}(t_1) \setminus \{x\}) \cup \{y\} \cup \mathcal{N}(t_2) = (\mathcal{N}(t_1) \cup \{y\} \cup \mathcal{N}(t_2)) \setminus \{x\} = \mathcal{N}(t) \setminus \{x\}$. Il est de plus clair que l'arbre renvoyé est un ABR : supprimer un noeud dans le fils gauche préserve la structure.
- $x > y$. Cas totalement analogue, sur le fils droit cette-fois ci.
- $x = y$. Si t_2 est vide, c'est évident. Sinon, m est le successeur de x dans l'ABRet il est facile de voir que la structure d'ABR est préservée en remplaçant x par m.

□

III.4 Équilibrage

Tous les algorithms que nous venons d'étudier sont en $O(h(t))$ où $h(t)$ est la hauteur de l'ABR t. Il nous reste donc à savoir si cette complexité est intéressante. Montrons tout d'abord un petit résultat :

Proposition 11. *Soit t un arbre binaire. On a $h(t) \leq |t| \leq 2^{h(t)} - 1$.*

Démonstration. Par récurrence sur la hauteur h de l'arbre t. Pour $h = 0$, t est un feuille, $|t| = 0$ donc on a une double égalité. Soit $h > 0$, supposons la double inégalité vraie pour tous les arbres de hauteur strictement inférieure à h. Soit $t = (t_1, x, t_2)$ un arbre de hauteur h. Pour fixer les idées, on a $h(t_1) \leq h(t_2) = h - 1$.

Tout d'abord, $|t| = |t_1| + |t_2| + 1 \geq |t_2| + 1 \geq h - 1 + 1 = h$.

Ensuite, $|t| = |t_1| + |t_2| + 1 \leq (2^{h(t_1)} - 1) + (2^{h(t_2)} - 1) + 1 \leq 2^{h-1} + 2^{h-1} - 2 = 2^h - 1$. □

Remarque 5. Cette inégalité s'écrit encore $\lg(|t| + 1) \leq h(t) \leq |t|$ ou encore $\lg ||t|| \leq h(t) \leq |t|$ où, rappelons-le, $||t||$ est le nombre de feuilles de t.

Remarque 6. On peut facilement trouver, pour tout entier n, des arbres à n noeuds et de hauteur n. Par exemple, l'ABR obtenu par insertion successive dans un ABR vide des entiers de 1 à n dans l'ordre croissant. Ainsi, les algorithmes étudiés ci-dessus peuvent for bien avoir une complexité en $O(|t|)$, ce qui n'est pas bon. En effet, autant travailler sur des listes, dans ce cas !

Définition 6. Soit t un arbre binaire. On dira que t est équilibré lorsque t est une feuille, ou bien $t = (t_1, x, t_2)$, avec t_1 et t_2 équilibrés, et $-1 \leq h(t_1) - h(t_2) \leq 1$.

On autorise donc un certain déséquilibre, mais celui-ci ne doit pas être trop important. Montrons que les arbres équilibrés répondent à nos attentes.

Proposition 12. *Soit t un arbre équilibré. Alors $h(t) = O(\lg |t|)$.*

Démonstration. Montrons par récurrence sur h l'existence d'un réel $\alpha > 0$ tel que pour tout arbre équilibré t on ait $h(t) \leq \alpha \lg ||t||$. On rappelle que $||t||$ est le nombre de feuilles de t.

- C'est évident pour $h = 0$. Cela marche auusi pour $h = 1$ à condition que $\alpha \geq 1$.
- Supposons trouvé un tel réel α pour tous les arbres équilibrés de hauteur strictement inférieure à un entier $h > 1$. Soit $t = (t_1, x, t_2)$ un arbre équilibré de hauteur h. Par exemple, l'arbre t_1 est de hauteur $h - 1$. La propriété d'arbre équilibré impose que $h(t_2)$ est égale à $h - 2$ ou $h - 1$. On a alors

$$\begin{aligned} ||t|| = ||t_1|| + ||t_2|| &\geq 1 + 2^{(h-1)/\alpha} + 2^{(h-2)/\alpha} \\ &= 2^{h/\alpha}(2^{-1/\alpha} + 2^{-2/\alpha}) \end{aligned}$$

  C'est gagné, à la condition suffisante de trouver α tel que $2^{-1/\alpha} + 2^{-2/\alpha} = 1$. Un tel réel existe, et c'est $\alpha = \frac{1}{\lg \Phi}$ où Φ est le nombre d'or.

On a en conclusion $||t|| \geq 2^{h(t)/\alpha}$ pour tout arbre équilibré T différent de Nil, ou, si l'on préfère, $h(t) \leq \alpha \lg ||t||$, ce que l'on voulait. □

Remarque 7. On en déduit bien entendu que

$$h(t) \leq \alpha \lg(|t| + 1).$$

IV Tas et files de priorité

IV.1 Structure de tas

Définition 7. Soit t un arbre binaire dont les noeuds appartiennent à un ensemble totalement ordonné. On dit que t est un tas lorsque pour tout noeud x de t, les noeuds des fils de x sont supérieurs ou égaux à x.

Remarque 8. Ce que l'on vient de définir est précisément la structure de tas-min. On peut également considérer des tas-max en remplaçant « supérieurs » par « inférieurs ».

La structure de tas est en particulier intéressante pour construire des *files de priorité*. Une file de priorité est un type abstrait de données contenant des objets munis de clés. Ces clés sont supposées appartenir à un ensemble totalement ordonné. Les opérations supportées par une file de priorité sont :

1. L'extraction de la file d'un objet de clé minimale.
2. L'insertion d'un objet dans la file.
3. La diminution de la clé d'un objet dans la file.

Nous allons voir deux implémentations de ce type de données, l'une impérative, l'autre fonctionnelle (persistante).

IV.2 Tas impératifs

Première implémentation concrète du type abstrait de données « tas », le tas impératif. Voici la signature de ce type :

```
module type TAS_IMPERATIF =
  sig
    type 'a t
    exception TasVide
    exception TasPlein

    val nouveau       : 'a -> int -> 'a t

    val est_vide      : 'a t -> bool
    val taille        : 'a t -> int

    val cle_min       : 'a t -> 'a
    val prio_min      : 'a t -> int
    val inserer       : 'a * int -> 'a t -> unit
    val diminuer_cle  : 'a * int -> 'a t -> unit
    val supprimer_min : 'a t -> unit

    val inserer_liste : ('a * int) list -> 'a t -> unit
    val to_list       : 'a t -> ('a * int) list
  end
```

Un tas est ici un arbre binaire dont tous les niveaux sont complets, sauf le dernier niveau qui peut posséder des places libres sur la droite. On représente un tas par un tableau de couples formés d'un objet et de sa priorité. À l'indice 0 du tableau se trouve la racine du tas. À l'indice 1, le fils gauche de la racine et à l'indice 2 le fils droit. Aux indices 3 et 4 les fils gauche et droit du fils gauche de la racine, etc.

```
module TasImperatif : TAS_IMPERATIF =
  struct

    type 'a t = {
      donnees        : ('a * int) array;
      mutable taille : int;
      taille_max     : int
    }

    exception TasVide
    exception TasPlein

    let nouveau x n = {
      donnees    = Array.make n (x, 0);
      taille     = 0;
      taille_max = n
    }

    let est_vide h = h.taille = 0
```

```
let taille h = h.taille
let cle_min h = fst h.donnees.(0)
let prio_min h = snd h.donnees.(0)

let pere i = (i - 1) / 2
let fg i = 2 * i + 1
let fd i = 2 * i + 2
```

Deux fonctions clés pour la manipulation des données sont les fonctions `percoler` et `entasser`. La fonction `percoler` fair remonter un noeud mal placé, c'est à dire de priorité trop élevée, à la bonne position. La fonction `entasser` fait descendre un noeud de priorité trop faible à la bonne position. Ces deux fonctions ont une complexité en O de la hauteur du tas, c'est à dire $O(\log n)$ où n est le nombre d'éléments du tas.

Notre structure de tas est trop simpliste pour pouvoir implémenter efficacement la fonction de diminution d'une clé. La solution proposée ici provoque un doublon de la donnée dans le tas. Une autre solution consisterait à diminuer la clé de l'objet à la position i dans le tableau qui représente le tas, mais en pratique cela n'a pas grand intérêt : on connaît la valeur d'un objet mais pas sa position exacte dans le tas.

Pour bien faire, il faudrait maintenir une structure de données qui renverrait, pour chaque objet, sa position dans le tas : un arbre binaire de recherche, par exemple. Ou une table de hachage.

```
let echanger h i j =
  let tmp = h.donnees.(i) in
  h.donnees.(i) <- h.donnees.(j);
  h.donnees.(j) <- tmp

let rec percoler h i =
  if i > 0 && h.donnees.(pere i) > h.donnees.(i) then (
      echanger h i (pere i);
      percoler h (pere i)
    )

let inserer (x, p) h =
  if h.taille = h.taille_max then raise TasPlein;
  h.taille <- h.taille + 1;
  h.donnees.(h.taille - 1) <- (x, p);
  percoler h (h.taille - 1)

(* diminuer_cle provoque un doublon dans le tas *)
let diminuer_cle xp h = inserer xp h

let rec entasser h i =
  let m = ref i
  and g = fg i
  and d = fd i in
  if g < h.taille && h.donnees.(g) < h.donnees.(!m) then m := g;
  if d < h.taille && h.donnees.(d) < h.donnees.(!m) then m := d;
  if !m <> i then (
    echanger h i !m;
    entasser h !m
```

```
      )
```

Les trois dernières fonctions s'expliquent d'elles-mêmes.

```
    let supprimer_min h =
      if h.taille = 0 then raise TasVide;
      h.donnees.(0) <- h.donnees.(h.taille - 1);
      h.taille <- h.taille - 1;
      entasser h 0

    let rec inserer_liste xs h =
      match xs with
      | [] -> ()
      | xp :: ys ->
          inserer xp h;
          inserer_liste ys h

    let to_list h =
      let xs = ref [] in
      while h.taille > 0 do
        let xp = h.donnees.(0) in
        xs := xp :: !xs;
        supprimer_min h
      done;
      !xs

  end
```

IV.3 Tas fonctionnels

Signature

Nous nous intéressons ici à une structure de tas persistante. La signature de la struxture de données doit évidemment être légèrement modifiée pour tenir compte du caractère persistant des données. Plus d'effet de bord, les fonctions manipulant des tas prennent un tas en paramètre et renvoie un nouveau tas.

```
module type TAS_FONCTIONNEL =
  sig
    type 'a t
    exception TasVide

    val vide          : 'a t
    val est_vide      : 'a t -> bool
    val taille        : 'a t -> int

    val cle_min       : 'a t -> 'a
    val prio_min      : 'a t -> int
    val inserer       : 'a * int -> 'a t -> 'a t
    val supprimer_min : 'a t -> 'a t
```

```
    val diminuer_cle  : 'a * int -> 'a t -> 'a t

    val from_list     : ('a * int) list -> 'a t
    val to_list       : 'a t -> ('a * int) list
  end
```

Le type « tas fonctionnel »

Un tas fonctionnel est un arbre binaire dont chaque noeud contient trois informations : un objet (de type `'a`), une priorité qui est un nombre entier, et le rang du noeud, qui est lui-aussi un entier et qui est la distance du noeud au noeud externe le plus à droite sous le noeud en question. Ainsi, le rang du tas vide est 0, le rang d'un tas composé d'un seul noeud est 1, le rang d'un tas ayant deux noeuds est 0 ou 1, etc.

```
module Tas : TAS_FONCTIONNEL =
  struct

    type rg_type = int

    type 'a t =
    | Feuille
    | Noeud of 'a t * ('a * int) * 'a t * rg_type

    exception TasVide

    let vide = Feuille

    let est_vide h =
      match h with
      | Feuille -> true
      | _       -> false

    let rec taille h =
      match h with
      | Feuille            -> 0
      | Noeud (l, _, r, _) -> 1 + taille l + taille r
```

Comme dans la description des tas min, le noeud de priorité minimale se trouve à la racine du tas.

```
    let cle_min h =
      match h with
      | Feuille                -> raise TasVide
      | Noeud (_, (x, _), _, _) -> x

    let prio_min h =
      match h with
      | Feuille                -> raise TasVide
      | Noeud (_, (_, p), _, _) -> p
```

Le rang d'un tas est immédiat à calculer, puisque sa valeur se trouve stockée à la racine du tas.

```
let rang h =
  match h with
  | Feuille            -> 0
  | Noeud (_, _, _, r) -> r
```

Supposons donnés un couple (noeud, priorité) `xp`, et deux tas `a` et `b`. On suppose que la priorité de `xp` est inférieure aux priorités des noeuds des deux tas `a` et `b`. La fonction `faire_tas` recolle les morceaux en mettant comme fils droit de `xp` celui des tas `a` et `b` qui a le plus petit rang. Sa complexité est évidemment $O(1)$.

```
let faire_tas xp a b =
  if rang a >= rang b then Noeud (a, xp, b, rang b + 1)
                      else Noeud (b, xp, a, rang a + 1)
```

L'idée est de construire des tas ayant des rangs aussi petits que possible. Si l'on parvient ensuite à écrire des fonctions sur les tas de complexité en O du rang du tas (ce qui est loin d'être évident, convenons-en), on aura réussi à créer une structure performante.

Proposition 13. *La fonction* `faire_tas` *renvoie bien un tas.*

Démonstration. C'est évident. □

Étant donné un tas `a`, on note `rg a` son rang et `|a|` le nombre de noeuds internes de `a`.

Proposition 14. *Soient a et b deux tas. Soit t le tas renvoyé par* `faire_tas` *avec a et b en paramètres.*

- *On a $rg\ t = 1 + \min(rg\ a, rg\ b)$.*
- *Si m et n sont deux entiers naturels, on a*

$$\lg(\min(m,n)+1)+1 \leq \lg(m+n+1+1)$$

- *Si les tas a et b sont tels que $rg\ a \leq \lg(|a|+1)$ et $rg\ b \leq \lg(|b|+1)$, alors $rg\ t \leq \lg(|t|+1)$.*

Démonstration.

- C'est clair, puisque le tas de rang minimal est placé en fils droit du tas t.
- On a $\lg(\min(m,n)+1)+1 = \lg(\min(m,n)+1)+\lg 2 = \lg(2\min(m,n)+2) \leq \lg(m+n+2)$.
- On a $\text{rg}\ t = \min(\text{rg}\ a, \text{rg}\ b)+1 \leq \min(\lg(|a|+1), \lg(|b|+1))+1 = \lg(\min(|a|,|b|)+1)+1 \leq \lg(|a|+|b|+1+1) = \lg(|t|+1)$.

□

Proposition 15. *Pour tout rang t fabriqué à l'aide d'appels à* `faire_tas`*, on a*

$$rg\ t \leq \lg(|t|+1)$$

Démonstration. Immédiat par induction structurelle, en utilisant la proposition précédente. □

Fusion de deux tas

Voici maintenant la fonction la plus importante : elle fusionne deux tas. Si l'un des tas est vide, c'est évident, on renvoie l'autre tas. Sinon, soient `h1 = Heap((_, x), a1, b1)` et `h2 = Heap((_, y), a2, b2)` les tas en question. Si $x \leq y$, on fusionne `b1` et `h2`, ce qui crée un tas `t`. On renvoie alors le tas dont la racine est `x` et les fils sont `a1` et `t`. Sinon, on fait « le contraire ».

```
let rec fusionner h1 h2 =
  match (h1, h2) with
  | (_, Feuille) -> h1
  | (Feuille, _) -> h2
  | (Noeud (a1, (x1, p1), b1, _),
     Noeud (_ , (_ , p2),  _ , _)) when p1 <= p2 ->
      faire_tas (x1, p1) a1 (fusionner b1 h2)
  | (Noeud _, Noeud (a2, (x2, p2), b2, _)) ->
      faire_tas (x2, p2) a2 (fusionner h1 b2)
```

Proposition 16. *La fonction* `fusionner` *renvoie un tas.*

Démonstration. On procède par récurrence sur $n = \text{rg } h_1 + \text{rg } h_2$. Si h_1 ou h_2 est vide, et donc a fortiori si $n = 0$, le résultat est clair. Sinon, soit $n \geq 1$. supposons la propriété vraie pour tous les tas h_1, h_2 de rangs r_1, r_2 avec $r_1 + r_2 < n$. Soient h_1 et h_2 deux tas de rangs r_1 et r_2 tels que $r_1 + r_2 = n$. Si l'un des deux tas est vide, la conclusion est évidente. Sinon, il y a deux cas symétriques à traiter. Avec les notations de la fonction, plaçons nous par exemple dans le cas où $x \leq y$. Alors a_1 est un tas, et $h =$ fusionner $b_1 h_2$ aussi par l'hypothèse de récurrence. De plus x est inférieur aux priorités de a_1 et à celles de b_1. De plus, y est inférieur aux priorités de h_2, et comme $x \leq y$, il en va de même pour x. La fonction renvoie donc bien un tas. □

Proposition 17. *Étant donnés deux tas* `a` *et* `b` *on note* $C(a, b)$ *le nombre de comparaisons de priorités nécessaires à la fusion des tas* `a` *et* `b`. *On a*

$$C(a, b) \leq rg\ a + rg\ b$$

Démonstration. On procède par récurrence sur $n = \text{rg } h_1 + \text{rg } h_2$. Si h_1 ou h_2 est vide, alors $C(h_1, h_2) = 0$. Sinon, en reprenant les notations de la fonction, $C(h_1, h_2) \leq 1 + \max(C(b_1, h_2), C(h_1, b_2))$. D'après l'hypothèse d'induction, $C(b_1, h_2) \leq \text{rg } b_1 + \text{rg } h_2 = \text{rg } h_1 + \text{rg } h_2 - 1$. Il en est de même de $C(h_1, b_2)$. D'où $C(h_1, h_2) \leq \text{rg } h_1 + \text{rg } h_2$.

□

Insertion, suppression d'un objet

La fonction d'insertion est simple :

```
let inserer xp h = fusionner (faire_tas xp Feuille Feuille) h
```

Proposition 18. *L'insertion d'un objet dans un tas à n noeuds est de complexité $O(\log n)$.*

Démonstration. Appelons $C_{\text{ins}}(n, x, h)$ le nombre de comparaisons de priorités nécessaire à l'insertion de l'objet x dans le tas h à n noeuds . On a $C_{\text{ins}}(n, x, h) = C_{\text{fusion}}(h_x, h)$, où h_x est un tas à 1 noeud x. On a donc $C_{\text{ins}}(n, x, h) \leq 1 + \lg(n + 1)$. □

```
let supprimer_min h =
  match h with
  | Feuille            -> raise TasVide
  | Noeud (a, _, b, _) -> fusionner a b
```

Proposition 19. *La suppression de l'objet de priorité minimale dans un tas à n noeuds est de complexité $O(\log n)$.*

Démonstration. Appelons $C_{\text{del}}(n)$ le nombre de comparaisons nécessaire pour cette suppression. On a $C_{\text{del}}(n) = C_{\text{fusion}}(a, b)$ où a et b sont les fils du tas à n noeuds. Le nombre de noeuds de a et de b est inférieur ou égal à $n - 1$. Il s'ensuit que $C_{\text{del}}(n) \leq \lg(|a| + 1) + \lg(|b| + 1) \leq 2 \lg n$. □

Diminution d'une priorité

Il n'existe pas (à ma connaissance) d'algorithme efficace permettant de diminuer la priorité d'un objet dans un tas fonctionnel : on ne sait pas où se trouve l'objet dans le tas, et un parcours du tas est nécessaire pour retrouver l'objet. Une alternative consiste à réinsérer l'objet avec sa nouvelle priorité. L'inconvénient est de provoquer un doublon dans le tas, mais cette méthode peut fonctionner lorsqu'elle est utilisée pour certains algorithmes, comme l'algorithme de Dijkstra que nous verrons dans le chapitre sur les graphes.

```
(* diminuer_cle provoque un doublon dans le tas *)
let diminuer_cle xp h = inserer xp h
```

Conversion entre listes et tas

La fonction `from_list` crée un tas à partir d'une liste contenant des couples (objet, priorité). La fonction `to_list` prend un tas et renvoie la liste de couples (objet, priorité) obtenue par un parcours infixe de ce tas. On prouve facilement que la liste renvoyée est triée par priorités croissantes.

```
let rec from_list xs =
  match xs with
  | []       -> Feuille
  | xp :: ys -> inserer xp (from_list ys)

let rec to_list h =
  match h with
  | Feuille             -> []
  | Noeud (a, xp, b, _) -> to_list a @ [xp] @ to_list b

end
```

Proposition 20. *Les complexités des fonctions* `from_list` *et* `to_list` *sont en* $O(n \log n)$ *où* n *est la taille des données passées en paramètre (liste ou tas).*

Démonstration. La fonction `from_list` insère successivement des objets dans des tas de taille $0, 1, \ldots, n-1$. Sa complexité (en nombre de comparaisons de priorités) est donc $\sum_{k=0}^{n-1} C_{\text{ins}}(k) \leq \sum_{k=0}^{n-1} 1 + \lg(k+1) = n + \sum_{k=1}^{n} \lg k = n + \lg(n!) \sim n \lg n$.

La fonction `to_list` supprime quant à elle des objets dans des tas de taille n, $n-1$, $\ldots$, 1. Sa complexité (en nombre de comparaisons de priorités) est donc $\sum_{k=1}^{n} C_{\text{del}}(k) \leq \sum_{k=1}^{n} 2\lg(k) = 2\lg(n!) \sim 2n \lg n$.

□

IV.4 Annexe : le tri par tas

Prenons une liste s de couples (objet, priorité) t appliquons lui la fonction `from_list`. On obtient un tas. Appliquons à ce tas la fonction `to_list` : on obtient une liste ayant les mêmes éléments que la liste de départ, mais cette nouvelle liste est triée par priorités croissantes.

```
let heap_sort s = to_list (from_list s)
```

Proposition 21. *La complexité de cette fonction est en* $O(n \ln n)$*, où* n *est la taille de la liste à trier.*

Démonstration. Très précisément $C_{\text{tri}}(n) \leq n + 3\lg(n!) \sim 3n \lg n$. □

Chapitre 3

Langages

I De quoi allons-nous parler ?

Un langage est un ensemble de mots dont les lettres appartiennent à un alphabet. En informatique théorique, les langages interviennent dans un grand nombre de situations : langage des arbres, langage de la logique des propositions, de la logique du premier ordre, langages de programmation, etc.

Comment définit-on un langage ? Une méthode très générale pour définir une classe très vaste de langages est donner ce que l'on appelle une grammaire du langage. Nous ne suivrons pas cette voie, esentiellement parce que nous étudierons seulement une certaine classe de langages assez simples, appelés langages rationnels (ou encore langages réguliers). Nous donnerons deux façons équivalentes de définir ces langages : les expressions rationnelles (ou régulières) et les automates finis.

Étant donné un mot, appartient-il ou pas à un langage ? Pour certains langages, il est facile de répondre à cette question. Ce sera le cas pour les langages rationnels ; un automate fini sera une machine abstraite associée à un langage L. Lorsqu'on fournit un mot à une telle machine, celle ci répond « oui » si et seulement si $u \in L$.

Les langages de programmation sont évidemment une classe importante de langages. Il existe également des algorithmes prouvant qu'un programme donné est correct : ce sont les compilateurs.

II Alphabet, Mot, Langage

II.1 Les bases

Définition 1. On appelle *alphabet* tout ensemble (fini). Les éléments d'un alphabet A sont appelés des *lettres*.

Exemple. Voici quelques exemples d'alphabets.

- $A = \{0, 1\}$: l'alphabet binaire.
- $A = \{., -\}$: l'alphabet morse.
- $A = \{A, ..., Z, a, ..., z, 0, ..., 9, ...\}$: l'alphabet ASCII.
- $A = \{(,)\}$: l'alphabet de Dyck.

Définition 2. Pour $p \geq 1$, notons $[|0, p-1|] = \{0, ..., p-1\}$. On appelle *mot* sur l'alphabet A toute application $u : [|0, p-1|] \to A$, c'est à dire toute suite finie de lettres de l'alphabet A.

L'entier p est appelé la longueur du mot u, et est noté $p = |u|$.

On convient que l'on peut prendre également une suite vide de lettres, ce qui correspond à $p = 0$: on obtient alors le *mot vide*, que l'on notera ε et qui est de longueur 0.

Notation. L'ensemble des mots sur l'alphabet A est noté A^*.

Remarque 1. On confondra systématiquement un mot de longueur 1 avec son unique lettre. Avec ce léger abus, on a alors $A \subset A^*$.

Notation. On notera un mot par la simple juxtaposition de ses lettres. Par exemple, sur l'alphabet binaire, le mot $(0, 1, 0, 0, 1, 1)$ sera plus simplement noté 010011. Avec l'alphabet ASCII, nous écrirons « bonjour » et pas (b,o,n,j,o,u,r). En général, cela n'entraîne pas de confusion. Il faut cependant prendre garde dans certaines situations : sur l'alphabet $A = \{0, 10, 100\}$, le mot 0 100 10 a 3 lettres alors que le mot 0 10 0 10 0 en a cinq.

Définition 3. Soit A un alphabet. On appelle *langage* sur A toute partie L de A^*.

Exemple. Voici quelques exemples de langages.

- A^* est un langage sur A, et c'est le plus grand possible. L'ensemble vide est un langage, à ne pas confondre avec le langage $\{\varepsilon\}$ qui possède un mot, en l'occurrence le mot vide.
- Les mots sur l'alphabet $A = \{0, 1\}$ qui sont les représentations des multiples de 3 en base 2.
- Les mots sur l'alphabet $A = \{a, b\}$ contenant autant de a que de b.
- Les mots sur l'alphabet $A = \{a, b\}$ commençant par deux a ou finissant par trois b.
- Les palindromes sur l'alphabet $A = \{a, b\}$.

II.2 Le Monoïde libre

Sauf mention contraire, A désigne un alphabet.

Définition 4. Soient $u = u_0 \dots u_{m-1}$ et $v = v_0 \dots v_{n-1}$ deux mots non vides sur A de longueurs respectives m et n. On appelle *produit* (ou concaténation) des mots u et v le mot

$$u.v = u_0 \dots u_{m-1} v_0 \dots v_{n-1}$$

Si u est le mot vide, on pose $u.v = v$. De même, si $v = \varepsilon$, on pose $u.v = u$.

Proposition 1. *On a pour tous mots u, v de A^*, $|u.v| = |u| + |v|$.*

Démonstration. C'est clair. □

Proposition 2. *$(A^*, .)$ est un monoïde. Ce monoïde est non-commutatif dès que A possède au moins deux lettres distinctes.*

Démonstration. L'associativité est conséquence directe de la définition. L'élément neutre pour le produit est le mot vide ε. Enfin, si a et b sont deux lettres distinctes de A, alors $ab \neq ba$: le monoïde A^* n'est donc pas commutatif. □

Proposition 3. *Soit $u \in A^*$, $u \neq \varepsilon$. Alors, u s'écrit de façon unique comme produit d'éléments de A.*

Démonstration. Si u est un mot de longueur p, il s'écrit $u = u_0.u_1...u_{p-1}$, d'où l'existence. De plus, si deux mots sont égaux, ils ont même longueur et mêmes lettres, d'où l'unicité. □

La proposition ci-dessous peut être sautée en première lecture.

Proposition 4. *Soit $(\mathcal{M}, \star)$ un monoïde quelconque. Soit $f : A \to \mathcal{M}$. Soit $\iota : A \to A^*$ l'injection canonique de A dans A^* (c'est-à-dire l'application $x \mapsto x$). Il existe alors un unique morphisme de monoïdes $\overline{f} : A^* \to \mathcal{M}$ tel que $\overline{f} \circ \iota = f$. En d'autres termes toute fonction sur les lettres se prolonge en un unique morphisme sur les mots.*

Démonstration.

□

Exemple.

- Prenons $\mathcal{M} = \mathbb{N}$, muni de l'addition des entiers naturels. Posons, pour tout $x \in A$, $f(x) = 1$. On a alors $\overline{f}(u) = |u|$ pour tout mot $u \in A^*$.
- On peut généraliser un peu : soit a une lettre fixée de A. Prenons encore $\mathcal{M} = \mathbb{N}$. Posons, pour tout $x \in A$, $f(x) = 0$ si $x \neq a$, et $f(a) = 1$. On a alors $\overline{f}(u) = |u|_a$, le nombre d'occurences de la lettre a dans le mot u, pour tout $u \in A^*$.

II.3 Factorisations d'un mot

Définition 5. Soient $u, v \in A^*$. On dit que v est un facteur de u lorsqu'il existe deux mots u_1 et u_2 tels que $u = u_1 v u_2$. Lorsque $u_1 = \varepsilon$ convient, on dit que v est un suffixe de u. Lorsqu'on peut prendre $u_2 = \varepsilon$, v est un préfixe de u.

Exemple. Les lettres d'un mot sont toutes des facteurs de ce mot. On parlera également de facteur strict de u pour tout facteur de u différent de u lui-même (de même, on parlera de préfixe strict, de suffixe strict).

Théorème 5. *Soient u_1, v_1, u_2, v_2 des mots tels que $u_1 v_1 = u_2 v_2$.*

- *Si $|u_1| = |u_2|$, alors $u_1 = u_2$ et $v_1 = v_2$.*
- *Si $|u_1| < |u_2|$, alors u_1 est préfixe de u_2 et v_2 est suffixe de v_1.*
- *Si $|u_1| > |u_2|$, alors u_2 est préfixe de u_2 et v_1 est suffixe de v_2.*

Démonstration. On revient à la définition des mots comme suites finies de lettres. □

Proposition 6. *La relation « est préfixe de » est une relation d'ordre sur A^*. Il en est de même pour les suffixes.*

Notation. On notera $u \sqsubseteq v$ lorsque u est préfixe de v. Attention, la notation $v \sqsupseteq u$ est risquée : veut-on parler de suffixes, ou de la relation inverse de la relation préfixe ? Le mieux est de ne pas s'en servir.

Démonstration. On a $u = \varepsilon u$, donc $u \sqsubseteq u$. Si $u \sqsubseteq v$ et $v \sqsubseteq w$, alors $v = v_1 u$ et $w = w_1 v$, donc $w = w_1 v_1 u$. D'où $u \sqsubseteq w$.

Enfin, si $u \sqsubseteq v$ et $v \sqsubseteq u$, alors $v = v_1 u$ et $u = u_1 v$. D'où $u = u_1 v_1 u$. On en déduit que $u_1 v_1 = \varepsilon$. Donc $|u_1| + |v_1| = 0$, et ainsi $u_1 = v_1 = \varepsilon$. D'où $u = v$ et l'antisymétrie. □

III Opérations sur les langages

III.1 Union, intersection, différence

Ce sont les opérations ensemblistes classiques. Nous aurons par exemple l'occasion, étant donné un langage L, sur l'alphabet A, de nous intéresser à $\overline{L} = A^* \setminus L$. La réunion de deux langages L_1 est L_2 est souvent notée dans la littérature $L_1 + L_2$ au lieu de $L_1 \cup L_2$.

III.2 Produit

Définition 6. Soient L_1, l_2 deux langages sur A. On pose

$$L_1.L_2 = \{u_1 u_2, u_1 \in L_1, u_2 \in L_2\}$$

Exemple. $\{a^n, n \geq 0\}.\{b^n, n \geq 0\} = \{a^m b^n, m, n \geq 0\}$

Définition 7. Soit L un langage. On définit par récurrence sur n le langage L^n en posant $L^0 = \{\varepsilon\}$ et, pour tout $n \in \mathbb{N}$, $L^{n+1} = L^n.L$.

Remarque 2. Attention à ne pas confondre L^2 et $\{u^2, u \in L\}$. Le second langage est évidemment inclus dans L^2, mais n'est en général pas égal à celui-ci.

III.3 Étoile d'un langage

Théorème 7. *Soit L un langage sur A. Il existe un plus petit langage sur A, contenant L et le mot vide, et stable pour la concaténation. On le note L^*. On a de plus*

$$L^* = \sum_{n \geq 0} L^n$$

Démonstration. Notons $\mathcal{E}$ l'ensemble des langages sur l'alphabet A contenant L et le mot vide, et stable par concaténation. $\mathcal{E}$ est non vide, car $A^* \in \mathcal{E}$. Soit $\mathcal{M} = \cap_{\mathcal{A} \in \mathcal{E}} \mathcal{A}$. Il est clair que $\mathcal{M}$ convient.

Cette définition "par le haut" est d'un intérêt pratique limité. C'est ce qui explique la seconde partie du théorème. Notons $\tilde{L} = \cup_{n \geq 0} L^n$, et montrons que $\tilde{L} = L^*$.

- Il est clair que $\tilde{L}$ contient L et le mot vide. Soient u et v deux éléments de $\tilde{L}$. Il existe deux entiers p et q tels que $u \in L^p$ et $v \in L^q$. On ne peut pas a priori prendre $p = q$, sauf si $\varepsilon \in L$. De toute façon, le mot uv est dans L^{p+q}, donc dans $\tilde{L}$.
- Soit A un langage stable par concaténation, contenant L et le mot vide. On démontre aisément par récurrence sur k que $\forall k \in \mathbb{N}, L^k \subset A$. Ainsi, $\tilde{L} \subset A$.

Le langage $\tilde{L}$ est donc le plus petit langage contenant L et ε, et stable par concaténation. Donc, $\tilde{L} = L^*$. □

Exemple. Voici quelques exemples d'étoiles de langages.

- $L = \{a\}$. On a $L^* = \{a^n, n \geq 0\}$.
- $L = \{ab\}$. On a $L^* = \{(ab)^n, n \geq 0\}$.
- $L = A$. On a $L^* = A^*$ (ce qui est heureux).
- $L =$ L'ensemble des mots binaires représentant les multiples de 3. $L^* = L \cup \{\varepsilon\}$. Pour le prouver, énoncer un critère de divisibilité par 3 en base 2.

III.4 Etoile stricte

On utilise aussi, au lieu de l'étoile d'un langage L, le langage $L^+ = \sum_{n>0} L^n$. L^+ est le plus petit langage contenant L et stable par concaténation.

Exercice. Montrer que $L^+ = L^*$ si et seulement si le mot vide est lui-même un élément de L.

III.5 Propriétés utiles

Les propriétés ci-dessous sont laissées en exercice. Tous les objets mis en jeu sont des langages sur un alphabet A.

Proposition 8. *L'ensemble $\mathcal{P}(A^*)$, muni de l'opération réunion, est un monoïde commutatif :*

- $\emptyset + L = L + \emptyset = L$
- $(L_1 + L_2) + L_3 = L_1 + (L_2 + L_3)$
- $L_1 + L_2 = L_2 + L_1$

Proposition 9. *L'ensemble $\mathcal{P}(A^*)$, muni de l'opération produit, est un monoïde non commutatif (lorsque l'alphabet comporte au moins deux lettres) :*

- $\{\varepsilon\}L = L\{\varepsilon\} = L$
- $(L_1.L_2).L_3 = L_1.(L_2.L_3)$

Proposition 10.

- *L'ensemble vide est absorbant pour le produit des langages :* $\emptyset L = L\emptyset = \emptyset$.
- *Le produit est distributif par rapport à la réunion, même si cette réunion est infinie :* $L.\sum_{i\in I} L_i = \sum_{i\in I}(L.L_i)$ *et, de même,* $(\sum_{i\in I} L_i).L = \sum_{i\in I}(L_i.L)$

Remarque 3. Toutes ces propriétés font de $(\mathcal{P}(A^*), +, .)$ ce que l'on appelle un semi-anneau complet.

Exemple. Soit L un langage. Alors $LL^*+\varepsilon = L.\sum_{n\geq 0} L^n+\varepsilon = \sum_{n\geq 0} L.L^n+\varepsilon = \sum_{n\geq 1} L^n+L^0 = \sum_{n\geq 0} L^n = L^*$.

et, de même, $L^*L + \varepsilon = L^*$.

Chapitre 4

Logique des propositions

I A propos de logique

La logique mathématique est une branche des mathématiques au même titre que la théorie des groupes, le calcul intégral ou l'algèbre linéaire. Son apparente difficulté est qu'elle étudie des objets qui font eux-mêmes partie des mathématiques. Ainsi, le mathématicien qui fait de la logique travaille sur deux plans distincts : il étudie des objets de sa théorie, dans lesquels vont apparaître des symboles et des concepts comme ET, OU, et IMPLIQUE, et ces objets sont décrits par le langage de la logique. Mais ce même mathématicien, pour démontrer des théorèmes de logique, travaille à un niveau où existent également les mots ET, OU et IMPLIQUE, et ces démonstrations sont écrites dans le langage courant des mathématiques, que l'on appelle parfois le métalangage pour le distinguer du langage de la logique. Il conviendra dans ce qui suit de toujours bien faire la distinction entre langage et métalangage. L'auteur de ce polycopié s'est donc efforcé, autant que faire se peut, de ne pas utiliser dans le métalangage d'abréviations et de symboles déjà utilisés dans le langage de la logique. Il n'a évidemment pas réussi à effacer du métalangage les mots ET et OU.

II Syntaxe

II.1 Variables, connecteurs

On suppose fixé dans toute la suite un ensemble dénombrable $V = \{v_i, i \in \mathbb{N}\}$ où v_0, v_1, etc. sont des objets distincts. Les éléments de l'ensemble V sont appelés les *variables propositionnelles*. Le nom de variable est d'ailleurs un peu mal choisi : on pourrait parfaitement prendre pour V l'ensemble $\mathbb{N}$ des entiers naturels, et dans ce cas on voit mal ce que cela voudrait dire que de faire varier le nombre 174.

Dans la suite, lorsque nous prononcerons une phrase du genre « Soient x et y deux variables propositionnelles » nous voudrons dire « Soient $i, j \in \mathbb{N}$. Soient $x = v_i$ et $y = v_j$ ». En particulier, x et y pourront parfaitement désigner la même variable propositionnelle.

Remarque 1. L'ensemble V des variables propositionnelles est supposé dénombrable : il est donc infini. Ceci de façon à pouvoir disposer à chaque instant d'autant de variables distinctes que nécessaire. Il est dénombrable, parce que les objets que nous allons fabriquer à l'aide des variables propositionnelles, les *formules*, ne peuvent contenir qu'un nombre fini de variables, et que nous ne manipulerons qu'un nombre fini de formules à la fois dans ce qui nous intéresse. Personne n'affirme cependant ici qu'il serait inintéressant de considérer un ensemble V infini non dénombrable, et d'étudier des propriétés relatives à des ensembles infinis de formules. Bien au contraire.

On se donne également deux ensembles de symboles. Un ensemble $\mathcal{O}_1 = \{\neg\}$ dont l'unique élément est appelé connecteur unaire de négation ou plus vulgairement NON. Et un ensemble $\mathcal{O}_2 = \{\wedge, \vee, \longrightarrow, \longleftrightarrow\}$ dont les éléments sont appelés connecteurs binaires de conjonction, de disjonction, d'implication et d'équivalence. Ou plus simplement ET, OU, IMPLIQUE et ÉQUIVAUT.

Remarque 2. Le choix des connecteurs peut varier d'une présentation de la logique à une autre. Le choix que nous faisons ici d'avoir beaucoup de connecteurs binaires aura l'avantage de nous permettre d'écrire très rapidement des formules très variées, mais parfois l'inconvénient de compliquer un peu certaines définitions ou démonstrations.

II.2 Formules

Définition 1. Dans tout le chapitre, nous noterons $\mathcal{A} = V \cup \{), (, \wedge, \vee, \longrightarrow, \longleftrightarrow\}$. L'ensemble $\mathcal{A}$ est l'alphabet à partir duquel seront constituées les formules.

Définition 2. Le langage $\mathcal{L}$ des formules est le langage sur l'alphabet $\mathcal{A}$ défini de la façon suivante : on pose $\mathcal{L}_0 = V$, et pour tout entier $n \in \mathbb{N}$, $\mathcal{L}_{n+1} = \mathcal{L}_n \cup \{\neg\varphi : \varphi \in \mathcal{L}_n\} \cup \{(\varphi\alpha\psi) : \varphi, \psi \in \mathcal{L}_n, \alpha \in \mathcal{O}_2\}$. On pose enfin $\mathcal{L} = \cup_{n\in\mathbb{N}}\mathcal{L}_n$.

Proposition 1. *$\mathcal{L}$ est le plus petit langage sur l'alphabet $\mathcal{A}$ vérifiant :*

1. *Toute variable propositionnelle est une formule.*
2. *Si φ est une formule, alors $\neg\varphi$ est une formule.*
3. *Si φ et ψ sont deux formules alors $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \longrightarrow \psi)$ et $(\varphi \longleftrightarrow \psi)$ sont aussi des formules.*

Démonstration. Soit $\mathcal{M}$ un langage sur $\mathcal{A}$ vérifiant 1, 2 et 3. On a $\mathcal{L}_0 = V \subseteq M$ par le point 1. Soit $n \in \mathbb{N}$, supposons $\mathcal{L}_n \subseteq \mathcal{M}$. Par les propriétés de stabilité 2 et 3, on a immédiatement $\mathcal{L}_{n+1} \subseteq \mathcal{M}$. On a donc montré par récurrence sur n que $\forall n \in \mathbb{N}, \mathcal{L}_n \subseteq \mathcal{M}$. Ainsi, $\mathcal{L} \subseteq \mathcal{M}$.

Montrons maintenant que $\mathcal{L}$ vérifie 1, 2 et 3. La propriété 1 est clairement vérifiée. Montrons par exemple 3 (2 se traite de façon identique) : soient $\varphi, \psi \in \mathcal{L}$. Soit $\alpha \in \mathcal{O}_2$. Il existe deux entiers m et n tels que $\varphi \in \mathcal{L}_m$ et $\psi \in \mathcal{L}_n$. Prenons par exemple $m \leq n$. La suite de langages $(\mathcal{L}_n)_{n\in\mathbb{N}}$ est croissante au sens de l'inclusion, donc $\varphi, \psi \in \mathcal{L}_n$ et ainsi $(\varphi\alpha\psi) \in \mathcal{L}_{n+1} \subseteq \mathcal{L}$. □

Informellement, le fait que $\mathcal{L}$ vérifie les propriétés 1 2 et 3 signifie qu'avec des formules des variables et des connecteurs on ne peut fabriquer que des formules. La propriété de minimalité de $\mathcal{L}$ nous apprend quant à elle que toute formule est composée, selon des règles précises, à partir de connecteurs et d'autres formules. En fait on a très précisément :

Théorème 2. *[Lecture unique des formules] Soit φ une formule. On est dans un, et un seul, des trois cas suivants :*

1. *φ est une variable propositionnelle.*
2. *Il existe une unique formule ψ telle que $\varphi = \neg\psi$.*
3. *Il existe une unique formule ψ_1, une unique formule ψ_2 et un unique connecteur binaire α tels que $\varphi = (\psi_1\alpha\psi_2)$.*

Démonstration. Ce théorème est démontré en annexe. Il nous apprend qu'il y a une une et une seule façon de décomposer une formule « composée » en formules « plus élémentaires », où ce que l'on appelle élémentaire reste à préciser. En fait, à chaque formule on peut associer des nombres qui mesurent son degré de simplicité. Plus ces nombres sont petits, plus la formule est simple. Nous allons nous concentrer sur deux exemples de tels nombres : la longueur et la hauteur d'une formule. □

II.3 Démontrer des propriétés des formules

Définition 3. Soit $\varphi \in \mathcal{L}$ une formule. On appelle longueur de φ le nombre de lettres composant le mot φ. On notera $|\varphi|$ la longueur de la formule φ.

Exemple. Aucune formule n'est de longueur 0. Une formule est de longueur 1 si et seulement si c'est une variable propositionnelle. La formule $\varphi = ((x \wedge (x \longrightarrow y)) \longrightarrow y)$ est de longueur 13.

Exercice. Soient φ, ψ deux formules, α un connecteur binaire. Calculer $|\neg\varphi|$ et $|(\varphi\alpha\psi)|$ en fonction des longueurs de φ et ψ.

Définition 4. Soit $\varphi \in \mathcal{L}$ une formule. On appelle hauteur de φ le plus petit entier n tel que $\varphi \in \mathcal{L}_n$. On note $h(\varphi)$ la hauteur de φ.

Remarque 3. Soit φ une formule qui n'est pas une variable propositionnelle. Soit h la hauteur de φ. On a

- $h > 0$ (car φ n'est pas une variable)
- $\varphi \in \mathcal{L}_h$
- $\varphi \notin \mathcal{L}_{h-1}$ (par minimalité de h)
- Pour tout entier $k \geq h$, $\varphi \in \mathcal{L}_k$ (par croissance de la suite des $\mathcal{L}_k$)

L'entier h est même le seul entier à vérifier les points 2 et 3.

Exemple. Une formule est de hauteur 0 si et seulement si c'est une variable propositionnelle. La formule $\varphi = ((x \wedge (x \longrightarrow y)) \longrightarrow y)$ est de hauteur 3. Remarquer ici que sans le théorème de lecture unique, on ne pourrait avoir de façon évidente qu'une inégalité : $h(\varphi) \leq 3$.

Proposition 3.

- *Pour toute variable propositionnelle v, on a $h(v) = 0$.*
- *Pour toute formule φ, on a $h(\neg\varphi) = 1 + h(\varphi)$.*
- *Pour toutes formules φ et ψ et tout connecteur binaire α, on a $h((\varphi\alpha\psi)) = 1 + \max(h(\varphi), h(\psi))$.*

Démonstration. Prouvons le troisième point. Posons $\chi = (\varphi\alpha\psi)$ et $k = h(\chi) > 0$ (k n'est pas nul car χ n'est pas une variable). Supposons pour fixer les idées que $h(\varphi) \leq h(\psi) = h$. Nous devons montrer que $k = 1 + h$.

On a $\varphi, \psi \in \mathcal{L}_h$, donc $\chi \in \mathcal{L}_{h+1}$ et donc $h(\chi) = k \leq 1 + h$.

Inversement, $\chi \in \mathcal{L}_k \setminus \mathcal{L}_{k-1}$. Or $\mathcal{L}_k = \mathcal{L}_{k-1} \cup \mathcal{U} \cup \mathcal{V}$ où $\mathcal{U} = \{(\neg\varphi) : \varphi \in \mathcal{L}_{k-1}\}$ et $\mathcal{V} = \{(\varphi\alpha\psi) : \varphi, \psi \in \mathcal{L}_{k-1}, \alpha \in \mathcal{O}_2\}$. Donc $\chi \in \mathcal{U} \cup \mathcal{V}$. Le théorème de lecture unique nous permet d'affirmer que $\chi \in \mathcal{V}$ ET que φ et ψ sont dans $\mathcal{L}_{k-1}$. En particulier, $h(\psi) \leq k - 1$, ou encore $1 + h \leq k$. □

Exercice. Montrer par récurrence sur la hauteur des formules que pour toute formule φ,

$$1 + h(\varphi) \leq |\varphi| \leq 2^{h(\varphi)} - 3$$

Un grand nombre de propriétés sur les formules se montrent par des méthodes de type récurrence. Nous avons déjà à notre disposition les récurrences sur la longueur des formules, ainsi que les récurrences sur la hauteur des formules. On peut faire plus élégant en effectuant ce que l'on appelle des *inductions structurelles*.

Proposition 4. *Soit $\mathcal{P}[u]$ une propriété des mots u sur l'alphabet $\mathcal{A}$. On suppose que*

- *$\mathcal{P}[v]$ pour toute variable propositionnelle v.*
- *Pour toute formule $\varphi \in \mathcal{L}$, si $\mathcal{P}[\varphi]$ alors $\mathcal{P}[\neg\varphi]$.*
- *Pour toutes formules $\varphi, \psi \in \mathcal{L}$ et tout connecteur binaire α, si $\mathcal{P}[\varphi]$ et $\mathcal{P}[\psi]$ alors $\mathcal{P}[(\varphi\alpha\psi)]$.*

Alors pour toute formule $\varphi \in \mathcal{L}$, on a $P[\varphi]$.

Démonstration. Nous allons prendre grand soin dans la démonstration à ne pas utiliser le théorème de lecture unique. En effet, la démonstration du théorème de lecture unique fera sans cesse appel au théorème d'induction structurelle ! Démontrons le théorème d'induction structurelle par récurrence sur la hauteur des formules. Montrons plus précisément par récurrence sur n que pour tout entier $n \in \mathbb{N}$, pour toute formule φ de hauteur n, on a $\mathcal{P}[\varphi]$.

C'est clair pour $n = 0$, puisqu'une formule de hauteur 0 est une variable propositionnelle. Soit maintenant $n \in \mathbb{N}$. Supposons $\mathcal{P}[\psi]$ vraie pour toutes les formules ψ de hauteur n. Soit φ une formule de hauteur $n+1$. Cela signifie que $\varphi \in \mathcal{L}_{n+1} \setminus \mathcal{L}_n$. Par définition même de $\mathcal{L}_{n+1}$, on a donc $\varphi = \neg\psi$ ou $\varphi = (\psi_1\alpha\psi_2)$ où $\psi, \psi_1, \psi_2 \in \mathcal{L}_n$ et $\alpha \in \mathcal{O}_2$. Traitons par exemple le second cas : par l'hypothèse de récurrence, on a $\mathcal{P}[\psi_1]$ et $\mathcal{P}[\psi_2]$, et donc, par les hypothèses du théorème, on a $\mathcal{P}[\varphi]$. Le cas de la négation se traite de façon identique. □

II.4 Définir des fonctions sur l'ensemble des formules

L'induction structurelle, associée au théorème de lecture unique, permettent également de définir des fonctions sur l'ensemble $\mathcal{L}$ des formules. Précisément :

Proposition 5. *Soit E un ensemble. Soit $f_0 : V \to E$. Soit $f_1 : E \to E$. Soit $f_2 : E \times \mathcal{O}_2 \times E \to E$. Il existe une unique fonction $F : \mathcal{L} \to E$ telle que :*

- *Pour toute variable propositionnelle v, $F(v) = f_0(v)$.*
- *Pour toute formule φ, $F(\neg\varphi) = f_1(F(\varphi))$.*
- *Pour toutes formules φ et ψ et tout connecteur binaire α, $F((\varphi\alpha\psi)) = f_2(F(\varphi), \alpha, F(\psi))$.*

Démonstration. Nous admettons ce théorème, dont la démonstration, quoique sans difficultés particulières, est un peu longuette. □

Exemple. La hauteur d'une formule est la fonction $h : \mathcal{L} \to \mathbb{N}$ définie par induction structurelle à partir de $f_0(v) = 0$, $f_1(x) = x + 1$ et $f_2(x, \_, y) = 1 + \max(x, y)$.

La longueur d'une formule est la fonction $\ell : \mathcal{L} \to \mathbb{N}$ définie par induction structurelle à partir de $f_0(v) = 1$, $f_1(x) = x + 3$ et $f_2(x, \_, y) = 3 + x + y$.

Regardons maintenant un exemple plus sophistiqué.

II.5 Formules, arbres et Caml

Une formule peut être « vue » comme un arbre dont les feuilles sont des variables propositionnelles, et les noeuds, soit unaires soit binaires, sont des connecteurs. Le théorème de lecture unique nous dit qu'à toute formule peut être associé un arbre unique, son arbre syntaxique. On peut être plus précis que cela

- L'arbre syntaxique associé à une variable propositionnelle possède une unique feuille, étiquetée par cette variable.
- Pour toute formule φ, l'arbre syntaxique associé à $\neg\varphi$ a une racine étiquetée par $\neg$ et un unique fils qui est l'arbre syntaxique associé à φ.
- Pour toutes formules φ et ψ et tout connecteur binaire α, l'arbre syntaxique associé à $(\varphi\alpha\psi)$ a une racine étiquetée par α, un fils gauche qui est l'arbre syntaxique associé à φ et un fils droit qui est l'arbre syntaxique associé à ψ.

La structure d'arbre est facile à à programmer et à manipuler, on peut par exemple définir le type « formule » en Caml comme suit :

```
type varprop = string
type opbin = AND | OR | IMPL | EQV

type formule =
| Feuille of varprop
| Unaire  of formule
| Binaire of formule * opbin * formule
```

La fonction hauteur devient alors

```
let rec hauteur f =
  match f with
  | Feuille _          -> 0
  | Unaire g           -> 1 + hauteur g
  | Binaire (g, _, h) -> 1 + max (hauteur g) (hauteur h)
```

et la fonction longueur est

```
let rec longueur f =
  match f with
  | Feuille _          -> 1
  | Unaire g           -> 3 + longueur g
  | Binaire (g, _, h) -> 3 + longueur g + longueur h
```

Exercice. Écrire une fonction de type `formule -> int` prenant une formule en paramètre et renvoyant le nombre de parenthèses ouvrantes de cette formule.

Exercice. On définit par induction structurelle le concept de sous-formule : une variable propositionnelle a une unique sous-formule : elle-même. Les sous formules de $\neg\varphi$ sont elle-même et les sous-formules de φ. Les sous-formules de $(\varphi\alpha\psi)$ sont elle-même, les sous-formules de φ et les sous-formules de ψ.

Écrire une fonction `sous_formules : formule -> formule list` prenant une formule φ en paramètre et renvoyant la liste des sous-formules de φ.

Exercice. Écrire une fonction de type `formule -> (varprop * int) list` prenant en paramètre une formule et renvoyant la liste des variables propositionnelles ayant une occurrence dans la formule, ainsi que le nombre d'occurrences de ces variables.

Remarque 4. En fait, nous venons de définir par induction structurelle une fonction qui à toute formule associe un arbre, son arbre syntaxique. Pour les fanatiques de morphismes, nous venons de fabriquer une bijection $S : \mathcal{L} \to \mathcal{T}$ où $\mathcal{T}$ est l'ensemble des arbres syntaxiques. Et nous avons

- $\forall x \in V, S(x) =$ `Feuille` x
- $\forall \varphi \in \mathcal{L}, S(\neg\varphi) =$ `Unaire` $(S\varphi)$.
- $\forall \varphi \in \mathcal{L}, \forall \psi \in \mathcal{L}, \forall \alpha \in \mathcal{O}_2, S((\varphi\alpha\psi)) =$ `Binaire` $(S\varphi, \alpha, S\psi)$.

II.6 Conventions de suppression de parenthèses

Nous allons convenir de ne pas écrire certaines parenthèses dans les formules. Ces conventions sont les suivantes :

1. On n'écrit pas les parenthèses les plus externes. Ainsi, on écrira simplement $x \wedge x$ au lieu de $(x \wedge y)$.
2. On affecte un ordre de priorité aux opérateurs. Dans l'ordre du plus prioritaire au moins prioritaire : $\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\longleftrightarrow$. Ainsi, par exemple, $x \wedge y \longrightarrow z \vee t \wedge x$ signifie $((x \wedge y) \longrightarrow (z \vee (t \wedge x)))$
3. Les opérateurs associent à droite. Par exemple, $x \longrightarrow y \longrightarrow z$ signifie $(x \longrightarrow (y \longrightarrow z))$ et $a \wedge b \wedge c \wedge d \wedge e$ signifie $(a \wedge (b \wedge (c \wedge (d \wedge e))))$.

Il faudra bien prendre garde de rétablir les parenthèses lorsqu'on s'occupe de certaines propriétés des formules. Ainsi, la longueur de la formule $x \longrightarrow y \longrightarrow z$ est 9 et pas 5 !

II.7 Remplacement d'une ou plusieurs variables par des formules

Définition 5. Soit x une variable propositionnelle. Soit χ une formule. On définit par induction structurelle le remplacement de x par χ dans la formule φ, noté $[\chi/x]\varphi$:

- $[\chi/x]x = \chi$ et $[\chi/x]y = y$ pour toute variable y différente de x.
- $[\chi/x](\neg\psi) = (\neg[\chi/x]\psi)$.
- $[\chi/x](\psi_1\alpha\psi_2) = ([\chi/x]\psi_1\alpha[\chi/x]\psi_2)$.

On vérifie sans peine par induction structurelle que $[\chi/x]\varphi$ est encore une formule

Remarque 5. On peut également définir le remplacement simultané de n variables propositionnelles distinctes $x_1, \ldots, x_n$ par n formules $\chi_1, \ldots, \chi_n$ dans la formule φ, que nous noterons $[\chi_1/x_1, \ldots, \chi_n/x_n]\varphi$. Attention, un remplacement simultané n'est pas équivalent à n remplacements successifs.

III Sémantique

III.1 Distributions de valeurs de vérité

Définition 6. On appelle distribution de valeurs de vérité, ou encore valuation, ou encore environnement, toute fonction $\gamma : V \to \{0,1\}$.

Notation. Nous noterons Γ l'ensemble des environnements. On a en fait $\Gamma = \{0,1\}^V$.

Remarque 6. On peut mettre à la place de $\{0,1\}$ n'importe quel ensemble à deux éléments. Lorsque nous programmerons en Caml, il sera parfois intéressant de choisir l'ensemble $\{false, true\}$ des booléens.

III.2 Fonctions booléennes

On définit pour chaque connecteur α une fonction F_α.

- La fonction $F_\neg : \{0,1\} \to \{0,1\}$ est définie par $F_\neg(x) = 1 - x$.
- La fonction $F_\wedge : \{0,1\}^2 \to \{0,1\}$ est définie par $F_\wedge(x,y) = xy$.
- La fonction $F_\vee : \{0,1\}^2 \to \{0,1\}$ est définie par $F_\vee(x,y) = x + y - xy$.
- La fonction $F_\longrightarrow : \{0,1\}^2 \to \{0,1\}$ est définie par $F_\longrightarrow(x,y) = 1 - x + xy$.
- La fonction $F_\longleftrightarrow : \{0,1\}^2 \to \{0,1\}$ est définie par $F_\longleftrightarrow(x,y) = xy + (1-x)(1-y)$.

Le formules définissant ces fonctions ne sont pas canoniques. On peut reformuler les définitions de façon plus verbeuse :

- $F_\neg(0) = 1$ et $F_\neg(1) = 0$.
- $F_\wedge(x,y) = 1$ si et seulement si $x = y = 1$.
- $F_\vee(x,y) = 0$ si et seulement si $x = y = 0$.
- $F_\longrightarrow(x,y) = 0$ si et seulement si $x = 1$ et $y = 0$.
- $F_\longleftrightarrow(x,y) = 1$ si et seulement si $x = y$.

Les symboles de connecteurs qui, jusqu'à présent n'avaient pas de sens particuliers, acquièrent maintenant un sens : nous allons nous occuper de *sémantique*.

III.3 Évaluations d'une formule

Définition 7. Soit γ un environnement. On apppelle fonction d'évaluation la fonction ev : $\mathcal{L} \times \Gamma \to \{0,1\}$ définie par induction structurelle par :

- Pour toute variable propositionnelle v, $ev(v,\gamma) = \gamma(v)$.
- Pour toute formule φ, $ev(\neg\varphi,\gamma) = F_\neg(ev(\varphi,\gamma))$.
- Pour toutes formules φ et ψ et tout connecteur binaire α, $ev((\varphi\alpha\psi),\gamma) = F_\alpha(ev(\varphi,\gamma), ev(\psi,\gamma))$.

La quantité $ev(\varphi,\gamma)$ est appelée évaluation de la formule φ dans l'environnement γ. Nous noterons également $\overline{\gamma}(\varphi) = ev(\varphi,\gamma)$ pour toute formule φ et tout environnement γ.

Voici en Caml la fonction d'évaluation, de type `formule -> (varprop -> bool) -> bool` :

```
let fonc_unaire x = not x

let fonc_binaire alpha x y =
  match alpha with
  | AND  -> x && y
  | OR   -> x || y
  | IMPL -> (not x) || y
  | EQV  -> x = y

let rec eval f gamma =
  match f with
  | Feuille v             -> gamma v
  | Unaire g              -> fonc_unaire (eval g gamma)
  | Binaire (g, alpha, h) -> fonc_bool alpha (eval g gamma) (eval h gamma)
```

Si on préfère une fonction d'évaluation de type `formule -> (varprop -> int) -> int` à valeurs dans $\{0, 1\}$, ce sera :

```
let fonc_unaire x = 1 - x

let fonc_binaire alpha x y =
  match alpha with
  | AND  -> x * y
  | OR   -> x + y - x * y
  | IMPL -> 1 - x + x * y
  | EQV  -> x * y + (1 - x) * (1 - y)
```

et la fonction d'évaluation aura exactement le même code.

III.4 Tables de vérité

Proposition 6. *Soit γ un environnement. Soit φ une formule. La valeur de $\Gamma(\varphi)$ ne dépend que de la valeur de γ en les variables propositionnelles ayant une occurrence dans φ.*

Démonstration. Par induction struturelle sur les formules. □

Remarque 7. Ainsi, si les variables propositionnelles intervenant dans φ sont $v_1, v_2, \ldots, v_n$, il suffit de considérer les environnements restreints à $\{v_1, v_2, \ldots, v_n\}$ pour connaître toutes les évaluations possibles de φ. Il existe exactement 2^n restrictions d'environnements à cet ensemble de cardinal n. On peut donc résumer toutes les évaluations possibles dans un tableau de 2^n lignes, « la » table de vérité de la formule φ.

Cette table n'est pas unique, quelques conventions ne feront pas de mal.

- Dans les premières colonnes, les variables propositionnelles rangée dans un ordre logique, alphabétique par exemple.
- Dans les colonnes suivantes, les sous-formules de φ utiles à la lecture et à la compréhension de la table.
- Dans la dernière colonne, la formule φ.

- Sur chaque ligne, un environnement. Les lignes seront elles-aussi rangées dans un ordre logique, par exemple l'ordre d'une énumération en binaire dans l'ordre croissant pour les colonnes relatives aux variables.

III.5 Satisfiabilité, tautologies

Définition 8. Soit φ une formule.
- Soit γ un environnement. On dit que φ est satisfaite par γ (ou que γ satisfait φ) lorsque $\Gamma(\varphi) = 1$.
- On dit que φ est satisfiable lorsqu'il existe au moins un environnement γ qui la satisfait. On note alors $\gamma \models \varphi$.
- On dit que φ est une contradiction lorsqu'elle n'est pas satisfiable.
- On dit que φ est une tautologie lorsqu'elle est satisfaite par tous les environnements. On note alors $\models \varphi$.

Remarque 8. φ est une tautologie si et seulement si $\neg\varphi$ est une contradiction.

Remarque 9. φ est une contradiction si et seulement si $\neg\varphi$ est une tautologie. Savoir qu'une formule est une contradiction est donc aussi précieux qu'avoir une tautologie, puisque la négation de la dite formule sera précisément une tautologie.

Proposition 7. *Soit φ une tautologie. Soient $\chi_1, \ldots, \chi_n$ n formules et $x_1, \ldots, x_n$ n variables propositionnelles distinctes. Alors, la formule $[\chi_1/x_1, \ldots, \chi_n/x_n]\varphi$ est une tautologie.*

Démonstration. Nous faisons la démonstration dans le cas où l'on remplace une seule variable x par une formule χ. Le cas de n remplacements simultanés est identique. Considérons donc la formule $\psi = [\chi/x]\varphi$. Il s'agit de prouver que ψ est une tautologie. Soit γ un environnement. Soit γ' l'environnement défini par $\gamma'(y) = \gamma(y)$ si $y \neq x$, et $\gamma'(x) = ev(\chi, \gamma)$. γ' est un environnement. Démontrons que $ev(\psi, \gamma) = ev(\varphi, \gamma')$. La démonstration se fait par induction structurelle sur φ.

- Supposons que $\varphi = y$ est une variable propositionnelle. Si $y \neq x$, alors $\psi = \varphi = y$, et $ev(\varphi, \gamma') = \gamma'(y) = \gamma(y) = ev(\psi, \gamma)$. Si $y = x$, alors $\psi = \chi$ et $ev(\varphi, \gamma') = \gamma'(x) = ev(\chi, \gamma) = ev(\psi, \gamma)$.
- Supposons maintenant que $\varphi = \neg\varphi_1$ et que $ev(\varphi_1, \gamma') = ev([\chi/x]\varphi_1, \gamma)$. On a alors $ev(\psi, \gamma) = ev([\chi/x](\neg\varphi_1), \gamma) = ev(\neg([\chi/x]\varphi_1), \gamma) = 1 - ev([\chi/x]\varphi_1, \gamma)$. D'après l'hypothèse d'induction structurelle, ce nombre est égal à $1 - ev(\varphi_1, \gamma') = ev(\neg\varphi_1, \gamma') = ev(\varphi, \gamma')$.
- Le cas des connecteurs binaires se traite de façon identique.

Pour terminer la preuve de notre théorème, pour tout environnement γ, il existe donc un environnement γ' tel que $ev([\chi/x]\varphi, \gamma) = ev(\varphi, \gamma') = 1$ puisque φ est une tautologie. Ainsi, $[\chi/x]\varphi$ est aussi une tautologie puisque son évaluation dans tout environnement est égale à 1. □

Remarque 10. Nous avons là ce que l'on appelle un schéma de théorèmes, ou méta-théorème. Chaque tautologie nous donne automatiquement une infinité de tautologies. Ainsi, si l'on sait que pour une certaine variable propositionnelle x, $\models x \vee \neg x$, on aura automatiquement pour toute formule φ, $\models \varphi \vee \neg\varphi$.

III.6 Équivalence

Définition 9. Soient φ et ψ deux formules. On dit que φ et ψ sont équivalentes, et on note $\varphi \equiv \psi$, lorsque pour tout environnement γ on a $ev(\varphi, \gamma) = ev(\psi, \gamma)$.

Remarque 11. Deux formules sont équivalentes si et seulement si elles ont la même table de vérité. Par définition même de la table de vérité.

Remarque 12. $\varphi \equiv \psi$ n'est pas une formule : c'est un jugement que fait le mathématicien sur les formules φ et ψ et qui exprime que d'un point de vue sémantique ces deux formules sont indiscernables.

Proposition 8. *L'équivalence des formules est une relation d'équivalence.*

Démonstration. Conséquence directe de la définition. □

Proposition 9. *Soient φ et ψ deux formules. On a $\varphi \equiv \psi$ si et seulement si $\models \varphi \longleftrightarrow \psi$.*

Démonstration. Supposons $\varphi \equiv \psi$. Soit γ un environnement. On a $ev(\varphi \longleftrightarrow \psi, \gamma) = F_{\longleftrightarrow}(ev(\varphi, \gamma), ev(\psi, \gamma)) = F_{\longleftrightarrow}(ev(\varphi, \gamma), ev(\varphi, \gamma)) = F_{\longleftrightarrow}(x, x) = 1$. Donc $\models \varphi \longleftrightarrow \psi$. Inversement, supposons que $\models \varphi \longleftrightarrow \psi$. Soit γ un environnement. On a $F_{\longleftrightarrow}(ev(\varphi, \gamma), ev(\psi, \gamma)) = 1$ (tautologie), donc $ev(\varphi, \gamma) = ev(\psi, \gamma)$. Ceci est vrai pour tout environnement, donc $\varphi \equiv \psi$. □

III.7 Quelques tautologies

Nous donnons ci-dessous quelques tautologies et équivalences. D'après la proposition précédente, toute équivalence donne lieu à une tautologie et donc tout ce qui suit est tautologie. Les démonstrations peuvent soit se faire à l'aide de tables de vérité, soit en utilisant des équivalences déjà démontrées. Dans ce qui suit, les lettres grecques désignent des formules quelconques.

Conjonction, disjonction

- $\varphi \wedge \varphi \equiv \varphi$ (idempotence de la conjonction)
- $\varphi \wedge \psi \equiv \psi \wedge \varphi$ (commutativité de la conjonction)
- $\chi \wedge (\varphi \wedge \psi) \equiv (\chi \wedge \varphi) \wedge \psi$ (associativité de la conjonction)
- $\varphi \vee \varphi \equiv \varphi$ (idempotence de la disjonction)
- $\varphi \vee \psi \equiv \psi \vee \varphi$ (commutativité de la disjonction)
- $\chi \vee (\varphi \vee \psi) \equiv (\chi \vee \varphi) \vee \psi$ (assoiciativité de la disjonction)
- $\chi \vee (\varphi \wedge \psi) \equiv (\chi \vee \varphi) \wedge (\chi \vee \psi)$ (distributivité de la disjonction par rapport à la conjonction)
- $\chi \wedge (\varphi \vee \psi) \equiv (\chi \wedge \varphi) \vee (\chi \wedge \psi)$ (distributivité de la conjonction par rapport à la disjonction)

Négation

- $\neg\neg\varphi \equiv \varphi$ (double négation)

- $\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$ (loi de Morgan)
- $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$ (loi de Morgan)

Implication

- $\models \varphi \longrightarrow (\varphi \vee \psi)$ (introduction du ou)
- $\models \psi \longrightarrow (\varphi \vee \psi)$
- $\models (\varphi \wedge \psi) \longrightarrow \varphi$ (élimination du et)
- $\models (\varphi \wedge \psi) \longrightarrow \psi$
- $(\varphi \longrightarrow \psi) \equiv \neg\varphi \vee \psi$
- $(\varphi \vee \psi) \equiv \neg\varphi \longrightarrow \psi$
- $\models \varphi \longrightarrow (\varphi \longrightarrow \psi) \longrightarrow \psi$ (modus ponens)
- $(\phi \wedge \psi) \longrightarrow \chi \equiv \varphi \longrightarrow \psi \longrightarrow \chi$
- $\varphi \longrightarrow \psi \equiv \neg\psi \longrightarrow \neg\varphi$ (contraposition)
- $\models (\varphi \longrightarrow \psi) \longrightarrow (\psi \longrightarrow \chi) \longrightarrow (\varphi \longrightarrow \chi)$ (transitivité de l'implication)

Équivalence

- $\varphi \longleftrightarrow \psi \equiv (\varphi \longrightarrow \psi) \wedge (\psi \longrightarrow \varphi)$ (une équivalence est une double implication)

Divers

- $\models \varphi \vee \neg\varphi$ (tiers exclu)
- $\models \neg(\varphi \wedge \neg\varphi)$ (non contradiction)

IV Annexe : le théorème de lecture unique

Théorème 10. *[Lecture unique des formules] Soit φ une formule. On est dans un, et un seul, des trois cas suivants :*

1. *φ est une variable propositionnelle.*
2. *Il existe une unique formule ψ telle que $\varphi = (\neg\psi)$.*
3. *Il existe une unique formule ψ_1, une unique formule ψ_2 et un unique connecteur binaire α tels que $\varphi = (\psi_1 \alpha \psi_2)$.*

On démontre facilement par induction structurelle que la première lettre d'une formule est une variable propositionnelle ou une parenthèse ouvrante ou le connecteur $\neg$, ces trois cas s'excluant mutuellement. On est donc dans au plus l'un des trois cas 1, 2, 3.

L'existence des écritures est également immédiate. Si φ n'est pas une variable propositionnelle, alors $\varphi \in \mathcal{L}_h \setminus \mathcal{L}_{h-1}$ où $h > 0$ est la hauteur de la formule φ. On conclut grâce à la définition de $\mathcal{L}_h$.

Il reste à prouver l'unicité, ce qui va nécessiter un certain nombre de résultats intermédiaires.

Définition 10. Pour tout mot u sur l'alphabet $\mathcal{A}$ défini au début du chapitre, on note $o[u]$ et $f[u]$ le nombre de parenthèses ouvrantes et le nombre de parenthèses fermantes apparaissant dans u.

Lemme 11. *Pour toute formule φ, on a $o[\varphi] = f[\varphi]$.*

Démonstration. Par une induction structurelle très facile. □

Lemme 12. *Soient φ une formule et u un préfixe de φ. Alors $o[u] \geq f[u]$.*

Démonstration. C'est encore une induction structurelle. □

Lemme 13. *Soit φ une formule dont le premier caractère est une parenthèse ouvrante. Soit u un préfixe propre de φ (c'est-à-dire un préfixe de φ distinct de φ et du mot vide). Alors $o[u] > f[u]$.*

Démonstration. La formule φ est donc de la forme $(\psi_1 \alpha \psi_2)$ où ψ_1 et ψ_2 sont deux formules et α est un connecteur binaire. Deux cas se présentent :

- Si $u = (v$ où v est un préfixe de ψ_1, alors $o[u] = 1 + o[v] \geq 1 + f[v]$ d'après le lemme précédent. Et $f[u] = f[v]$. Donc $o[u] > f[u]$.
- Si $u = (\psi_1 \alpha w$ où w est un préfixe de ψ_2, alors $o[u] = 1 + o[\psi_1] + o[w] = 1 + f[\psi_1] + o[w] \geq 1 + f[\psi_1] + f[w]$, et $f[u] = f[\psi_1] + f[w]$. D'où la conclusion.

□

Lemme 14. *Soit φ une formule. Soit u un préfixe strict de φ (c'est à dire un préfixe de φ différent de φ). Alors, u n'est pas une formule.*

Démonstration. On raisonne par induction structurelle.

- Une variable propositionnelle n'a que le mot vide comme préfixe strict. Et le mot vide n'est pas une formule.
- Soit ψ une formule dont aucun préfixe strict n'est une formule. Soit $\varphi = \neg\psi$. Soit u un préfixe strict de F. Alors, ou bien $u = \varepsilon$, le mot vide, et alors ce n'est pas une formule, ou bien $u = \neg v$ où v est un préfixe strict de ψ. Si jamais $\neg v$ était une formule, alors on aurait $\neg v = \neg\chi$ où χ est une formule. Mais alors $v = \chi$ serait une formule. Ce n'est pas possible car v est un préfixe propre de ψ.
- Soient ψ_1 et ψ_2 deux formules dont aucun préfixe propre n'est une formule. Soit α un connecteur binaire. Soit $\varphi = (\psi_1 \alpha \psi_2)$. Soit u un préfixe strict de φ. Si c'est le mot vide, ce n'est pas une formule. Sinon, c'est un préfixe propre et $o[u] > f[u]$ d'après le lemme précédent. Donc, u n'est pas une formule d'après le lemme 1.

□

Il reste à terminer la preuve du théorème de lecture unique. L'unicité dans les cas 1 et 2 étant évidente, nous devons seulement vérifier l'unicité de l'écriture des formules commençant par une parenthèse. Soit φ une formule. On suppose que $\varphi = (\varphi_1 \alpha \varphi_2) = (\psi_1 \beta \psi_2)$ où $\varphi_1, \varphi_2, \psi_1, \psi_2$ sont des formules et α, β des connecteurs binaires. On raye les premières lettres des mots, et on en déduit $\varphi_1 \alpha \varphi_2) = \psi_1 \beta \psi_2)$. Attention, les mots qui apparaissent dans cette égalité ne sont plus des formules. Supposons par exemple que le mot φ_1 soit plus court que le mot ψ_1. Alors, φ_1 est un préfixe de ψ_1 et ce ne peut être un préfixe propre vu que φ_1 et ψ_1 sont des formules (lemme précédent, un préfixe propre d'une formule n'est jamais une formule). Donc, $\varphi_1 = \psi_1$. On peut alors encore simplifier l'égalité, pour en tirer $\alpha \varphi_2) = \beta \psi_2)$. On en déduit $\alpha = \beta$, on re-simplifie, et on obtient enfin $\varphi_2 = \psi_2$ et l'unicité tant espérée.

Chapitre 5

Graphes

I Le vocabulaire des graphes

I.1 Notion de graphe

Définition 1. Un graphe (simple, non orienté) est un couple $G = (S, A)$ où S est un ensemble fini, l'ensemble des sommets de G, et A est un ensemble de paires d'éléments distincts de S, les arêtes de G.

Notation. On notera $\mathcal{P}_2(S)$ l'ensemble des parties à deux éléments de l'ensemble S. Ainsi, l'ensemble A des arêtes du graphe $G = (S, A)$ vérifie $A \subset \mathcal{P}_2(S)$.

Définition 2. Soit $s = \{x, y\}$ une arête du graphe G. Les sommets x et y sont appelés les extrémités de l'arête s.

Deux sommets qui sont les extrémités d'une même arête sont dits **voisins** ou **adjacents**.

Le nombre de voisins d'un sommet x du graphe G est appelé le **degré** de x, et noté $d_G(x)$ ou $d(x)$ s'il n'y a pas de confusion. Un sommet de degré 0 est dit isolé. Un sommet de degré 1 est dit pendant.

Remarque 1. Commme leur nom l'indique, les graphes sont souvent représentés graphiquement. Chaque sommet d'un graphe est représenté par un point ou un cercle. Une arête d'extrémités les sommets a et b est représentée par une ligne (courbe, droite) joignant les points correspondants.

Proposition 1. *Soit $G = (S, A)$ un graphe possédant m arêtes. On a*

$$\sum_{x \in S} d_G(x) = 2m$$

Démonstration. Par récurrence sur m. C'est évident si $m = 0$. Tous les sommets du graphe sont alors de degré 0. Supposons maintenant la propriété vraie pour les graphes possédant m arêtes. Soit G un graphe ayant $m+1$ arêtes. Soit $s = \{a, b\}$ une arête de G. Soit $G' = (S, A \setminus \{s\})$ le graphe obtenu à partir de G en enlevant l'arête s. G' a donc m arêtes, donc $\sum_{x \in V} d_{G'}(x) = 2m$. Pour tout $x \in V$ différent de a et b on a $d_{G'}(x) = d_G(x)$. Et de plus $d_{G'}(a) = d_G(a) - 1$ et $d_{G'}(b) = d_G(b) - 1$. Donc $\sum_{x \in V} d_{G'}(x) = 2m = \sum_{x \in V} d_G(x) - 2$. D'où $\sum_{x \in V} d_G(x) = 2(m + 1)$. □

I.2 Quelques exemples de graphes

Définition 3. Pour tout entier $n \geq 0$, « le » graphe complet à n sommets, noté K_n, est le graphe possédant n sommets et dont tous les sommets sont reliés deux à deux par une arête.

Proposition 2. *Le nombre d'arêtes de K_n est $\frac{n(n-1)}{2}$.*

Démonstration. C'est en effet $\binom{n}{2}$, le nombre de parties à 2 éléments d'un ensemble de cardinal n. □

Exercice. Dessiner K_n pour $n = 1, 2, 3, 4$.

Définition 4. Soit $n \geq 0$. « Le » chemin P_n est le graphe dont les sommets sont les entiers $0, 1, \ldots, n$ et dont les arêtes relient les sommets i et $i+1$ pour $i = 0, 1, \ldots, n-1$. L'entier n est appelé la longueur de P_n.

Définition 5. Soit $n \geq 3$. « Le » cycle C_n est le graphe dont les sommets sont les entiers $0, 1, \ldots, n-1$ et dont les arêtes relient les sommets i et $i+1$ pour $i = 0, 1, \ldots, n-2$, ainsi que les sommets $n-1$ et 0. L'entier n est appelé la longueur de C_n.

Exercice. Dessiner P_n pour $n = 0, 1, 2, 3, 4$ et C_n pour $n = 3, 4, 5, 6$.

Remarque 2. Remarquer le « le » entre guillemets : renommer les sommets d'un graphe ne change rien à sa structure. D'autres graphes que ceux ci-dessus méritent d'être appelés cycle ou chemin parce que ce sont des graphes isomorphes à C_n ou P_n. Nous n'insisterons pas ici sur la notion (importante !) d'isomorphisme de graphes.

I.3 Chemins et cycles dans un graphe

Nous donnons ici une nouvelle définition des mots *chemin* et *cycle*. Elles seront équivalentes aux précédentes lorsque nous aborderons la notion de sous-graphe.

Définition 6. Soit $G = (S, A)$ un graphe. Un **chemin** de longueur $n \geq 0$ du sommet x au sommet y est une suite $P = (x_0 = x, x_1, \ldots, x_n = y)$ de sommets de G telle que pour tout $k \in [0, n-1]$, $\{x_k, x_{k+1}\} \in A$. On dit que les sommets x et y sont reliés par le chemin P. On note en cas de besoin par $|P|$ la longueur du chemin P.

Remarque 3. Selon les cas on pourra imposer à un chemin d'avoir tous se sommets distincts (chemin élémentaire). Le contexte permet de trancher. Par exemple, une propriété du type « il existe une unique chemin du sommet x au sommet y » impose clairement aux sommets du chemin d'être distincts deux à deux.

Définition 7. Soient x et y deux sommets du graphe G. La distance de x à y, notée $d_G(x, y)$, est la longueur du plus court chemin, s'il en existe, reliant x et y. S'il n'en existe pas, on pose $d(x, y) = +\infty$.

Définition 8. Un graphe dans lequel tous les sommets sont reliés par au moins un chemin est dit **connexe**.

Proposition 3. *Soit $G = (S, A)$ un graphe connexe. La fonction distance d_G est effectivement une distance sur l'ensemble S.*

Démonstration. La fonction $d : S \times S \to \mathbb{R}$ est évidemment positive. Les chemins de longueur 0 sont de la forme (x) où $x \in S$. On en déduit donc que $d(x, x) = 0$ et $d(x, y) = 0 \Rightarrow y = x$.

Soient x et y deux sommets de G. Pour tout chemin P reliant x à y, il existe une chemin $\tilde{P}$ reliant y à x obtenu en renversant la liste P. Ainsi, $d(x, y) = d(y, x)$.

Soient enfin x, y et z trois sommets de G. Soient P un chemin de longueur minimale reliant x à y et Q un chemin de longueur minimale reliant y à z. Le chemin R obtenu en juxtaposant les chemins P et Q est un chemin de x à z. On a de plus $|R| = |P| + |Q| = d(x, y) + d(y, z)$. Par minimalité de $d(x, z)$, on en déduit $d(x, z) \leq d(x, y) + d(y, z)$. □

Définition 9. Soit $n \geq 3$. Un **cycle** de longueur n dans le graphe G est une liste $C = (x_0 = x, x_1, \ldots, x_n = x)$ de sommets tous distincts de G, sauf $x_0 = x_n$ et tels que deux sommets successifs de C soient reliés par une arête de G.

I.4 Sous-graphes. Composantes connexes

Définition 10. Soit $G = (S, A)$ un graphe.

Un **sous-graphe** de G est un graphe $G' = (S', A')$ tel que $S' \subseteq S$ et $A' \subseteq A \cap \mathcal{P}_2(S')$. On notera $G' \subseteq G$.

Soit $S' \subseteq S$. Le sous graphe de G **induit** par S' est le graphe $G' = (S', A \cap \mathcal{P}_2(S')$. C'est à dire que l'on met dans A' **toutes** les arêtes de G reliant deux sommets de S'.

Proposition 4. *Soit $G = (S, A)$ un graphe. La relation $\mathcal{R}$ sur S définie, pour tous x, $y \in S$ par $x\mathcal{R}y$ si et seulement si il existe un chemin reliant x à y est une relation d'équivalence sur S.*

Définition 11. Cette relation d'équivalence induit une partition de l'ensemble des sommets de G. Les sous-graphes induits par les ensembles de cette partition s'appellent les composantes connexes de G.

Définition 12. Étant donnée une propriété $\mathcal{P}$ des graphes, G' est **maximal** pour la propriété $\mathcal{P}$ parmi les sous-graphes de G lorsque $\mathcal{P}(G')$ et, pour tout sous-graphe G'' de G, $\mathcal{P}(G'') \wedge G' \subseteq G'' \Rightarrow G' = G''$.

Proposition 5. *Les composantes connexes de G sont les sous-graphes de G connexes maximaux.*

Démonstration. Laissé en exercice. □

I.5 Graphes pondérés

Définition 13. Un graphe **pondéré** est un graphe $G = (S, A, f)$ où S et A sont définis comme précédemment et $f : A \to \mathbb{R}$.

Remarque 4. À toute arête s du graphe pondéré $G = (S, A, f)$ on associe son **poids** $f(s)$. Ce poids peut représenter un coût (gain ou perte selon son signe), une distance (s'il est positif), etc.

I.6 Graphes orientés

Définition 14. Un graphe orienté est un couple $G = (S, A)$ où S est un ensemble fini (les sommets de G) et A est un ensemble de couples d'éléments distincts de S (les arêtes de G).

Remarque 5. Un graphe non orienté peut aussi être vu comme un graphe orienté $G = (S, A)$ tel que pour tout $(x, y) \in A$, on ait $(y, x) \in A$. Ainsi, les structures de données que nous utiliserons pour représenter les graphes orientés conviennent tout aussi bien pour les graphes non orientés, à condition, dans les algorithmes que nous écrirons, de préserver la symétrie dans les arêtes.

Remarque 6. Les définitions vues pour les graphes non orientés s'adaptent facilement. Il faut juste faire attention à l'orientation des arêtes. Voici quelques exemples d'adaptations :

Définition 15. Soit $G = (S, A)$ un graphe non orienté. Soient x et y deux sommets de G. On dit que y est voisin de x, ou adjacent à x lorsque $(x, y) \in A$. Le *degré sortant* $d^+(x)$ d'un sommet x est le nombre de voisins de x. Le *degré entrant* $d^-(x)$ du sommet x est le nombre de sommets dont x est voisin.

Proposition 6. *Soit $G = (S, A)$ un graphe orienté possédant m arêtes. On a*

$$\sum_{x \in S} d^+(x) = \sum_{x \in S} d^-(x) = m$$

Démonstration. Récurrence sur m, comme pour les graphes non orientés. □

Proposition 7. *Soit G un graphe orienté. La relation entre sommets de G définie par « il existe un chemin de x à y et un chemin de y à x » est une relation d'équivalence sur l'ensemble des sommets de G.*

Remarque 7. Pour un graphe orienté, on ne parle plus de composantes connexes mais de **composantes fortement connexes**.

II Représentation des graphes

II.1 Représentation par listes d'adjacence

Soit $G = (S, A)$ un graphe ayant n sommets. Quitte à renommer ces sommets, on peut supposer que $S = [0, n-1]$. On peut représenter le graphe G par un tableau de taille n, le $i$ème élément du tableau étant la liste des voisins du sommet i.

```
type graphe = int list array
```

Remarque 8. Inconvénient de cette approche : on ne peut pas rajouter ou supprimer facilement de sommet au graphe. Il peut également être coûteux de savoir si deux sommets sont voisins.

Voici en exemple quelques fonctions évidentes opérant sur un graphe orienté :

```
let voisins g i = g.(i)
let ajouter_arete g i j = g.(i) <- inserer j g.(i)
let supprimer_arete g i j = g.(i) <- supprimer j g.(i)
let sont_voisins i j g = List.mem j g.(i)
```

où les fonctions `inserer` et `supprimer` insèrent et suppriment respectivement un objet dans une liste. La fonction `supprimer` est de complexité $O(d)$ en pire cas, où d est le degré du sommet d'origine de l'arête. Pour la fonction `inserer`, tout dépend. Si on insère des arêtes dont on sait qu'elles n'existent pas encore, alors on peut faire des ajouts en tête de liste. Complexité $O(1)$. Sinon, complexité $O(d)$.

Dans le cas où le graphe est non orienté, ces fonctions deviennent :

```
let voisins g i = g.(i)
let ajouter_arete g i j =
  g.(i) <- inserer j g.(i);
  g.(j) <- inserer i g.(j)
let supprimer_arete g i j =
  g.(i) <- supprimer j g.(i);
  g.(j) <- supprimer i g.(j)
let sont_voisins i j g = List.mem j g.(i)
```

Écrivons enfin une fonction transformant un graphe orienté en le graphe non orienté correspondant :

```
let desorienter g =
  for i = 0 to Array.length g - 1 do
```

```
    symetriser g g.(i) i
  done
```

où la fonction `symetriser` est définie par

```
let symetriser g s i =
  match s with
  | [] -> ()
  | j :: s' ->
      g.(j) <- inserer i g.(j);
      symetriser g s' i
```

Remarque 9. Dans le cas d'un graphe pondéré, on peut encore utiliser une représentation par liste d'adjacence, mais cette fois-ci `g.(i)` contient la liste des couples `(j, pij)` où j décrit l'ensemble des voisins de i et p_{ij} est le poids de l'arête (i, j).

```
type graphe = (int * float) list array
```

II.2 Représentation par matrices d'adjacence

Une autre représentation possible d'un graphe est la représentation par matrice d'adjacence. Donnons nous un graphe $G = (S, A)$ où l'on suppose encore que $S = [0, n-1]$. On représente le graphe G par une matrice $M \in \mathcal{M}_n(\{0, 1\})$ où $m_{ij} = 0$ si j n'est pas voisin de i et $m_{ij} = 1$ si j est voisin de i. On a ainsi

```
type graphe = int array array
```

Remarque 10. Inconvénient de cette approche : Il est coûteux d'obtenir la liste des voisins d'un sommets. Avantage : il est facile de savoir si deux sommets sont voisins.

Remarque 11. Dans le cas d'un graphe pondéré, on peut stocker à la ligne i et à la colonne j de la matrice d'adjacence le poids de l'arête (i, j) (et $+\infty$ si j n'est pas voisin de i).

Les fonctions écrites pour la représentation par listes d'adjacence deviennent, pour un graphe orienté non pondéré :

```
let voisins g i =
  let n = Array.length g in
  let s = ref [] in
  for i = 0 to n - 1 do
    if g.(i).(j) <> 0 then s := j :: !s
  done;
  List.rev !s

let ajouter_arete g i j = g.(i).(j) <- 1
let supprimer_arete g i j = g.(i).(j) <- 0
let sont_voisins g i j = g.(i).(j) <> 0
```

Complexité de `voisins` : $O(n)$ où n est le nombre de sommets. Complexité de l'insertion et de la suppression d'une arête, du test de voisinage : $O(1)$.

Tout ceci peut être facilement adapté au cas des graphes pondérés ou non orientés.

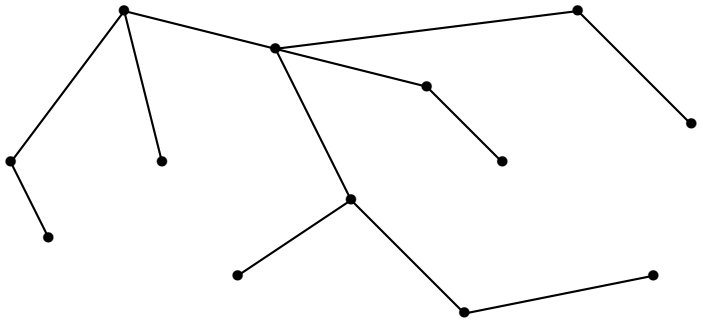
III Parcours d'un graphe

Dans tout ce qui suit, on supposera les chemins formés de sommets distincts, sauf pour les cycles qui ont leurs deux extrémités égales. Par ailleurs, on travaille sur des graphes non orientés, bien que ce que nous dirons s'adapte aux graphes orientés.

III.1 Arbres

Définition 16. Un arbre est un graphe (non orienté) connexe sans cycle. Un arbre enraciné est un couple (T, r) où T est un arbre et r est sommet de T, sa racine.

Exemple. Voici un arbre. Il possède 13 sommets et 12 arêtes. 6 des sommets sont de degré 1 : ce sont les feuilles de l'arbre. Parmi les 7 autres sommets, 4 sont de degré 2, 2 sont de degré 3, et 1 est de degré 4. On constate évidemment que la somme des degrés est 24, le double du nombre d'arêtes.



Proposition 8. *Dans un arbre, il existe un unique chemin entre deux sommets quelconques.*

Démonstration. L'existence vient de la connexité. Soient P_1 et P_2 deux chemins distincts entre les sommets u et v. P_1 et P_2 ont la même origine. Soit w le premier point d'intersection de ces chemins entre u exclus et v inclus. En juxtaposant la première partie de P_1 et la première partie de P_2 « à l'envers », on obtient un cycle. □

Notation. L'unique chemin dans l'arbre T entre les sommets u et v sera noté uTv. Si v est sur l'unique chemin reliant u à w, on notera $uTvTw$. Etc.

Définition 17. Soit (T, r) un arbre enraciné. On définit une relation « u est un ancêtre de v », lorsque l'unique chemin de r à v passe par le sommet u.

Remarque 12. Tout sommet est un ancêtre de lui-même, selon cette définition. On prendra quelquefois la notion d'ancêtre au sens strict (i.e. ancêtre et distinct), sans forcément le préciser.

Proposition 9. *La relation « ancêtre » est une relation d'ordre (en général partiel) sur l'ensemble des sommets de l'arbre T.*

Démonstration.

- La réflexivité est évidente.
- Soient u, v deux sommets de (T, r). Supposons que chacun de ces sommets est un ancêtre de l'autre et qu'ils sont distincts. Le sommet v étant ancêtre de u, on a un chemin $rTvTu$ de r à u. Mais u est ancêtre de v, donc $rTv = rTuTv$. Mais alors $rTu = rTvTu = rTuTvTu$ contient deux sommets égaux à u, contradiction. La relation est donc antisymétrique.
- Soient u, v, w trois sommets de (T, r). Supposons que u est ancêtre de v et v est ancêtre de w. On a donc le chemin $rTvTw$. Et $rTv = rTuTv$ puisque u est ancêtre de v. Donc, on a le chemin $rTuTvTw$. u est l'un des sommets du chemin rTw donc u est bien un ancêtre de w.

□

Proposition 10. *Soit T un arbre ayant au moins deux sommets. Alors T contient au moins deux sommets de degré 1.*

Remarque 13. Un sommet de degré 1 est appelé un sommet *pendant*.

Démonstration. Soit $P = (x_0, \ldots, x_n)$ un chemin de longueur maximale dans T. Supposons que le sommet x_n est de degré au moins 2. x_n a donc un voisin différent de x_{n-1}. Ce ne peut être l'un des x_i, $i = 0, \ldots, n-2$, sinon on aurait un cycle dans l'arbre. Ce ne peut être un autre sommet de l'arbre, sinon cela contredirait la maximalité de P. Donc $d_T(x_n) = 1$. De même, $d_T(x_0) = 1$. □

Proposition 11. *Soit T un arbre. Soient n et m le nombre de sommets et le nombre d'arêtes de T. Alors $m = n - 1$.*

Démonstration. On montre $m = n - 1$ par récurrence sur le nombre de sommets n. C'est évident si $n = 1$ ou 2. Supposons la propriété vraie pour tous les arbres ayant n sommets. Soit T un arbre ayant $n + 1$ sommets. L'arbre T a un sommet x de degré 1. On enlève de T le sommet x et l'unique arête issue de x. Il est clair que le graphe T' obtenu est encore un arbre, qui possède n sommets. On a donc $m' = n - 1$ d'où $m = m' + 1 = (n + 1) - 1$. □

Le théormème précédent admet des semi-réciproques.

Proposition 12. *Soit $G = (S, A)$ un graphe connexe non vide. Soient n et m le nombre de sommets et le nombre d'arêtes de G. Si $m = n - 1$ alors G est un arbre.*

Démonstration. Vérifions d'abord que si $n \geq 2$, alors G possède un sommet pendant. Supposons que cela ne soit pas le cas. Comme G est connexe, tous les sommets sont de degré au moins 2. On a donc $2m = \sum_{x \in S} d(x) \geq 2n = 2(m + 1)$. Donc $0 \geq 1$, visiblement une contradiction.

Maintenant, montrons la propriété par récurrence sur n. C'est clair pour $n = 1$. Soit n un entier non nul, supposons la propriété vraie pour tous les graphes à n sommets. Soit $G = (S, A)$ un graphe ayant $n + 1$ sommets. G a un sommet pendant, x. Soit $G' = (S', A')$ le graphe obtenu à partir

de G en enlevant x et l'unique arête d'origine x. On a $|S'| = |A'| + 1$, donc par l'hypothèse de récurrence, G' est un arbre. En remettant le sommet x et l'arête issue de x, on retrouve le graphe G qui est bien connexe et acyclique (vérification laissée en exercice). □

Remarque 14. L'hypothèse de connexité ne peut pas être enlevée. Par exemple soit $G = (A, S)$ la « réunion » des graphes complets K_3 et K_2. Le graphe G n'est ni connexe ni acyclique. Pourtant, $|S| = 3 + 2 = 5$ et $|A| = 3 + 1 = 4 = |S| - 1$.

Proposition 13. *Soit $G = (S, A)$ un graphe acyclique ayant n sommets et m arêtes. Si $n = m + 1$, alors G est un arbre.*

Démonstration. Soit k le nombre de composantes connexes de G. Pour $i = 1, \ldots, k$, soit $G_i = (S_i, A_i)$ la $i$ème composante connexe de G. G_i est connexe, et acyclique en tant que sous-graphe de G. Donc G_i est un arbre. Donc $|S_i| = |A_i| + 1$. On additione tout : $\sum_{i=1}^{k} |S_i| = |S| = \sum_{i=1}^{k} (|A_i| + 1) = |A| + k$. Mais $|S| = |A| + 1|$, donc $k = 1$: G a une unique composante connexe, il est connexe. □

III.2 Notion de parcours

On grand nombre d'algorithmes sur les graphes nécessitent le parcours du graphe en question.

On part d'un sommet donné et on explore le graphe à partir de celui-ci pour atteindre tous les sommets accessibles (tous les sommets si le graphe est connexe) depuis le sommet de départ. À chaque rencontre d'un sommet non encore découvert, on effectue une opération (qui dépend du problème que l'on cherche à résoudre, du type de parcours, etc.) sur ce sommet.

Un parcours de graphe crée en général un *arbre couvrant* de la composante connexe du graphe contenant le sommet d'origine du parcours. Chaque sommet découvert a pour père le sommet à partir duquel il a été découvert. Nous allons étudier deux algorithmes de parcours de graphes, le parcours en largeur et le parcours en profondeur. Chacun d'entre eux apporte des informations différentes sur le graphe parcouru.

III.3 Parcours en largeur - Plus court chemin

L'algorithme de parcours en largeur (Breadth First Search) utilise une file d'attente (FIFO) initialement vide. On place un sommet r dans la file : ce sera la racine de l'arbre couvrant. À chaque étape, on examine le sommet x de la file (sans l'enlever) et on regarde ses voisins y non encore visités. On effectue quelques mises à jour et on met ces voisins y dans la file. Puis on enlève x de la file.

La convention adoptée dans le code ci-dessous est qu'un sommet mis dans la file est « colorié en noir ». Concrètement, ceci peut être réalisé avec un tableau auxiliaire de booléens, avec accés et coloriages en $O(1)$. Au départ, tous les sommets sont blancs.

La fonction ci-dessous (écrite en pseudo-code) prend en paramètres un graphe connexe G et un sommet racine r. Elle renvoie

- une fonction p qui à chaque sommet associe son père dans un arbre couvrant enraciné en r du graphe G.

- Une fonction ℓ telle que pour tout sommet v, $\ell(v) = d_G(r, v)$, où $d_G(r, v)$, la distance de r à v dans G, est la longueur du plus court chemin de r à v dans le graphe G.
- Une fonction de temps t. Pour tout sommet v, $t(v)$ est le numéro de découverte du sommet v. On démarre à $t(r) = 1$.

```
1  fonction bfs(G, r) :
2    i <- 1
3    Q <- file_vide
4    p(r) <- rien; l(r) <- 0; t(r) <- 1
5    ajouter r à Q
6    colorier r en noir
7    while (non_vide Q) do
8      x <- premier element de Q
9      pour chacun des voisins blancs y de x :
10       i <- i + 1
11       p(y) <- x; l(y) <- l(x) + 1; t(y) <- i
12       ajouter y à Q
13       colorier y en noir
14     supprimer x de Q
15   done
16   (p, l, t)
```

Remarque 15. Si l'on appelle cette fonction avec un graphe G non connexe, on obtient un arbre couvrant de la composante connexe de r. On peut bien évidemment tester s'il existe des sommets de G non encore découverts, rappeler cette fonction sur l'un de ces sommets, etc. On obtient alors, pour un graphe pas forcément connexe, une *forêt couvrante* du graphe.

Ci-dessous un parcours en largeur d'un graphe. Pour chacun des sommets est indiqué le temps de découverte. L'état de la file d'attente au cours de l'exécution est

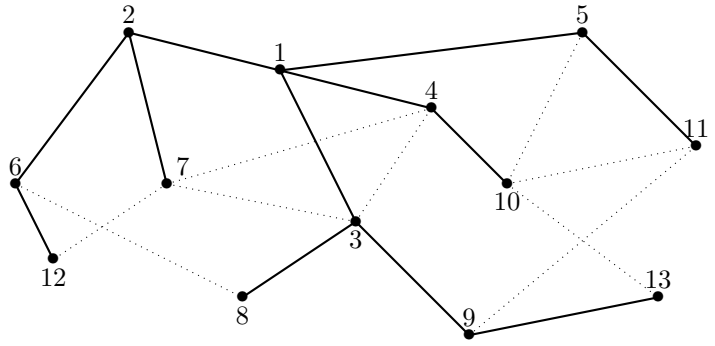
$$\emptyset \to 1 \to 1,2 \to 1,2,3 \to 1,2,3,4 \to 1,2,3,4,5 \to 2,3,4,5 \to 2,3,4,5,6 \to 2,3,4,5,6,7$$

$$\to 3,4,5,6,7 \to 3,4,5,6,7,8 \to 3,4,5,6,7,8,9 \to 4,5,6,7,8,9 \to 4,5,6,7,8,9,10$$

$$\to 5,6,7,8,9,10 \to 5,6,7,8,9,10,11 \to 6,7,8,9,10,11 \to 6,7,8,9,10,11,12$$

$$\to 7,8,9,10,11,12 \to 8,9,10,11,12 \to 9,10,11,12 \to 9,10,11,12,13 \to 10,11,12,13$$

$$\to 11,12,13 \to 12,13 \to 13 \to \emptyset$$

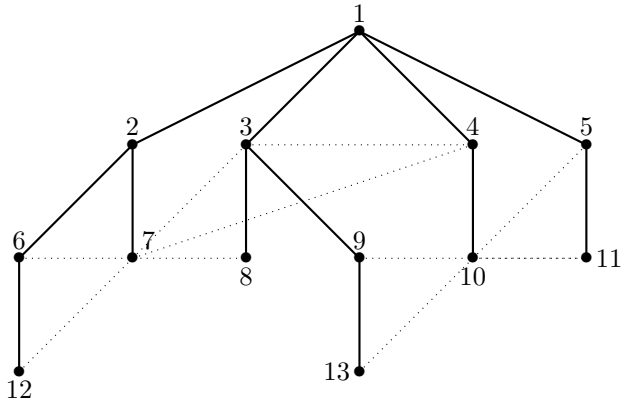


Proposition 14. *La complexité de la fonction* `bfs` *est* $O(m+n)$ *où* m *est le nombre d'arêtes du graphe* G *et* n *le nombre de ses sommets.*

Démonstration. La file d'attente peut facilement être implémentée de sorte que les opérations de file soient en $O(1)$. Chaque sommet est enfilé une unique fois car lors de son enfilement il est colorié en noir et on n'enfile que des sommets blancs. On effectue donc au plus n défilements, où n est le nombre de sommets de G. Comme à chaque itération de la boucle `while` on défile le sommet x de la file, on a au plus n itérations, une par sommet du graphe G. On ne peut pas défiler plus que ce que l'on a enfilé !

À chaque itération, le sommet x examiné est différent. On scanne la liste des voisins de x, ce qui donne une complexité en $O(d(x)+1)$ où $d(x)$ est le degré du sommet x. Le $+1$ permet de tenir compte du cas où x n'a aucun voisin. On fait quand même quelque chose dans ce cas là ! On a donc une complexité totale en $O(\sum_x d(x)+1) = O(2m)+O(n) = O(m+n)$. □

Voici une autre représentation du résultat de l'algorithme sur l'exemple ci-dessus. L'arbre couvrant est cette fois représenté sous une forme plus standard.



On constate sur l'exemple que les arêtes du graphe G qui ne font pas partie de l'arbre couvrant (en pointillés sur le dessin) relient uniquement des noeuds qui sont soit à la même profondeur dans l'arbre, soit à des profondeurs qui diffèrent de 1. En conséquences, ces noeuds appartiennent évidemment à des branches différentes de l'arbre couvrant. Ce résultat est général, c'est ce que nous allons maintenant démontrer.

Lemme 15. *Soit* (T,r) *l'arbre enraciné obtenu par l'algorithme de parcours en largeur sur le graphe connexe* G *à partir du sommet* r. *Alors, pour chaque sommet* v *de* G, $\ell(v) = d_T(r,v)$, *la distance de* r *à* v *dans l'arbre* T.

Démonstration. On a $\ell(r) = 0$ et, pour tout sommet $v \neq r$, $\ell(v) = \ell(p(v))+1$. Le premier point est donc acquis par une récurrence sur la distance des sommets à r dans l'arbre T. □

Lemme 16. u *et* v *sont deux sommets tels que* $\ell(u) < \ell(v)$, *alors le sommet* u *a été mis dans la file d'attente* Q *avant le sommet* v.

Démonstration. C'est une récurrence sur $\ell(u)$.

- C'est clair si $\ell(u) = 0$ puisque dans ce cas $u = r$, la racine de T. Soit $k > 0$.
- Supposons la propriété vraie pour tout sommet u tel que $\ell(u) < k$. Soit u tel que $\ell(u) = k$. Soient $x = p(u)$ et $y = p(v)$. La ligne 11 de l'algorithme montre que $\ell(x) = \ell(u) - 1 < \ell(v) - 1 = \ell(y)$. Par l'hypothèse de récurrence, x a été mis dans la file Q avant y. Donc u, voisin de x, a été mis dans la file Q avant v, voisin de y.

□

Lemme 17. *Soit (T, r) l'arbre enraciné obtenu par l'algorithme de parcours en largeur sur le graphe connexe G à partir du sommet r. Soit $\{u, v\}$ une arête du graphe G. Alors $|\ell(u)-\ell(v)| \leq 1$.*

Démonstration. Il suffit de prouver que si $\ell(u) < \ell(v)$, alors $\ell(u) = \ell(v) - 1$. Si $u = p(v)$, c'est terminé. Sinon, soit $y = p(v)$. Comme v a été ajouté à la file Q par l'arête $\{y, v\}$, et non par l'arête $\{u, v\}$, Le sommet y a été mis dans Q avant le sommet u. Donc, on n'a pas $\ell(u) < \ell(y)$ d'après le premier paragraphe. Donc $\ell(y) \leq \ell(u)$ d'après le lemme précédent. Donc, $\ell(v) - 1 = \ell(y) \leq \ell(u) \leq \ell(v) - 1$ d'où le résultat. □

Proposition 18. *Soit G un graphe connexe. Les valeurs de la fonction ℓ renvoyées par l'algorithme du parcours en profondeur sont les distances de la racine r aux sommets du graphe. Pour tout sommet v de G :*

$$\ell(v) = d_G(r, v)$$

Démonstration. On sait par le premier lemme que $\ell(v) = d_T(r, v)$. De plus, comme T est un sous-graphe de G, on a $d_T(r, v) \geq d_G(r, v)$.

Pour l'autre inégalité on raisonne par récurrence sur la longueur d'un plus court chemin de r à v. Soit P un plus court chemin de r à v dans G. Soit u le prédécesseur de v sur le chemin P. On a clairement $d_G(r, u) = d_G(r, v) - 1$. Sinon cela contredirait la minimalité du chemin P de r à v. Par l'hypothèse de récurrence, $\ell(u) \leq d_G(r, u)$. Or, d'après le second lemme, $|\ell(u) - \ell(v)| \leq 1$. Donc $\ell(v) \leq \ell(u) + 1 \leq d_G(r, u) + 1 = d_G(r, v)$. □

III.4 Parcours en profondeur

L'algorithme de parcours en profondeur (Depth First Search) utilise une pile (LIFO) initialement vide. On place la racine dans la pile. À chaque étape, on examine le sommet x de la pile et on regarde s'il a un voisin non encore visité. Si c'est le cas, on effectue quelques mises à jour et on met le voisin dans la pile. Sinon, on enlève x de la pile.

La fonction ci-dessous (écrite en pseudo-code) prend en paramètres un graphe connexe G et un sommet racine r. Elle renvoie

- une fonction p qui à chaque sommet associe son père dans un arbre couvrant enraciné en r du graphe G.
- Une fonction f telle que pour tout sommet v, $f(v)$ est l'instant de l'empilement de v lors du parcours en profondeur (f comme first).

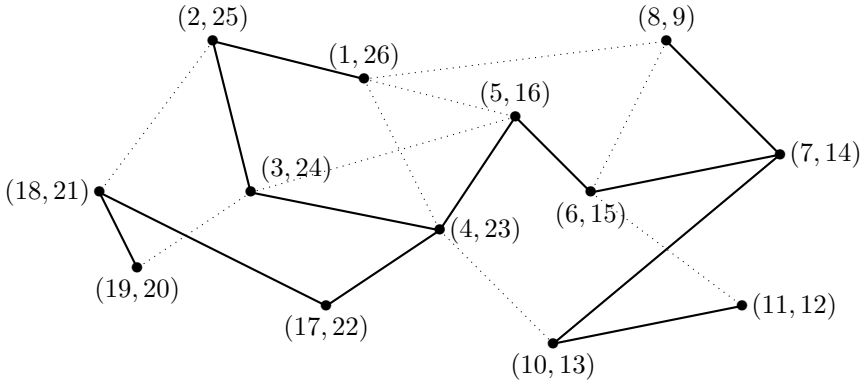
- Une fonction ℓ telle que pour tout sommet v, $\ell(v)$ est l'instant du dépilement de v lors du parcours en profondeur (ℓ comme last).

```
1  fonction dfs(G, r) :
2    i <- 1;
3    S <- pile vide
4    colorier r en noir
5    f(r) <- 1; p(r) <- rien
6    ajouter r à S
7    while (non_vide S) do
8      x <- premier element de S
9      i <- i + 1
10     if (x a un voisin non colorié y) then
11       colorier y en noir
12       p(y) <- x; f(y) <- i
13       ajouter y à S
14     else
15       l(x) <- i
15       supprimer x de S
16   done
17   (p, f, l)
```

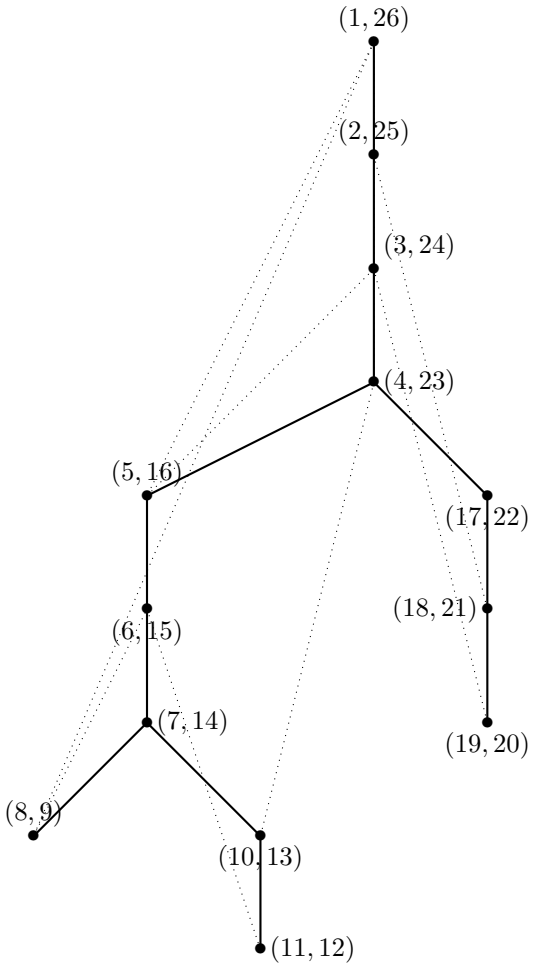
Remarque 16. L'algorithme présenté est un peu plus compliqué que celui du parcours en largeur, afin de pouvoir donner des valeurs correctes à la fonction f. À chaque itération, un parcours de la liste des voisins de x est nécessaire pour *chaque* recherche d'un voisin blanc de x. Cela donne une complexité en $O(\sum_x d(x)^2) = O(\sum_x d(x))^2 = O(m^2)$ où m est le nombre d'arêtes du graphe G. Une écriture plus soigneuse de la fonction permet cependant d'obtenir une complexité en $O(m)$.

Ci-dessous un parcours en profondeur du même graphe que dans le paragraphe précédent. Pour chacun des sommets v est indiqué le couple $(f(v), \ell(v))$. L'état de la pile au cours de l'exécution est indiqué ci-dessous. Les sommets sont numérotés par leur ordre d'empilement.

$$\emptyset \to 1 \to 1,2 \to 1,2,3 \to 1,2,3,4 \to 1,2,3,4,5 \to 1,2,3,4,5,6 \to 1,2,3,4,5,6,7$$
$$\to 1,2,3,4,5,6,7,8 \to 1,2,3,4,5,6,7 \to 1,2,3,4,5,6,7,10 \to 1,2,3,4,5,6,7,10,11$$
$$\to 1,2,3,4,5,6,7,10 \to 1,2,3,4,5,6,7 \to 1,2,3,4,5,6 \to 1,2,3,4,5$$
$$\to 1,2,3,4 \to 1,2,3,4,17 \to 1,2,3,4,17,18 \to 1,2,3,4,17,18,19$$
$$\to 1,2,3,4,17,18 \to 1,2,3,4,17 \to 1,2,3,4 \to 1,2,3 \to 1,2 \to 1 \to \emptyset$$



Que remarque t-on pour le parcours en profondeur lorsqu'on représente l'arbre couvrant sous forme « arborescente » ? Voici le dessin.



La situation est très différente de celle du parcours en largeur. Maintenant, les arêtes en pointillés vont toujours d'un sommet à un sommet sur la même branche de l'arbre couvrant. On voit également que sur les branches de l'arbre couvrant, en partant de la racine, ℓ décroît et f croît. Enfin, on voit (ou on ne voit pas jusqu'à ce qu'on nous l'ait dit) que cette propriété est caractéristique des sommets sur une même branche. Tout ceci est bien sûr général comme nous allons le montrer.

Proposition 19. *Soient G un graphe connexe et T l'arbre couvrant renvoyé par un parcours en profondeur de G. Soient u et v deux sommets* **adjacents** *de G tels que $f(u) < f(v)$. Alors $\ell(u) > \ell(v)$.*

Démonstration. Selon les lignes 8 à 12 de la fonction `dfs`, le sommet u est retiré de la pile S uniquement lorsque tous ses fils potentiels (les voisins non coloriés de u) dans T ont été empilés. L'un de ces voisins est v, puisque $f(u) < f(v)$. Ainsi, v est ajouté à la pile alors que u y est encore, donc u ne peut pas être retiré de S tant que v y est. Ainsi, $\ell(u) > \ell(v)$. □

Proposition 20. *Soient G un graphe connexe et T l'arbre couvrant renvoyé par un parcours en profondeur de G. Soient u et v deux sommets de G tels que $f(u) < f(v)$. Alors u est un ancêtre de v dans T si et seulement si $\ell(u) > \ell(v)$.*

Démonstration. Supposons que u est un ancêtre de v dans T. Les lignes 9 et 12 de la fonction `dfs` montrent que f croît le long du chemin de u à v. On applique la proposition précédente pour en déduire que $l(v) < l(u)$.

Inversement, supposons que u n'est pas un ancêtre de v dans l'arbre T. Comme $f(u) < f(v)$, v n'est pas non plus un ancêtre de u, d'après le sens direct de la démonstration. Ainsi, u n'est pas l'un des sommets de l'unique chemin de r à v dans T, et de même v n'est pas l'un des sommets de l'unique chemin de r à u dans T. Soit s le dernier sommet commun de ces deux chemins (s est le plus proche ancêtre commun de u et v dans T). Comme $f(u) < f(v)$, un argument déjà vu montre que les descendants de s sur le chemin de s à v ont été empilés avant que les descendants de s sur le chemin de s à u n'aient été dépilés. En particulier, v a été empilé avant que u n'ait été dépilé. Ainsi, $l(u) < l(v)$. □

Proposition 21. *Soit T l'arbre couvrant obtenu par un parcours en profondeur du graphe G. Soient u et v deux sommets voisins dans le graphe G. Alors u est un ancêtre de v dans T ou le contraire.*

Démonstration. Supposons par exemple que $f(u) < f(v)$. Par le premier point de la proposition précédente, on a alors $l(v) < l(u)$. Par le second point, u est un ancêtre de v dans l'arbre T. □

IV Plus courts chemins dans un graphe

IV.1 Notion de plus court chemin

Définition 18. Soit $G = (S, A, f)$ un graphe pondéré (orienté ou non). Soit $P = (x_0 = x, x_1, \ldots, x_n = y)$ un chemin du sommet x au sommet y. Le poids de P est le réel $f(P) = \sum_{i=0}^{n-1} f(\{x_i, x_{i+1}\})$. La distance $d(x, y)$ de x à y est le poids minimal des chemins reliant x à y s'il en existe, et $+\infty$ sinon.

Remarque 17. Le terme « distance » réfère évidemment au cas où les poids des arêtes sont de vraies distances. On a le résultat suivant :

Proposition 22. *Soit $G = (S, A, f)$ un graphe pondéré connexe non orienté. Si la fonction f est à valeurs strictement positives, d est une distance sur l'ensemble des sommets de G.*

Démonstration. Le poids de chaque arête étant strictement positif, le chemin trivial d'un sommet à lui-même est le seul chemin de poids nul. Ainsi, $d(x, y) \geq 0$ et $d(x, y) = 0$ si et seulement si $x = y$.

La symétrie de la fonction d est évidente (graphe non orienté).

Soient enfin trois sommets x, y, z. Soient P un plus court chemin de x à y et Q un plus court chemin de y à z. Alors la concaténation de P et Q est un chemin de x à z de poids $d(x, y) + d(y, z)$. Donc $d(x, z)$ est inférieur à cette valeur. D'où $d(x, z) \leq d(x, y) + d(y, z)$. □

IV.2 Plus court chemins dans les graphes non pondérés

Soient u et v deux sommets d'un graphe non pondéré $G = (S, A)$. Le graphe G peut aussi être considéré comme un graphe pondéré dont toutes les arêtes sont de poids 1. Ce que nous avons démontré dans la section précédente, c'est que pour trouver le plus court chemin de u à v, il sufffit d'effectuer un parcours en largeur de G à partir du sommet u.

Dans le cas d'un graphe pondéré, il va falloir parcourir le graphe différemment. Nous allons étudier dans la suite deux algorithmes. L'algorithme de Floyd-Warshall n'est pas un algorithme de parcours : il calcule la longueur des plus courts chemins entre tous les couples de sommets, mais il ne donne pas ces chemins. L'algorithme de Dijkstra détermine quant à lui les plus courts chemins entre un sommet et tous les autres sommets.

IV.3 L'algorithme de Floyd-Warshall

Soit $G = (S, A, f)$ un graphe pondéré. On suppose que le graphe ne possède pas de cycle de poids strictement négatif.

On suppose le graphe G donné par sa matrice d'adjacence $M \in \mathcal{M}_n(\mathbb{R} \cup \{+\infty\})$, où n est le nombre de sommets du graphe G. Les sommets de G sont donc, pour simplifier, les entiers de 1 à n. Pour i, j entre 1 et n, M_{ij} est le poids de l'arête du sommet i vers le sommet j si une telle arête existe, et $+\infty$ sinon.

On définit par récurrence sur k une suite $(M^{(k)})_{k\in\mathbb{N}}$ en posant $M^{(0)} = M$ et, pour tout $k \in \mathbb{N}$ et tous $i, j \in [1, n]^2$,

$$M_{i,j}^{(k+1)} = \min\left(M_{i,j}^{(k)}, M_{i,k+1}^{(k)} + M_{k+1,j}^{(k)}\right)$$

On étend évidemment l'opération d'addition à $\mathbb{R} \cup \{+\infty\}$ en posant $x + \infty = \infty + x = \infty$ pour tout x réel ou infini.

Proposition 23. *Pour tous $i, j \in [1, n]$ et tout $k \leq n$ tel que $M_{i,j}^{(k)} \neq +\infty$, le réel $M_{i,j}^{(k)}$ est le poids minimal des chemins de i à j dont les sommets intermédiaires (i.e. hormis i et j) sont des sommets de $[1, k]$.*

Démonstration. On fait une récurrence sur k. Pour $k = 0$ c'est clair : il existe un unique chemin de i à j n'ayant aucun sommet intermédiaire, c'est (i, j) et son poids est le poids de l'arête (i, j), c'est à dire $M_{i,j} = M_{i,j}^{(0)}$. Supposons maintenant la propriété vraie pour un entier $k < n$. Soit c un chemin de i à j dont les sommets intermédiaires sont dans $[1, k + 1]$.

— Si les sommets intermédiaires de c sont dans $[1,k]$, alors le poids de c est supérieur ou égal à $M_{i,j}^{(k)}$.
— Si au moins un sommet intermédiaire est égal à $k+1$, c est alors un chemin c_1 de i à $k+1$ dont les sommets intermédiaires sont dans $[1,k]$, suivi de cycles de $k+1$ à $k+1$, suivi d'un chemin c_2 de $k+1$ vers j dont les sommets intermédiaires sont dans $[1,k]$. Le poids des cycles étant positif, le poids de c est supérieur ou égal à la somme des poids de c_1 et c_2, elle-même supérieure à $M_{i,k+1}^{(k)} + M_{k+1,j}^{(k)}$.

On vient de démontrer que $M_{i,j}^{(k+1)}$ est supérieur ou égal au poids minimal qui nous intéresse. De plus, toujours grâce à l'hypothèse de récurrence, on vérifie aisément qu'un chemin de poids $M_{i,j}^{(k+1)}$ de i à j dont les sommets intermédiaires appartiennent à $[1,k+1]$ existe bien. D'où le résultat. □

Corollaire 24. *Pour tous $i,j \in [1,n]$, $M_{i,j}^{(n)}$ est le poids minimal des chemins de i à j.*

Démonstration. Car un chemin de i à j est un chemin dont les sommets intermédiares appartiennent à $[1,n]$. □

Proposition 25. *La complexité de l'algorithme de Floyd-Warshall est $O(n^3)$ où n est le nombre de sommets de G.*

Démonstration. Appelons complexité de l'algorithme le nombre d'opérations effectuées sur des nombres réels, éléments de matrices. Pour k variant de 0 à $n-1$, on calcule une matrice, donc n^2 coefficients. Chaque coefficient est obtenu à l'aide d'un min et d'une addition. On a donc une complexité égale à $2n^3$. □

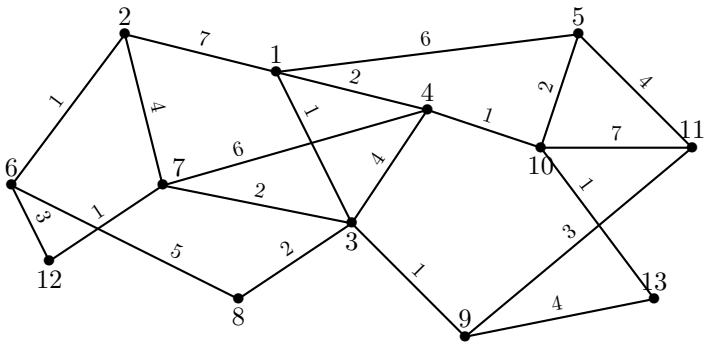
Voici la fonction Caml. Les tableaux Caml étant indexés à partir de 0, il convient évidemment de décaler les indices de 1 par rapport à la théorie.

```
let floyd_warshall g =
  let n = Array.length g in
  let a = ref (copy_matrix g) in
  for k = 0 to n - 1 do
    a := floyd_aux !a (k - 1)
  done;
  a
```

où la fonction `copy_matrix` est évidente et `floyd_aux` est définie par

```
let floyd_aux a k =
  let n = Array.length a in
  let b = Array.make_matrix n n 0 in
  for i = 0 to n - 1 do
    for j = 0 to n - 1 do
      b.(i).(j) <- min a.(i).(j) (a.(i).(k + 1) + a.(k + 1).(j))
    done
  done;
  b
```

Exemple. Considérons le graphe déjà utilisé pour les parcours en largeur et en profondeur, que nous pondérons cette fois ci. Nous prenons pour exemple un graphe non orienté mais l'algorithme de Floyd-Warshall fonctionne aussi très bien sur des graphes orientés.



Sa matrice d'adjacence est

$$M = M^{(0)} = \begin{pmatrix} 0 & 7 & 1 & 2 & 6 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 7 & 0 & \infty & \infty & \infty & 1 & 4 & \infty & \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & 4 & \infty & \infty & 2 & 2 & 1 & \infty & \infty & \infty & \infty \\ 2 & \infty & 4 & 0 & \infty & \infty & 6 & \infty & \infty & 1 & \infty & \infty & \infty \\ 6 & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty & 2 & 4 & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & 0 & \infty & 5 & \infty & \infty & \infty & 3 & \infty \\ \infty & 4 & 2 & 6 & \infty & \infty & 0 & \infty & \infty & \infty & \infty & 1 & \infty \\ \infty & \infty & 2 & \infty & \infty & 5 & \infty & 0 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 1 & \infty & \infty & \infty & \infty & \infty & 0 & \infty & 3 & \infty & 4 \\ \infty & \infty & \infty & 1 & 2 & \infty & \infty & \infty & \infty & 0 & 7 & \infty & 1 \\ \infty & \infty & \infty & \infty & 4 & \infty & \infty & \infty & 3 & 7 & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 3 & 1 & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 4 & 1 & \infty & \infty & 0 \end{pmatrix}$$

On obtient

$$M^{(13)} = \begin{pmatrix} 0 & 7 & 1 & 2 & 5 & 7 & 3 & 3 & 2 & 3 & 5 & 4 & 4 \\ 7 & 0 & 6 & 9 & 12 & 1 & 4 & 6 & 7 & 10 & 10 & 4 & 11 \\ 1 & 6 & 0 & 3 & 6 & 6 & 2 & 2 & 1 & 4 & 4 & 3 & 5 \\ 2 & 9 & 3 & 0 & 3 & 9 & 5 & 5 & 4 & 1 & 7 & 6 & 2 \\ 5 & 12 & 6 & 3 & 0 & 12 & 8 & 8 & 7 & 2 & 4 & 9 & 3 \\ 7 & 1 & 6 & 9 & 12 & 0 & 4 & 5 & 7 & 10 & 10 & 3 & 11 \\ 3 & 4 & 2 & 5 & 8 & 4 & 0 & 4 & 3 & 6 & 6 & 1 & 7 \\ 3 & 6 & 2 & 5 & 8 & 5 & 4 & 0 & 3 & 6 & 6 & 5 & 7 \\ 2 & 7 & 1 & 4 & 7 & 7 & 3 & 3 & 0 & 5 & 3 & 4 & 4 \\ 3 & 10 & 4 & 1 & 2 & 10 & 6 & 6 & 5 & 0 & 6 & 7 & 1 \\ 5 & 10 & 4 & 7 & 4 & 10 & 6 & 6 & 3 & 6 & 0 & 7 & 7 \\ 4 & 4 & 3 & 6 & 9 & 3 & 1 & 5 & 4 & 7 & 7 & 0 & 8 \\ 4 & 11 & 5 & 2 & 3 & 11 & 7 & 7 & 4 & 1 & 7 & 8 & 0 \end{pmatrix}$$

qui est bien la matrice des distances entre tous les sommets de G.

IV.4 L'algorithme de Dijkstra

Soit $G = (S, A, f)$ un graphe pondéré connexe. Soit r un un sommet racine. L'algorithme de Dijkstra construit un arbre couvrant T enraciné en r tel que pour tout sommet v de G, on ait $d_G(r, v) = d_T(r, v)$. Voici le pseudo-code de l'algorithme. Il prend en paramètres un graphe pondéré G et un sommet r et il renvoie un couple (p, ℓ) où p est la fonction père associé à un arbre couvrant enraciné en r et ℓ est la fonction distance à r dans le graphe G.

```
1  fonction dijkstra(G, r):
2    pour tous les sommets v:
3      p(v) <- 0; l(v) <- infini
4    l(r) <- 0
5    while (il y a un sommet blanc u tel que l(u) < infini) do
6      choisir un tel u tel que l(u) est minimum
7      colorier u en noir
8      pour chaque voisin non colorié v de u tel que l(v) > l(u) + f(u,v):
9        p(v) <- u
10       l(v) <- l(u) + f(u, v)
11   done
12   (p, l)
```

Pour que l'algorithme de Dijkstra ait une complexité intéressante, il faut choisir judicieusement les structures de données. Les fonctions p et ℓ, ainsi que l'ensemble des couleurs des sommets peuvent être représentés par des tableaux pour avoir des accès en lecture et écriture en $O(1)$. Pour que la ligne 6 soit exécutée rapidement, on peut utiliser une file de priorité où les priorités sont les distances (provisoires). On remarque que les priorités doivent être diminuées lors des itérations. Une méthode consiste par exemple à implémenter la file de priorité par un tableau, et de maintenir en parallèle un tableau `aux` tel que `aux.(v)` contienne la position du sommet v dans la file de priorité. Ce tableau permet égamement de savoir si un sommet est dans la file ou pas. Une diminution de la distance de r à v consiste à faire remonter v, dont on connait la position grâce au tableau auxiliaire, dans la file de priorité (tout en maintenant le tableau auxiliaire à jour). L'algorithme devient ainsi :

```
1  fonction dijkstra(G, r):
2    pour tous les sommets v:
3      p(v) <- 0; l(v) <- infini
4    l(r) <- 0
5    F <- file vide
6    inserer r dans F
6    while (F non vide) do
7      oter u tel que l(u) est minimum de F
8      colorier u en noir
9      pour chaque voisin blanc v de u tel que l(v) > l(u) + f(u,v):
10       p(v) <- u
11       l(v) <- l(u) + f(u, v)
12       si v n'est pas dans F: inserer v dans F
13       sinon: diminuer la priorite de v dans F
14   done
15   (p, l)
```

Proposition 26. *Soient n et m le nombre de sommets et le nombre d'arêtes du graphe G. La complexité de l'algorithme de Dijkstra est en $O((n+m)\log n)$.*

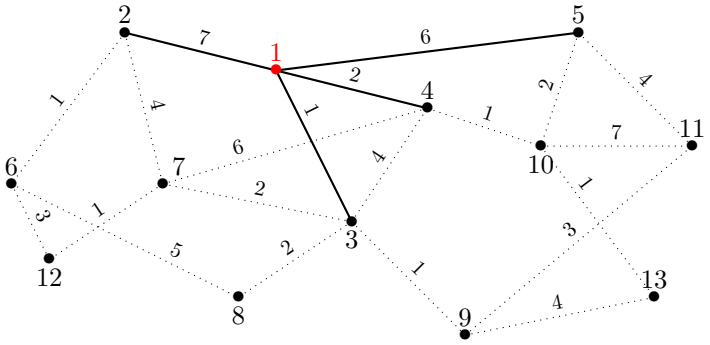
Démonstration.

- Un sommet n'est inséré dans F que s'il est blanc (ligne 12) et un sommet retiré de F est colorié en noir. Un sommet ne peut donc pas être inséré deux fois. Ainsi, on a $O(n)$ itérations pour la boucle `while`.
- Ligne 7 : complexité $O(\log n)$. On suppose la file de priorité implémentée de façon standard.
- Boucle lignes 9-13 : Pour chaque sommet u retiré de F, on traite ses voisins, on a donc $d_G(u)$ opérations. Pour chaque voisin de u, on diminue sa priorité ou on l'insère dans la file : complexité $O(\log n)$.

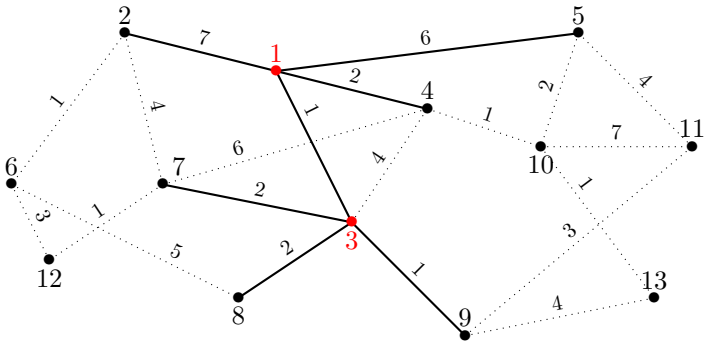
La complexité de l'algorithme de Dijkstra est donc $O(\sum_u(\ln n+d(u)\ln n)) = O(n\ln n+\ln n\sum_u d(u)) = O((n+m)\ln n)$. Pour un graphe ayant très peu d'arêtes en comparaison du nombre de sommets, la complexité est donc $O(n\ln n)$. Pour un graphe ayant beaucoup d'arêtes, ce sera $O(m\ln n)$. □

Ci-dessous, une simulation de l'algorithme de Dijkstra sur notre exemple maintenant classique, que nous avons cette fois-ci pondéré. L'arbre couvrant calculé par l'algorithme est en traits gras, le sommet source est le sommet numéro 1. La distance de chaque sommet u au sommet source s'obtient en partant du sommet u et en additionnant les poids des arêtes en gras. Les sommets noircis sont mis en rouge (sic). Les sommets dans la file de priorité sont ceux qui sont adjacents à un sommet rouge via une arête en gras.

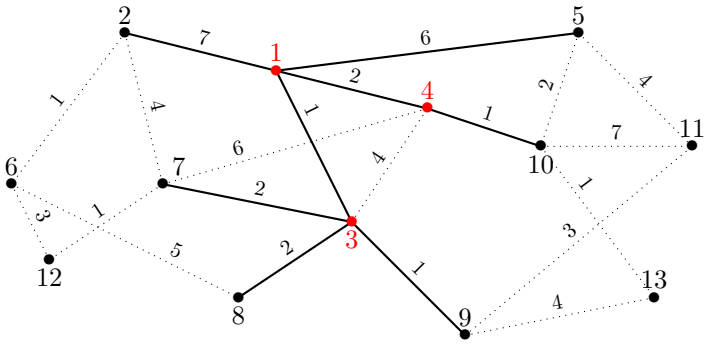
Après la première itération, nous obtenons



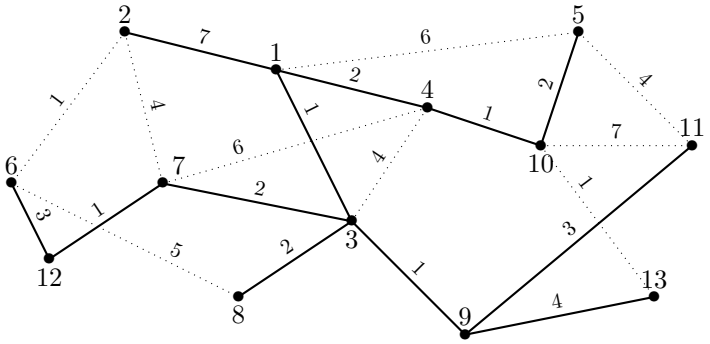
À la seconde itération, le sommet défilé est le sommet 3. À la fin de cette itération on obtient



À la troisième itération, le sommet défilé peut être le 4 ou le 9. Mettons que ce soit le sommet 4. À la fin de cette itération on obtient



Et ainsi de suite. Remarquer qu'un trait en gras ne le reste pas forcément. Regarder par exemple l'arête $\{1, 5\}$. Voici le résultat final :



Il nous reste à prouver la correction de l'algorithme.

Proposition 27. *Après chaque itération, les propriétés suivantes sont vérifiées :*

1. *Pour tout sommet noir du graphe G, $\ell(u) = d(r, u)$, la distance de r à u dans le graphe G.*
2. *Pour tout sommet blanc v de G adjacent à un sommet noir, $\ell(v)$ est la longueur du plus court chemin reliant r à v et dont tous les sommets, hormis v, sont noirs.*

Si le graphe G est connexe, l'algorithme termine lorsque tous les sommets sont noirs. On a donc le

Corollaire 28. *Lorsque l'algorithme termine, on a, pour tout sommet u de G, $\ell(u) = d_G(r, u)$.*

Démonstration. Remarque préliminaires :

— Tous les sommets de la file d'attente sont adjacents à un sommet noir. En effet, la seule façon d'être placé dans la file de priorité est d'être adjacent à un sommet u défilé qui est noirci à la fin d'une itération.

— Un sommet noir u ne voit pas la valeur de $\ell(u)$ modifiée.

On fait bien évidemment une récurrence sur k, le numéro de l'itération. Pour $k = 1$, le seul sommet noir est r. On a $\ell(r) = 0 = d_G(r, r)$. De plus, lors de la première itération, tous les voisins v de r sont placés dans la file et pour ces sommets $\ell(v) = f(r, v)$ qui est bien la longueur du plus court chemin de r à v ne contenant, hormis v, que r.

Supposons maintenant la propriété vraie pour un entier k. Soit u le sommet noirci à la $k + 1$-ième itération. D'après l'hypothèse de récurrence 2, $\ell(u)$ est la longueur du plus court chemin reliant r à u et n'utilisant que des sommets noirs. Soit P un chemin de r à u. Ce chemin emprunte des sommets noirs, puis une arête passant d'un sommet noir à un sommet blanc. La longueur de P est donc au moins $\ell(u)$, puisque le sommet u est choisi de sorte que $\ell(u)$ soit minimal parmi les sommets de la file. Ainsi, $d_G(r, u) \geq \ell(u)$. Par ailleurs, toujours par l'hypothèse 2, $d_G(r, u) \leq \ell(u)$. Le premier point est démontré.

Soit maintenant un sommet blanc v adjacent à un sommet noir. Par l'hypothèse 2, le plus court chemin reliant r à v en n'empruntant que des sommets noirs a pour longueur $\ell(v)$ à la fin de la $k$ième itération. Il faut maintenant considérer un tel chemin contenant u. Ce chemin part de r, arrive à u en n'empruntant que des sommets noirs, puis emprunte l'arête $\{u, v\}$. Bref c'est un chemin du type $P = (r, \ldots, u, v)$. Un tel chemin est de longueur $\ell(P) \geq d_G(r, u) + f(u, v) = \ell(u) + f(u, v)$ d'après la première partie de la preuve. Mais cette quantité est supérieure à $\ell(v)$. Donc $\ell(P) \geq \ell(v)$. Enfin la valeur affectée à $\ell(v)$ l'a été lors d'une itération de la boucle entre l'étape 1 et l'étape $k + 1$. Il existe donc un chemin P ne comportant que des sommets noirs et tel que $\ell(P) = \ell(v)$. D'où la propriété 2.

□

Chapitre 6

Langages et expressions Rationnels

I Motifs

I.1 Quest-ce qu'un motif ?

Le concept de motif est un concept extrêmement vaste. La recherche et la reconnaissance de motifs parmi des données sont une des préoccupations majeures de l'informatique moderne. Citons quelques exemples :

- reconnaissance de formes dans une image : visages sur une photographie, corps ensevelis dans des avalanches, caractères manuscrits.
- recherche de séquences dans des brins d'ADN.
- lecture de codes-barres ou de flashcodes.
- analyse du langage naturel.
- etc.

Sans donner une définition précise, un motif est un ensemble de formes qui se ressemblent, la notion de ressemblance restant à préciser en fonction du contexte. Prenons par exemple la recherche de croix dans une image : une croix possède deux branches, une hauteur, une largeur, une couleur. Ses deux branches se croisent selon un certain angle. Toutes les croix se ressemblent, d'une certaine façon.

Un très grand nombre d'outils mathématiques sont utilisés dans les méthodes modernes de reconnaissance de motifs : interpolation polynomiale, probabilités, intelligence articifcielle, réseaux de neurones, etc. Nous nous concentrerons dans ce qui va suivre sur des motifs unidimensionnels, que nous identifierons à des ensembles de mots sur un certain alphabet. Un code barre, par exemple, peut être vu comme un ensemble de mots sur l'alphabet $\{0,1\}$. Un numéro de carte bancaire est un mot sur l'alphabet $[0,9]$.

Nous prendrons dorénavant la définition suivante :

Définition 1. Soit A un alphabet. Un motif sur l'alphabet A est un langage sur l'alphabet A.

En clair, « motif » est synonyme de « langage ».

I.2 Quelques exemples naïfs de recherche de motifs

Nous allons, sur deux exemples, écrire des algorithmes permettant de savoir si un mot appartient à un motif donné.

Les multiples de 3 en base 2

On se place sur l'alphabet $A = \{0,1\}$. Soit $L \subseteq A^*$ l'ensemble des multilpes de 3 en base 2. On autorise le mot vide, on permet également qu'un mot commence par des zéros. Soyons rigoureux : si $u = x_{n-1} \dots x_0 \in L$, notons $\varphi(u) = \sum_{k=0}^{n-1} x_k 2^k$ le nombre représenté par u, avec la convention que $\varphi(\varepsilon) = 0$. On a pour tout $u \in A^*$, $\varphi(u0) = 2\varphi(u)$ et $\varphi(u1) = 2\varphi(u) + 1$. Ainsi, si $\varphi(u) \equiv 0[3]$,

$\varphi(u0) \equiv 0[3]$ et $\varphi(u1) \equiv 1[3]$. De même si $\varphi(u) \equiv 1[3]$ ou $\varphi(u) \equiv 2[3]$. Représentons un mot par une liste dont les éléments valent 0 ou 1. L'algorithme s'en déduit :

```
let rec psi u s =
  match u with
  | [] -> s
  | a :: u' -> psi u' ((2 * s + a) mod 3)

let multiple3 u = psi u 0 = 0
```

Un exemple avec des lettres

Prenons pour alphabet $A = \{a, b, c\}$. Soit L le langage sur A dont les mots u véifient la propriété suivante : le premier éventuel a de u apparaît avant le premier éventuel b, qui apparaît avant le premier éventuel c. Par exemple, les mot *abacab* et *bbcb* sont dans L, les mot *acaba* et *cca* n'y sont pas. La fonction `aux` ci-dessous prend deux paramètres : un mot u, liste de caractères et un triplet $x = (p, q, r)$. p, q, r valent 0 ou 1 selon qu'on a déjà lu un a, un b ou un c.

```
let rec aux u x =
  match (u, x) with
  | [], _ -> true
  | 'a' :: u', (0, 1, _) -> false
  | 'a' :: u', (0, _, 1) -> false
  | 'a' :: u', (_, q, r) -> aux u' (1, q, r)
  | 'b' :: u', (_, 0, 1) -> false
  | 'b' :: u', (p, _, r) -> aux u' (p, 1, r)
  | 'c' :: u', (p, q, _) -> aux u' (p, q, 1)
  | _ -> failwith "aux"

let dansL u = aux u (0, 0, 0)
```

I.3 Bilan

Que nous apprennent les deux exemples précédents ? Étant donné un motif particulier, il est possible la plupart du temps d'écrire un algorithme lui aussi particulier qui permet de décider si un mot appartient ou pas au motif. Les deux exemples précédents fournissent deux algorithmes qui n'ont apparemment pas de lien l'un avec l'autre.

On peut se demander s'il est possible de régler de façon générale la question pour un motif quelconque. Ceci demanderait tout d'abord de mettre au point un langage de description des motifs. Nous allons le faire pour une classe assez générale de motifs, en introduisant le concept d'expression rationnelle, ou expression régulière : les expressions rationnelles seront un moyen de décrire une classe de motifs appelés les langages rationnels.

Il faudra ensuite se poser la qestion suivante : étant donnée une expression rationnelle e décrivant un langage rationnel, peut-on écrire un algorithme (paramétré par e) décidant de l'appartenance d'un mot au langage en question ? La réponse est positive. Les automates finis sont des machines abstraites qui résolvent notre problème, et nous étudierons un algorithme qui permet de transformer une expression rationnelle en un automate fini : l'algorithme de Berry-Sethi.

II Expressions rationnelles

II.1 Le langage des expressions rationnelles

Définition 2. Soit A un alphabet. Soit l'alphabet $A' = A \cup \{\emptyset, \varepsilon, +, ., ^*, (,)\}$. Le langage des expressions rationnelles sur l'alphabet A est le plus petit langage $\mathcal{R}$ sur l'alphabet A' vérifiant :

- $A \subseteq \mathcal{R}$.
- $\emptyset \in \mathcal{R}$, $\varepsilon \in \mathcal{R}$.
- $\forall e, e' \in \mathcal{R}, (e + e') \in \mathcal{R}$.
- $\forall e, e' \in \mathcal{R}, (e.e') \in \mathcal{R}$.
- $\forall e \in \mathcal{R}, e^* \in \mathcal{R}$.

Exemple. Prenons $A = \{a, b\}$. Les expressions $(a + b)^*$, $(\varepsilon + ((a.b^*).a))$, sont des expressions rationnelles.

Remarque 1. Devant l'amoncellement des parenthèses, on fait les conventions suivantes : l'étoile est prioritaire devant ., elle même prioritaire devant +. De plus, + et . associent à gauche. Enfin, on ne note en général pas le point. Ainsi, par exemple, $(a+b)^*aba(a+b)^*$ est l'expression $(((((a+b)^*.a).b).a).(a+b)^*)$

Remarque 2. Informellement (pour l'instant), une expression rationnelle représente un ensemble de mots sur l'alphabet A. Le + représente une alternative, le . une concaténation, l'étoile un joker signifiant « 0, 1 ou plusieurs fois ce qui précède ». Par exemple, a^* représente tous les mots ne contenant que des a. $(a + b)^*aba(a + b)^*$ représente tous les mots contenant le facteur aba. Et $a(a + b)^* + (a + b)^*bb$ représente tous les mots commençant par a ou finissant par bb.

Exercice. Donner une définition par le bas (c'est à dire sous forme d'une réunion croissante de langages) du langage des expressions rationnelles.

II.2 Expressions rationnelles généralisées

On peut également souhaiter représenter des mots satisfaisant d'autres contraintes, comme par exemple les mots vérifiant deux conditions simultanées, ou encore les mots satisfaisant une condition mais pas une autre. On introduit pour cela deux nouveaux types d'expressions rationnelles (généralisées), qui sont $e \wedge e'$ et $e \setminus e'$. Nous verrons plus tard que les expressions rationnelles généralisées, bien qu'apportant une simplicité dans l'écriturent, ne permettent pas de décrire des ensembles de mots plus riches que les expressions rationnelles classiques.

III Langages rationnels

A désigne un alphabet. L'ensemble $\mathcal{P}(A^*)$ est l'ensemble de tous les langages sur A.

III.1 Notion de langage rationnel

Définition 3. On note $Rat(A^*)$ la plus petite partie de $\mathcal{P}(A^*)$ contenant les langages finis, et stable par union ($\cup$), concaténation (.), et fermeture de Kleene ($*$). On appelle langage rationnel (sur A) tout langage élément de $Rat(A^*)$.

Remarque 3. Une telle partie existe, c'est l'intersection de toutes les parties de $\mathcal{P}(A^*)$ stables par concaténation, union et fermeture de Kleene, et contenant les langages finis. C'est immédiat en remarquant qu'au moins une partie de $\mathcal{P}(A^*)$ vérifie ces propriétés : c'est $\mathcal{P}(A^*)$ lui-même.

III.2 Une construction par le bas des langages rationnels

On note Rat_0 l'ensemble des langages finis sur A.

Soit n un entier naturel. Supposons défini l'ensemble Rat_n. On définit alors Rat_{n+1} par

$$Rat_{n+1} = Rat_n \cup \{L_1 \cup L_2, L_1, L_2 \in Rat_n\} \cup \{L_1.L_2, L_1, L_2 \in Rat_n\} \cup \{L^*, L \in Rat_n\}$$

Théorème 1. $Rat(A^*) = \cup_{n=0}^{\infty} Rat_n$.

Démonstration. On a bien sûr $Rat_0 \subset Rat(A^*)$. Supposons, pour un certain entier n, que $Rat_n \subset Rat(A^*)$. Soit $L \in Rat_{n+1}$. Si L est de la forme $L_1 \cup L_2$ ou $L_1.L_2$ avec L_1 et L_2 dans Rat_n, alors $L \in Rat(A^*)$ puisque $Rat(A^*)$ est stable par union et par concaténation. Il en est de même si $L = L_1^*$ avec $L_1 \in Rat_n$. Ainsi, $\cup_{n=0}^{\infty} Rat_n \subset Rat(A^*)$.

Inversement, posons $Rat' = \cup_{n=0}^{\infty} Rat_n$. Soient L_1 et L_2 dans Rat'. Alors il existe un entier n tel que L_1 et L_2 soient dans Rat_n. Donc, $L_1.L_2$ et $L_1 \cup L_2$ sont dans Rat_{n+1}, donc dans Rat'. Ainsi, Rat' est stable par union et concaténation. De même, il est stable par fermeture de Kleene. Enfin, il contient Rat_0, c'est à dire les langages finis. Par minimalité de $Rat(A^*)$, on en déduit $Rat(A^*) \subset Rat'$. □

Exemple. Sur $A = \{a, b\}$, le langage $(a.(a \cup b^*))^*$ est dans Rat_4 (ou peut-être mieux).

Exercice. Le langage $(a^* \cup b^*)^*$ est rationnel. Trouver un entier n minimal pour lequel ce langage appartienne à Rat_n (il y a un piège).

III.3 Langages rationnels et expressions rationnelles

Soit $\Phi : \mathcal{R} \to Rat$ définie comme suit :

- $\Phi(\emptyset) = \emptyset$.
- $\forall \alpha \in A \cup \{\varepsilon\}, \Phi(\alpha) = \{\alpha\}$ (où $\{\varepsilon\}$ désigne le singleton formé par le mot vide).
- $\forall u, v \in R, \Phi((u + v)) = \Phi(u) \cup \Phi(v)$.
- $\forall u, v \in R, \Phi((u.v)) = \Phi(u).\Phi(v)$.
- $\forall u \in R, \Phi((u)^*) = \Phi(u)^*$.

Le lecteur se convaincra que :

- Une telle application est bien définie. Ceci résulte de la non-ambiguité du langage des expressions rationnelles. Pour plus de détails, se reporter au théorème de lecture unique du chapitre de logique.
- L'application Φ est surjective. Ceci peut être démontré par récurrence sur n pour les langages de Rat_n. Tout langage rationnel sera donc l'image d'une expression rationnelle.
- Φ n'est pas injective. Deux expressions rationnelles distinctes peuvent donner lieu au même langage rationnel. Par exemple, $\Phi(a.a^*) = \Phi(a^*.a)$, ou encore $\Phi((a+b)^*) = \Phi((a^*b^*)^*)$.

Exercice. Se convaincre.

Définition 4. Si E est une expression rationnelle et $L = \Phi(E)$ le langage rationnel associé, on dit que L est dénoté par E (ou que E dénote L).

Remarque 4. Nous ferons dorénavant un abus, en confondant dans les écritures les langages rationnels avec les expression rationnelle qui les dénotent. Ainsi, par exemple, nous pourrions nous donner le langage "$L = a^*(a+b)^*$". Il faut toutefois bien garder à l'esprit qu'un langage rationnel n'est pas une expression rationnelle, et qu'en particulier, plusieurs expressions rationnelles bien distinctes peuvent décrire le même langage. Ainsi, par exemple, les expressions rationnelles $(a^*b^*)^*$ et $(a+b)^*$ sont distinctes, mais via leur assimilation à des langages rationnels, on écrira pourtant $(a^* + b^*)^* = (a+b)^*$.

IV Bilan

La classe des langages rationnels est une classe de langages relativement simple, mais suffisamment vaste pour contenir des langages (très) intéressants. Nous reviendrons sur les langages rationnels dans les chapitres suivants, en nous posant entre autres les questions suivantes :

- Existe-t-il des algorithmes permettant de reconnaître les mots d'un langage rationnel donné ? La réponse est oui, avec des algorithmes de complexité linéaire en la longueur des mots (c'est à dire optimaux). C'est le concept d'automate fini qui règlera cette question.
- Existe-t-il des langages qui ne soient pas rationnels ? La réponse est oui. Nous donnerons une condition nécessaire pour qu'un langage soit rationnel, et nous fabriquerons des langages qui, bien que fort simples, ne sont pas rationnels.

Chapitre 7

Automates

I Généralités

I.1 Notion d'automate fini

Définition 1. [AUTOMATE FINI NON DÉTERMINISTE]

Un automate fini (non déterministe) est un 5-uplet $\mathcal{A} =< Q, A, E, I, T >$ où

- Q est un ensemble fini : l'ensemble des états.
- A est un ensemble fini : l'alphabet de l'automate.
- I est une partie de Q : l'ensemble des états initiaux (ou de départ).
- T est une partie de Q : l'ensemble des états terminaux (ou finaux, ou accepteurs).
- $E \subset Q \times A \times Q$ est l'ensemble des transitions.

Lorsque (p, a, q) est une transition de l'automate $\mathcal{A}$, on note $p \xrightarrow{a} q$. La lettre$a$ est l'étiquette de la transition, l'état p est l'origine de la transition et l'état q est l'extrémité de la transition.

I.2 Représentation par un graphe étiqueté

On représente un automate fini par un graphe étiqueté. Les sommets du graphe sont les états de l'automate (les éléments de Q). Lorsque le triplet (q, x, q') appartient à E, on trace entre q et q' une arête étiquetée par x. On distingue sur le graphe, d'une façon ou d'une autre, les états de départ et les états accepteurs. Nous adopterons la convention suivante : lorsque q est un état de départ, on fait pointer une flèche de l'extérieur du graphe vers le sommet étiqueté par q. Lorsque q est un état accepteur, on fait partir une flèche de q vers l'extérieur du graphe.

I.3 Exemples

On a pour l'automate de la **figure 7.1** : $A = \{a, b\}$, $Q = \{1, 2\}$, $I = \{1\}$, $T = \{1, 2\}$.

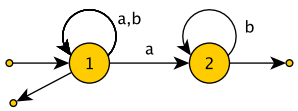


FIGURE 7.1 – Un automate fini

Pour l'automate de la **figure 7.2** , $A = \{a, b\}$, $Q = \{1, 2, 3, 4, 5\}$, $I = \{1, 4\}$, $T = \{3, 5\}$. Nous appellerons $\mathcal{A}_0$ cet automate et nous l'utiliserons dans la discussion qui va suivre.

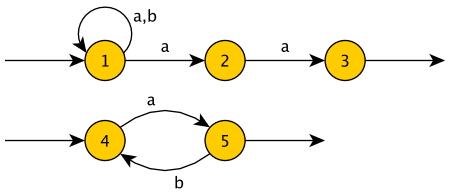


FIGURE 7.2 – L'automate $\mathcal{A}_0$

I.4 Calculs dans un automate

Définition 2. [CALCUL DANS UN AUTOMATE]

Soit $\mathcal{A} =< Q, A, E, I, T >$ un automate fini. Soient p et q deux états de $\mathcal{A}$. On appelle calcul dans $\mathcal{A}$ toute suite de transitions de la forme $q_i \overset{a_i}{\to} q_{i+1}$, $i = 0 \dots n-1$, l'extrémité de chaque transition étant l'origine de la suivante. On note plus simplement un tel calcul

$$c := q_0 \overset{a_0}{\to} q_1 \overset{a_1}{\to} q_2 \overset{a_2}{\to} \dots \overset{a_{n-1}}{\to} q_n$$

L'état q_0 est l'origine du calcul c, et l'état q_n est son extrémité.

On convient également que le calcul vide ε est un calcul d'origine q et d'extrémité q, ceci pour tout état q de l'automate.

Exemple. Dans l'automate de la figure 7.2, $c := ((1, b, 1), (1, a, 2), (2, a, 3))$ est un calcul dont l'origine est l'état 1 et l'extrémité est l'état 3.

Définition 3. [ÉTIQUETTE D'UN CALCUL]

Soit c un calcul dans l'automate $\mathcal{A}$. Avec les notations de la définition précédente, on appelle étiquette de c le mot de A^*

$$a_0 a_1 ... a_{n-1}$$

On note alors

$$c := q_0 \overset{a_0 \dots a_{n-1}}{\longrightarrow} q_n$$

L'étiquette de c est notée

$$|c| = a_0 \dots a_{n-1}$$

Exemple. Dans l'automate de la figure 7.2, l'étiquette du calcul $c := 1 \overset{b}{\to} 1 \overset{a}{\to} 2 \overset{a}{\to} 3$ est le mot $|c| = baa$.

Définition 4. [CALCUL RÉUSSI]

On dit qu'un calcul dans l'automate $\mathcal{A}$ est réussi lorsque son origine est un état initial de $\mathcal{A}$,

et son extrémité est un état final.

Définition 5. [LANGAGE RECONNU]

On dit qu'un mot $u \in A^*$ est reconnu (ou accepté) par l'automate $\mathcal{A}$ lorsqu'il est l'étiquette d'un calcul réussi dans $\mathcal{A}$.

On appelle langage reconnu par l'automate $\mathcal{A}$ l'ensemble des mots reconnus par cet automate : $\mathcal{L}(\mathcal{A}) = \{u \in A^*, \exists p \in I, \exists q \in T, p \xrightarrow{u} q\}$.

On dira qu'un langage est reconnaissable lorsqu'il existe un automate fini qui le reconnaît. On notera $Rec(A^*)$ l'ensemble des langages sur l'alphabet A reconnaissables.

Exemple. L'automate de la figure 7.2 reconnaît les mots aa, baa, bbaa, ..., a, aba, ... et plus généralement les mots du langage $(a+b)^*aa + a(ba)^*$.

Exercice.

1. On appelle identificateur tout mot sur l'alphabet ASCII commençant par une lettre et finissant par une suite de chiffres ou de lettres. Écrire un automate reconnaissant les identificateurs.
2. Soit l'alphabet $A = \{0, 1\}$. Écrire un automate reconnaissant le langage des mots binaires qui sont l'écriture en base 2 d'un multiple de 3.
3. On se place sur l'alphabet $A = \{+, -, 0, 1, ..., 9\}$. Écrire un automate reconnaissant les entiers relatifs écrits en base 10.
4. Soit $A = \{a, b, c\}$. Écrire un automate qui reconnaît le langage des mots dont l'éventuel premier a est avant l'éventuel premier b, lui-même avant l'éventuel premier c.

Exercice. Montrer qu'un automate reconnaît le mot vide si et seulement si il possède un état qui est à la fois initial et final.

Exercice. Créeer des automates reconnaissant le langage vide, le langage $\{\varepsilon\}$, le langage $\{a\}$ où a est une lettre.

II Propriétés de fermeture des langages reconnaissables

Cette section peut être sautée en première lecture. Le but de celle-ci est essentiellement de démontrer que $Rec(A^*)$ contient tous les langages rationnels. Nous montrerons d'une autre façon ce résultat dans le chapitre sur les langages locaux, au moyen de l'algorithme de Berry-Sethi.

II.1 Réunion de langages reconnaissables

Théorème 1. *$Rec(A^*)$ est stable par réunion finie.*

Démonstration. Il suffit bien sûr de montrer que la réunion de deux langages reconnaissables est encore reconnaissable.

Soient L_1 un langage reconnu par l'automate $\mathcal{A}_1 =< Q_1, A, E_1, I_1, T_1 >$ et L_2 un langage reconnu par l'automate $\mathcal{A}_2 =< Q_2, A, E_2, I_2, T_2 >$. Quitte à renommer les états, on peut supposer que $Q_1 \cap Q_2 = \emptyset$. Soit l'automate

$$\mathcal{A} =< Q_1 \cup Q_2, A, E_1 \cup E_2, I_1 \cup I_2, T_1 \cup T_2 >$$

Alors, $\mathcal{A}$ reconnaît le langage $L_1 \cup L_2$.

□

Nous venons de voir ici la première propriété de stabilité des langages reconnaissables par un automate fini. La suite du chapitre permettra de montrer que $Rec(A^*)$ est en fait stable par toutes les opérations classiques sur les langages.

II.2 Langages finis

Théorème 2. *Tout langage fini est reconnaissable.*

Démonstration. Soit $u = a_0 a_1 \dots a_{n-1} \in A^*$.L'automate de la figure 7.3 reconnaît le langage à un seul mot $L = \{u\}$. Un langage n'ayant qu'un seul mot est donc reconnaissable. Toute union finie de langages reconnaissables étant encore reconnaissable, un langage fini est donc reconnaissable. □

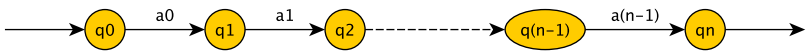


FIGURE 7.3 – Langage ayant un seul mot

Exercice. Traiter les cas oubliés dans la démonstration ci-dessus : le cas où le mot u est vide, et le cas où le langage fini est le langage $L = \emptyset$.

Avant de poursuivre, il nous faut introduire une sous-famille de la famille des automates : celle des automates standard.

II.3 Automates standard

Définition 6. Un automate fini $\mathcal{A} =< Q, A, E, I, T >$ est dit standard lorsque :
- I est un singleton (unique état initial).
- L'unique état initial de $\mathcal{A}$ n'est l'extrémité d'aucune transition de $\mathcal{A}$.

Théorème 3. *Si un langage est reconnaissable, il est reconnaisable par un automate standard.*

Démonstration. Soit L un langage, reconnu par l'automate $\mathcal{A} =< Q, A, E, I, T >$.

Soit $\mathcal{A}' =< Q \cup \{d\}, A, E', \{d\}, T' >$ où d est un état n'appartenant pas à Q, et

- $T' = T$ si $\varepsilon \notin L$ et $T' = T \cup \{d\}$ si $\varepsilon \in L$.
- $E' = E \cup \{(d, x, q'), \exists q \in I, (q, x, q') \in E\}$.

Alors, l'automate $\mathcal{A}'$ reconnaît L. □

Remarque 1. En gros, on peut dire que le standardisé d'un automate est obtenu en « dédoublant » les transitions issues de ses états initiaux.

Exemple. Considérons l'automate de la **figure 7.4** .

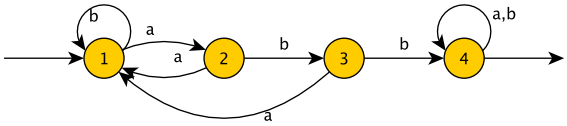


FIGURE 7.4 – Un automate non standard

Il est équivalent à l'automate standard de la **figure 7.5** .

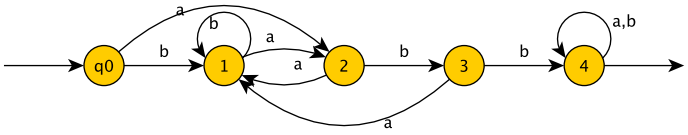


FIGURE 7.5 – Un automate standard équivalent au précédent

II.4 Stabilité des langages reconnaissables par concaténation

Théorème 4. *Soient L_1 et L_2 deux langages reconnaissables. Alors, le langage $L_1.L_2$ est reconnaissable.*

Démonstration. Soit $\mathcal{A}_i =< Q_i, A, E_i, I_i, T_i >$ $(i = 1, 2)$ un automate reconnaissant L_i, où $\mathcal{A}_1$ est un automate fini quelconque et $\mathcal{A}_2$ est un automate standard. On pose de plus $I_2 = \{j\}$. On définit

- $Q = (Q_1 \cup Q_2) \setminus \{d_2\}$.
- $I = I_1$.
- $T = T_2$ si $j \notin T_2$ et $T = T_1 \cup T_2 \setminus \{j\}$ si $j \in T_2$.
- $E = E_1 \cup E_2' \cup E_3$ où $E_2' = \{(q, x, q') \in E_2, q \neq j\}$ et $E_3 = \{(q, x, q'), q \in T_1, (j, x, q') \in E_2\}$.

Alors, l'automate $\mathcal{A} =< Q, A, E, I, T >$ reconnaît $L_1.L_2$.

□

Remarque 2. On peut très bien laisser l'état j lorsque l'on concatène les deux automates. Cela dit, aucun calcul issu d'un état de initial ne pourra y parvenir : cet état est inaccessible.

Nous nous contentons d'indiquer le procédé de fabrication de l'automate $\mathcal{A}$ par deux schémas. La démonstration rigoureuse est laissée en exercice.

Le cas où $j \notin T_2$ est résumé dans la **figure 7.6** . Lorsque $j \in T_2$, il faut conserver aux états finaux de $\mathcal{A}_1$ leur statut accepteur.

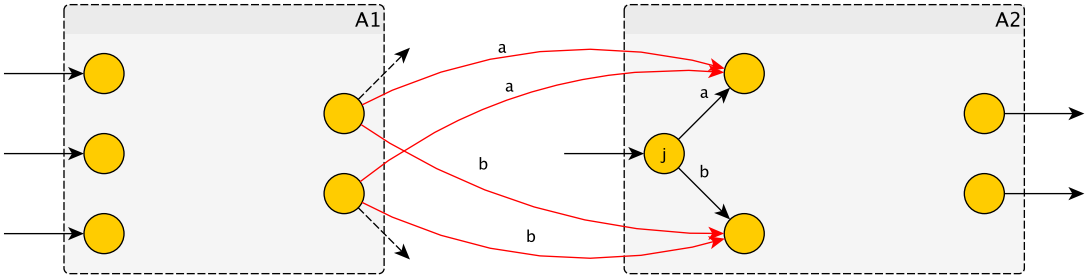


FIGURE 7.6 – Concaténation de deux automates

Exercice. Construire un automate $\mathcal{A}_1$ reconnaissant $L_1 = a^*$ et un automate standard $\mathcal{A}_2$ reconnaissant $L_2 = b^*$. En déduire un automate reconnaissant le langage $L = a^*b^*$.

II.5 Stabilité des langages reconnaissables par fermeture de Kleene (étoile)

Théorème 5. *Soit L un langage reconnaissable. Alors, L^* est reconnaissable.*

Démonstration. Nous nous contentons d'indiquer le principe de construction, très proche de ce qui a été vu lors de la concaténation. On se donne un automate standard $\mathcal{A} =< Q, A, E, I = \{d\}, T >$ qui reconnaît le langage L. On fabrique ensuite l'automate $\mathcal{A} =< Q \setminus \{d\}, A, E', I, T \cup I >$ où $E' = E \cup \{(q, x, q'), q \in T, (d, x, q') \in E\}$ (voir figure **??**). □

Prenons pour exemple le langage $L = aba$. Un automate reconnaissant L est donné par la **figure 7.7**(cet automate est déjà standard). Un automate reconnaissant L^*, fabriqué selon la construction du théorème, est donné à la **figure 7.8**.

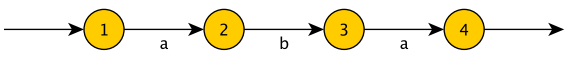


FIGURE 7.7 – Un automate reconnaissant *aba*

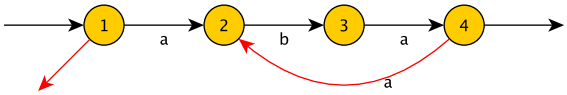


FIGURE 7.8 – L'étoile de l'automate précédent, reconnaît $(aba)^*$

III Automates déterministes

III.1 Inconvénient des automates « généraux »

Le principal inconvénient d'un automate fini quelconque est son non déterminisme : l'acceptation ou le refus d'un mot par l'automate ne peut pas être traitée de façon efficace. En effet, il faut effectuer tous les calculs possibles des états initiaux vers les états finals, et regarder si l'un de cescalculs a pour étiquette le mot qui nous intéresse. D'un point de vue algorithmique, cette situation est inacceptable, surtout lorsque l'on sait qu'en pratique, certains automates peuvent avoir des milliers, voire des millions d'états. On remédie à cette situation en introduisant les automates déterministes, pour lesquels il n'y a qu'une seule façon d'effectuer un calcul donné.

Définition 7. [AUTOMATE FINI DÉTERMINISTE]

Soit $\mathcal{A} =< Q, A, E, I, T >$ un automate fini. On dit que $\mathcal{A}$ est déterministe lorsque :

- Il y a un unique état initial.
- Pour tout état $q \in Q$ et toute lettre $x \in A$, il existe au plus un état $q' \in Q$ tel que $q \xrightarrow{x} q'$.

E peut donc être vu comme le graphe d'une fonction $\delta : Q \times A \to Q$, que l'on appelle la fonction de transition de l'automate.

Si, mieux que cela, δ est une application, alors l'automate est qualifié d'automate fini déterministe complet (AFDC).

III.2 Équivalence entre AFD et AFDC

Théorème 6. *Tout AFD est équivalent à un AFDC.*

En clair, étant donné un automate déterministe $\mathcal{A}$, il existe un automate déterministe complet $\mathcal{A}'$ qui reconnaît le même langage que $\mathcal{A}$.

Démonstration. L'idée est simple. On rajoute à l'automate $\mathcal{A}$ un état ω vers lequel on fait pointer toutes les transitions manquantes. On rajoute également les transitions (ω, x, ω) pour toutes les lettres x de l'alphabet (**figure 7.9**). L'état ω est un état-puits : une fois que l'on y est entré, on ne peut plus en ressortir. Dans le cas présenté ici, l'état-puits n'est pas un état final. □

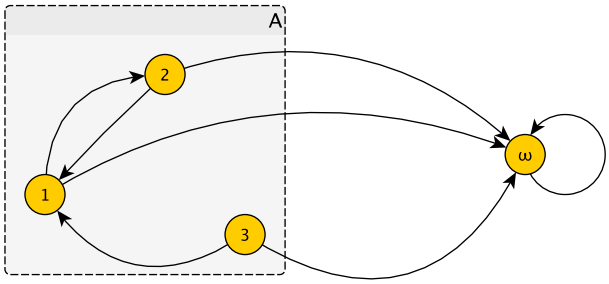


FIGURE 7.9 – Complétion d'un automate déterministe

Exemple. L'automate de la **figure 7.10**, construit sur l'alphabet $\{a, b\}$, est un AFD qui reconnaît le langage $L = (a^2) * ab$. On le complète en l'automate de la **figure 7.11**.

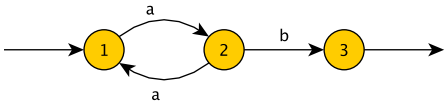


FIGURE 7.10 – Un automate déterministe

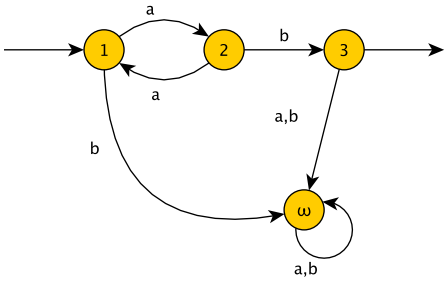


FIGURE 7.11 – Complétion de l'automate précédent

Si, en revanche, l'état 3 était un état accepteur, alors l'automate reconnaîtrait le langage $L = (a^*baA^*) \cup (a^*b^2a^*bA^*)$

III.3 Prolongement de la fonction de transition

La fonction de transition δ est a priori définie sur une $Q \times A$. On peut la prolonger à $Q \times A^*$ de la façon suivante. On pose

- $\forall q \in Q, \delta(q, \varepsilon) = q$.
- $\forall q \in Q, \forall u \in A^*, \forall x \in A, \delta(q, xu) = \delta(\delta(q, x), u)$.

Il est alors facile de vérifier que

$$\forall q \in Q, \forall u, v \in A^*, \delta(q, uv) = \delta(\delta(q, u), v)$$

Exercice. Prouver la propriété ci-dessus. On pourra faire une récurrence sur la longueur des mots.

Théorème 7. *Soit $\mathcal{A}$ un automate déterministe d'état de départ d, et de fonction de transition δ. L'automate $\mathcal{A}$ reconnaît le mot u si et seulement si $\delta(d, u)$ est un état final de $\mathcal{A}$.*

Démonstration. Il suffit de faire une récurrence sur la longueur de u. □

III.4 Déterminisation d'un automate fini

Nous allons montrer que tout automate est équivalent à un automate déterministe, et que cette équivalence est effective : il existe un algorithme permettant de calculer l'automate « déterminisé » à partir de l'automate donné au départ.

Théorème 8. *Soit L un langage. Alors, L est reconnaissable si et seulement si il est reconnaissable par un automate déterministe.*

Démonstration. Idée de la démonstration : Soit L le langage reconnu par l'automate (non déterministe a priori) $\mathcal{A} =< Q, A, E, I, T >$. On pose

- $Q' = \mathcal{P}(Q)$ (l'ensemble des parties de Q).
- $I' = \{I\}$.
- $T' = \{P \in Q', P \cap T \neq \emptyset\}$.
- $\delta' : Q' \times A \to Q'; (P, x) \mapsto \{q' \in Q, \exists q \in P, (q, x, q') \in E\}$. On note E' le graphe de l'application δ'.

Nous allons, après examen d'un exemple, que l'automate déterministe $\mathcal{A}' =< Q'A, E', I', T' >$ reconnaît le langage L.

□

Exemple. On prend $A = \{a, b\}$. On considère l'automate $\mathcal{A}$ de la **figure 7.12**. Cet automate reconnaît le langage $L = A^*abaA^*$.

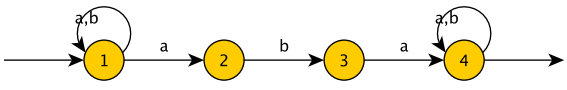


FIGURE 7.12 – Un automate qui reconnaît $(a+b)^*aba(a+b)^*$

L'automate $\mathcal{A}'$ est donné à la **figure 7.13**.

Exercice. Écrire un automate reconnaissant $L = (a+b)^*(babb)(a+b)^*$. Le déterminiser.

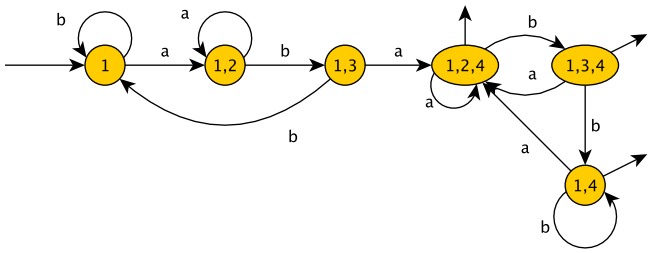


FIGURE 7.13 – Le déterminisé de l'automate de la figure 7.12

Démonstration. Démonstration du théorème : Soit $u \in A^*$. On cherche une condition portant sur $\mathcal{A}'$ pour que u soit reconnu par $\mathcal{A}$.

Si $u = \varepsilon$, alors $\mathcal{A}$ reconnaît u si et seulement si $I \cap T \neq \emptyset$ ou encore si et seulement si $I \in \mathcal{T}'$, c'est à dire $\{I\} \cap \mathcal{T}' \neq \emptyset$. Mais cela signifie que $\mathcal{A}'$ reconnaît u.

Si $u = x_0 x_1 \dots x_{n-1} \neq \varepsilon$, alors :

1. $\mathcal{A}'$ reconnaît u si et seulement si $\delta'(I, u) \in T'$, c'est à dire s'il existe $P_0, P_1, ..., P_n \in Q'$ vérifiant $P_0 = I$, $P_n \in T'$, et $\delta'(P_0, x_0) = P_1, \delta'(P_1, x_1) = P_2, ..., \delta'(P_{n-1}, x_{n-1}) = P_n$.
2. $\mathcal{A}$ reconnaît u si et seulement si il existe $q_0, ..., q_n \in Q$, il existe $q_0 \in I$, $q_n \in T$ tels que $\forall i \in \{0, ..., n-1\}, (q_i, x_i, q_{i+1}) \in E$.

Il faut voir que ces deux conditions sont équivalentes.

- $1 \Rightarrow 2$: On suppose que $\mathcal{A}$ reconnaît le mot u. Avec les notations ci-dessus, on pose $P_0 = I$. On a alors $q_0 \in P_0$. Donc, $q_1 \in P_1 = \delta'(P_0, x_0)$. On obtient par récurrence que pour $k = 1..n-1$, $q_k \in P_k = \delta'(P_{k-1}, x_{k-1})$. De là, $q_n \in P_n$, donc P_n rencontre T et finalement $P_n \in T'$. Ainsi, $\mathcal{A}'$ reconnaît le mot u.
- $2 \Rightarrow 1$: On suppose que $\mathcal{A}'$ reconnaît le mot u. On a $P_n \in T'$, donc $P_n \cap T \neq \emptyset$. Soit $q_n \in P_n \cap T$. On a $\delta'(P_{n-1}, x_{n-1}) = P_n$, donc il existe $q_{n-1} \in P_{n-1}$ tel que $(q_{n-1}, x_{n-1}, q_n) \in \delta$. On trouve ainsi par récurrence $q_0 \in P_0$, $q_1 \in P_1$, ... $q_n \in P_n \cap T$ tels que pour tout i, $(q_i, x_i, q_{i+1}) \in E$. On en déduit que $\mathcal{A}$ reconnaît u.

Finalement, les automates $\mathcal{A}$ et $\mathcal{A}'$ reconnaissent les mêmes mots, et sont donc équivalents.

□

Remarque 3. Le déterminisé d'un automate fini possédant n états est donc un automate fini déterministe et complet possédant 2^n états. On a là une explosion combinatoire qui peut conduire à des situations très désagréables (par exemple, un automate à 100 états se déterminisant en un auomate à 1267650600228229401496703205376 états). Dans la pratique, on se contente des états qui sont accessibles à partir de l'état initial : on fabrique le déterminisé de proche en proche à partir de l'état I, en rajoutant des transitions et de nouveaux états jusqu'à ce que l'on ne retombe que sur des états déjà vus. Cette méthode est appelée *déterminisation par la méthode des sous-ensembles.*

Il faut toutefois savoir que, bien que dans les cas rencontrés en pratique le déterminisé est « petit », il existe des exemples d'automates à n états dont le déterminisé par la méthode des sous-ensembles

possède 2^n états. Il existe même des automates à n états tels que tout automate déterministe équivalent possède au moins 2^n états.

III.5 Stabilité des langages reconnaissables par passage au complémentaire

Théorème 9. *Soit L un langage reconnaissable. Alors, le langage $A^* \setminus L$ est reconnaissable.*

Démonstration. On peut supposer que le langage L est reconnu par un AFDC $\mathcal{A} =< Q, A, E, I = \{d\}, T >$. On considère $\mathcal{A}' =< Q, A, E, I, Q \setminus T >$, c'est à dire que les états finals de $\mathcal{A}$ deviennent non finals dans $\mathcal{A}'$ et vice versa. Alors, l'automate $\mathcal{A}'$ reconnaît le langage $A^* \setminus L$. En effet, soit $u \in A^*$. Alors $\mathcal{A}$ ne reconnaît pas u si et seulement si $\delta(d, u) \notin T$, c'est à dire si et seulement si $\mathcal{A}'$ reconnaît u. □

III.6 Stabilité des langages reconnaissables par intersection et différence

Nous admettons ici la propriété vue à la section III : la réunion de deux langages reconnaissable est reconnaissable.

Si L_1 et L_2 sont deux langages reconnaissables, alors $L_1 \cap L_2 = A^* \setminus ((A^* \setminus L_1) \cup (A^* \setminus L_2))$. Ainsi, $Rec(A^*)$ est aussi stable par intersection. Enfin, $L_1 \setminus L_2 = L_1 \cap (A^* \setminus L_2))$ et $Rec(A^*)$ est donc stable par différence.

IV Produit d'automates

Nous allons définir une nouvelle opération sur les automates finis : leur produit cartésien. Pour simplifier nous ne considérerons que des AFDC. Un certain nombre de résultats (mais pas tous) restent valables même si les automates ne sont pas complets, voire s'ils sont non déterministes.

IV.1 C'est quoi ?

Définition 8. Soient $\mathcal{A}' =< Q', A, \delta', i', T' >$ et $\mathcal{A}'' =< Q'', A, \delta'', i'', T'' >$ deux AFDC sur un même alphabet A. Le produit des automates $\mathcal{A}'$ et $\mathcal{A}''$ est l'AFDC $\mathcal{A} =< Q, A, \delta, i, T >$ où

- $Q = Q' \times Q''$
- $\forall q' \in Q', \forall q'' \in Q'', \forall a \in A, \delta((q', q''), a) = (\delta'(q', a), \delta''(q'', a))$.
- $i = (i', i'')$

L'ensemble T des états finaux de $\mathcal{A}$ sera fixé ultérieurement de plusieurs façons différentes.

IV.2 Prolongement de la fonction de transition

Proposition 10. *Avec les notations de la définition, on a*

$$\forall q' \in Q', \forall q'' \in Q'', \forall u \in A^*, \delta((q', q''), u) = (\delta'(q', u), \delta''(q'', u))$$

Démonstration. Par récurrence sur la longueur du mot u. □

IV.3 Intersection de deux langages rationnels

Prenons $T = T' \times T''$. Soit $u \in A^*$. Le mot u est reconnu par $\mathcal{A}$ si et seulement si $\delta(i, u) \in T$, ou encore $\delta'(i', u) \in T'$ et $\delta''(i'', u) \in T''$, c'est à dire u est reconnu par $\mathcal{A}'$ et $\mathcal{A}''$. En conclusion,

$$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}') \cap \mathcal{L}(\mathcal{A}'')$$

Et en corollaire :

Proposition 11. *L'intersection de deux langages rationnels est un langage rationnel.*

IV.4 Réunion de deux langages rationnels

Prenons $T = (T' \times Q'') \cup (Q' \times T'')$. Soit $u \in A^*$. Le mot u est reconnu par $\mathcal{A}$ si et seulement si $\delta(i, u) \in T$, ou encore $\delta'(i', u) \in T'$ ou $\delta''(i'', u) \in T''$, c'est à dire u est reconnu par $\mathcal{A}'$ ou par $\mathcal{A}''$. En conclusion,

$$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}') \cup \mathcal{L}(\mathcal{A}'')$$

Et en corollaire :

Proposition 12. *La réunion de deux langages rationnels est un langage rationnel.*

Exemple. On a représenté sur la figure 7.14 :

- Sur la première ligne, un AFDC qui reconnaît les mots commençant par un a.
- Sur la première colonne, un AFDC qui reconnaît les mots finissant par un b.
- le produit de ces deux automates, avec un ensemble T d'états finaux lui faisant reconnaitre le langage des mots commençant par a ou finissant par b : $a(a+b)^* + (a+b)^*b$.

IV.5 Diférence de deux langages rationnels

Prenons $T = (T' \times (Q'' \setminus T''))$. Soit $u \in A^*$. Le mot u est reconnu par $\mathcal{A}$ si et seulement si $\delta(i, u) \in T$, ou encore $\delta'(i', u) \in T'$ et $\delta''(i'', u) \notin T''$, c'est à dire u est reconnu par $\mathcal{A}'$ et pas par $\mathcal{A}''$. En conclusion,

$$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}') \setminus \mathcal{L}(\mathcal{A}'')$$

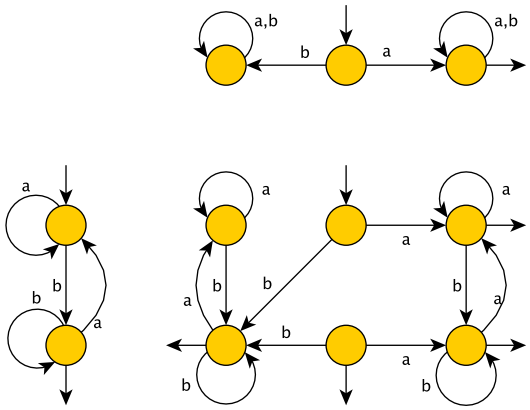


FIGURE 7.14 – Un produit d'automates

Et en corollaire :

Proposition 13. *La différence de deux langages rationnels est un langage rationnel.*

Exercice. Que peut-on prendre pour T afin que le langage reconnu par $\mathcal{A}$ soit $\mathcal{L}(\mathcal{A}')\Delta\mathcal{L}(\mathcal{A}'')$, où Δ est l'opérateur de différence symétrique ?

V Résumé

Nous avons mis en évidence dans ce chapitre 4 types d'automates finis :

- Les automates non déterministes qui sont le type le plus général d'automate vu dans ce cours.
- Les automates standard.
- Les automates déterministes.
- Les automates déterministes complets.

Tous ces types d'automates reconnaissent les mêmes langages, ce qui a été à chaque fois prouvé par des théorèmes d'équivalence. Il est fondamental de constater que toutes les démonstrations données sont constructives, et permettent de fabriquer effectivement les automates dont on parle. On ne saurait trop insister sur le fait que, dans un cours d'informatique, une démonstration se doit d'être constructive !

Enfin, l'ensemble $Rec(A^*)$ s'avère avoir de remarquables propriétés de stabilité. Pour commencer :
— $Rec(A^*)$ contient les langages finis.
— $Rec(A^*)$ est stable par réunion, concaténation et étoile.

Le plus petit ensemble de langages vérifiant ces propriétés étant l'ensemble $Rat(A^*)$ des langages rationnels sur l'alphabet A, on en déduit :

$$Rat(A^*) \subset Rec(A^*)$$

Nous verrons au chapitre suivant que l'inclusion réciproque est également vraie.

Enfin, $Rec(A^*)$ est également stable par intersection, complémentation et différence, et bien d'autres opérations importantes.

Chapitre 8

Langages locaux

I Notion de langage local

I.1 C'est quoi ?

Soit L un langage sur l'alphabet A. Notons $P = P(L)$ l'ensemble des premières lettres des mots de L, $S = S(L)$ l'ensemble des dernières lettres des mots de L, $F = F(L)$ l'ensemble des facteurs de longueur 2 des mots de L et $N = N(L) = A^2 \setminus F(L)$. Plus précisément,

- $P(L) = \{x \in A, xA^* \cap L \neq \emptyset\}$.
- $S(L) = \{x \in A, A^*x \cap L \neq \emptyset\}$.
- $F(L) = \{u \in A^2, A^*uA^* \cap L \neq \emptyset\}$.
- $N(L) = A^2 \setminus F(L)$.

On a l'inclusion $L \setminus \{\varepsilon\} \subseteq (P(L)A^* \cap A^*S(L)) \setminus A^*N(L)A^*$. L'autre inclusion n'est pas toujours vraie. Les langages pour lesquels on a l'égalité sont appelés *langages locaux*.

Définition 1. Soit L un langage sur l'alphabet A. On dit que L est local lorsqu'il existe deux ensembles finis de lettres $P \subseteq A$ et $S \subseteq A$ et un ensemble fini de mots de 2 lettres $N \subseteq A^2$ tels que

$$L \setminus \{\varepsilon\} = (PA^* \cap A^*S) \setminus A^*NA^*$$

- Les lettres de P sont les premières lettres des mots de $L \setminus \{\varepsilon\}$.
- Les lettres de S sont les dernières lettres des mots de $L \setminus \{\varepsilon\}$.
- Les mots de N sont les facteurs de longueur 2 interdits des mots de $L \setminus \{\varepsilon\}$.

Remarque 1. Un langage local est donc presque caractérisé par le triplet (P, S, N). On peut également le caractériser par le triplet (P, S, F) où $F = A^2 \setminus N$ est l'ensemble des facteurs de longueur 2 autorisés des mots de L. Pour caractériser complètement le langage L, il faut également préciser si le mot vide est ou n'est pas dans L. Nous serons assez souples sur ce détail dans la suite. Le lecteur est invité à combler cette lacune.

Exemple.

- $L = (ab)^*$ (ou $(ab)^+$). L est local avec $P = \{a\}$, $S = \{b\}$ et $N = \{aa, bb\}$ (ou $F = \{ab, ba\}$).
- $P = \{a\}$, $S = \{b\}$ et $N = \{ab, ba\}$. ALors $L = \emptyset$ ou ε.
- $L = (abc)^*$. L est local avec $P = \{a\}$, $S = \{c\}$ et $F = \{ab, bc, ca\}$. L'ensemble N contient 6 éléments : $N = \{aa, ac, ba, bb, cb, cc\}$.

I.2 Langages locaux et langages rationnels

Proposition 1. *Tout langage local est rationnel. La réciproque est fausse.*

Démonstration. Avec les notations de la définition, les langages P, N et S sont finis donc rationnels. Les langages PA^*, A^*S et A^*NA^* sont rationnels, et l'intersection et la différence de deux langages rationnels sont encore rationnels. Donc L l'est aussi.

Pour la réciproque, soient $L_1 = ab$ et $L_2 = a^*$. Le langage $L = L_1 \cup L_2$ est rationnel mais pas local. Supposons le contraire. On a alors $P = \{a\}$, $S = \{a, b\}$ et $F = \{aa, ab\}$. Mais alors $aab \in L$, ce qui n'est pas. □

I.3 Calcul des ensembles P, S et F

Soit L un langage local sur l'alphabet A, dénoté par une expression rationnelle e. On se propose de déterminer les ensembles $P(L)$, $S(L)$ et $F(L)$. Notons $\Lambda(L) = \{\varepsilon\} \cap L$. L'ensemble $\Lambda(L)$ est vide si le mot vide n'appartient pas à L et est égal au singleton $\{\varepsilon\}$ sinon. On a, pour tout $a \in A$ et toutes expressions rationnelles e et e' :

- $\Lambda(\emptyset) = \emptyset$.
- $\Lambda(\varepsilon) = \varepsilon$.
- $\Lambda(a) = \emptyset$.
- $\Lambda(e + e') = \Lambda(e) \cup \Lambda(e')$.
- $\Lambda(e.e') = \Lambda(e) \cap \Lambda(e')$.
- $\Lambda(e^*) = \varepsilon$.

De là :

- $P(\emptyset) = \emptyset, S(\emptyset) = \emptyset$.
- $P(\varepsilon) = \emptyset, S(\varepsilon) = \emptyset$.
- $P(a) = \{a\}, S(a) = \{a\}$.
- $P(e + e') = P(e) \cup P(e'), S(e + e') = S(e) \cup S(e')$.
- $P(e.e') = P(e) \cup \Lambda(e)P(e'), S(e.e') = S(e') \cup S(e)\Lambda(e')$.
- $P(e^*) = P(e), S(e^*) = S(e)$.

Ce n'est pas plus compliqué pour les facteurs de longueur 2 :

- $F(\emptyset) = \emptyset$.
- $F(\varepsilon) = \emptyset$.
- $F(a) = \emptyset$.
- $F(e + e') = F(e) \cup F(e')$.
- $F(e.e') = F(e) \cup F(e') \cup S(e)P(e')$.
- $F(e^*) = F(e) \cup S(e)P(e)$.

On dispose donc d'un algorithme permettant le calcul de $\Lambda(L)$, $P(L)$, $S(L)$ et $F(L)$.

Exercice. Appliquer l'algorithme ci-dessus au langage $L = a(a + b)^*$.

II Propriétés de stabilité

II.1 Intersection de langages locaux

Proposition 2. *L'intersection de deux langages locaux est un langage local.*

Démonstration. Soient L' et L'' deux langages locaux caractérisés par les triplets (P', S', F') et (P'', S'', F''). Soit L le langage local caractérisé par (P, S, F) où $P = P' \cap P''$, $S = S' \cap S''$ et $F = F' \cap F''$. Montrons que $L = L' \cap L''$.

- Soit $u \in L$. La première lettre de u est une lettre de P, donc de P'. De même la dernière lettre de u est dans S' et les facteurs de longueur 2 de u sont dans F'. Donc $u \in L'$. De même, $u \in L''$.
- L'inclusion réciproque se traite de façon identique.

□

II.2 Étoile d'un langage local

Proposition 3. *L'étoile d'un langage local est un langage local.*

Démonstration. Soit L le langage local sur A caractérisé par (P, S, F). Soit L' le langage local caractérisé par $(P, S, F \cup (S \times P))$.

- Soit $u \in L'$. Montrons par récurrence forte sur $|u|$ que $u \in L^*$. C'est clair si u est vide. Supposons la propriété vraie pour tous les mots de L' de longueur strictement inférieure à n. Soit $u \in L'$ de longueur n. u commence par une lettre de P et finit par une lettre de S. Si tous les facteurs de longueur 2 de u sont dans F, alors u est dans L donc dans L^*. Supposons que u possède un facteur v de longueur 2, $v \notin F$. Alors $v \in S \times P$. On peut donc écrire $u = xspy$ où $x, y \in A^*$, $s \in S$, $p \in P$. Par l'hypothèse d'induction, xs et py sont dans L^*, donc $u \in L^*$. D'où $L' \subseteq L^*$.
- L'autre inclusion est facile.

□

II.3 Union et produit de langages locaux

L'union de deux langages locaux n'est en général pas locale. Prenons l'exemple vu plus haut : $L_1 = \{ab\}$ et $L_2 = \{a\}^*$ sont bien locaux. Supposons en revanche que $L_1 \cup L_2$ est local, associé au triplet (P, S, F). ALors $P = \{a\}$, $S = \{a, b\}$ et $F = \{aa, ab\}$. Donc, le mot aab est dans $L_1 \cup L_2$, contradiction.

Exercice. Montrer que $L_1 L_2$ n'est pas non plus local.

On a cependant le résultat suivant :

Proposition 4. *Soient L_1 et L_2 deux langages locaux sur des alphabets disjoints. Les langages $L_1 L_2$ et $L_1 \cup L_2$ sont locaux.*

Démonstration. Le langage L_i $(i = 1, 2)$ est caractérisé par le triplet (P_i, S_i, F_i). Alors $L_1 \cup L_2$ est le langage local caractérisé par $(P_1 \cup P_2, S_1 \cup S_2, F_1 \cup F_2)$ et $L_1 L_2$ est le langage local caractérisé par $(P_1, S_2, F_1 \cup F_2)$ □

III L'algorithme de Berry-Sethi

III.1 Automates locaux

Définition 2. Soit $\mathcal{A} =< Q, A, \delta, i, T >$ un automate déterministe. On dit que l'automate $\mathcal{A}$ est local lorsque, pour toute lettre $a \in A$, toutes les transitions étiquetées par a (s'il y en a) arrivent sur un même état. Ou encore : l'ensemble $\{\delta(q, a), q \in Q\}$ est l'ensemble vide ou un singleton.

Exemple. L'automate ci-dessous est un automate local :

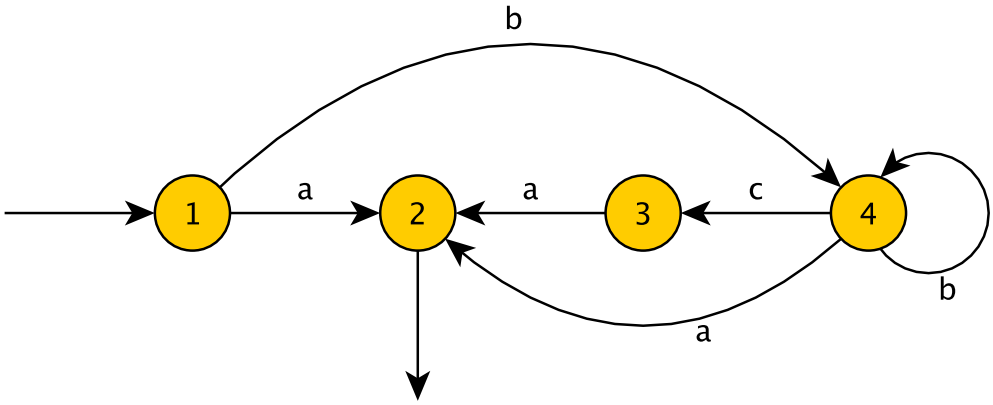


FIGURE 8.1 – Un automate local

III.2 Langages locaux et automates locaux

Proposition 5. *Tout langage local est reconnu par un automate local. Plus précisément : soit L un langage local sur l'alphabet A. Alors L est reconnu par l'automate local $\mathcal{A} =< Q, A, \delta, \alpha, T >$, où $\alpha \notin A$ et :*

1. *$Q = A \cup \{\alpha\}$*
2. *$T = S(L)$ si $\varepsilon \notin L$ et $S(L) \cup \{\alpha\}$ sinon.*
3. *Pour tous $a, b \in A$, si $ab \in F(L)$, alors $\delta(a, b) = b$. Sinon $\delta(a, b)$ n'est pas défini.*
4. *Pour tout $b \in A$, $\delta(\alpha, b) = b$ si $b \in P(L)$. Sinon, $\delta(\alpha, b)$ n'est pas défini.*

Démonstration. Soit $u = a_0 a_1 \dots a_{n-1}$ un mot non vide (le cas du mot vide est évident). Le mot u est reconnu par $\mathcal{A}$ si et seulement si u est l'étiquette du (déterminisme !) calcul réussi

$\alpha \xrightarrow{a_0} a_0 \xrightarrow{a_1} a_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-2}} a_{n-2} \xrightarrow{a_{n-1}} a_{n-1}$. Ceci équivaut à dire que

- a_{n-1} est un état accepteur, c'est à dire que $a_{n-1} \in S(L)$.
- On a une transition $\alpha \xrightarrow{a_0} a_0$, c'est à dire que $a_0 \in P(L)$.
- Pour tout $j \in [0, n-2]$ on a une transition $a_j \xrightarrow{a_{j+1}} a_{j+1}$, c'est à dire $a_j a_{j+1} \in F(L)$.

Bref, u est reconnu par $\mathcal{A}$ si et seulement si $u \in L$. □

Remarque 2. Plus précisément, L est reconnu par un automate local *standard* c'est à dire un automate local pour lequel l'état initial n'est l'extrémité d'aucune transition.

Exemple. Soit $L = (abc)^*$. On a $P(L) = \{a\}$, $S(L) = \{c\}$ et $F(L) = \{ab, bc, ca\}$. On en déduit (figure 8.2) un automate local reconnaissant L :

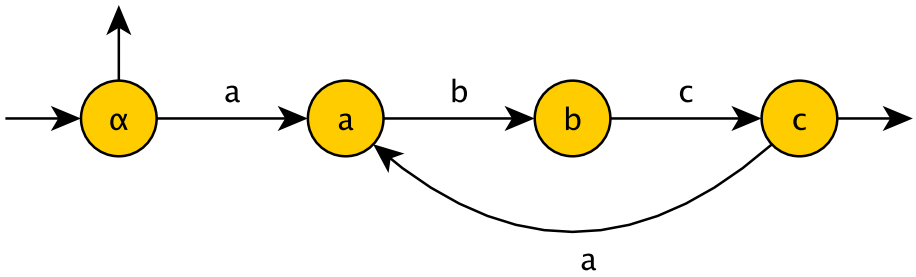


FIGURE 8.2 – Un automate local qui reconnaît $(abc)^*$

Proposition 6. *Si un langage est reconnu par un automate local standard, alors c'est un langage local.*

Démonstration. Nous donnons juste l'idée. Soit $\mathcal{A} =< Q, A, \delta, \alpha, T >$ un automate local standard. Soient

- P l'ensemble des étiquettes des transitions d'origine α.
- S l'ensemble des étiquettes des transitions d'extrémité dans T.
- F l'ensemble des mots de longueur 2 apparaissant dans un calcul réussi dans l'automate $\mathcal{A}$.

Il reste à montrer que le langage reconnu par $\mathcal{A}$ est le langage local caractérisé par (P, S, F). □

III.3 Expressions rationnelles linéaires

Définition 3. Soit e une expression rationnelle sur un alphabet A. On dit que e est linéaire lorsque chaque lettre de A apparaît au plus une fois dans l'expression e.

Proposition 7. *Le langage dénoté par une expression rationnelle linéaire est local.*

Démonstration. Soit e une expression rationnelle sur l'alphabet A. On raisonne par induction structurelle.

- C'est évident si $e = \emptyset$ ou ε ou $e \in A$.
- Soient e_1 et e_2 deux expressions linéaires qui n'ont aucune lettre de A en commun. Supposons que les langages L_1 et L_2 dénotés par e_1 et e_2 sont locaux. Ces deux langages sont des langages sur des alphabets distincts, donc leur produit et leur union sonr des langages locaux.
- L'étoile d'un langage local étant un langage local, la démonstration est terminée.

Remarque 3. Le résultat reste vrai pour des expressions rationnelles généralisées ($\wedge$, $\backslash$). □

III.4 L'algorithme de Berry-Sethi

L'algorithme de Berry-Sethi, appelé aussi algorithme de Glushkov, permet de déterminer, à partir d'une expression rationnelle e, un automate reconnaissant le langage dénoté par e. L'algorithme est le suivant :

1. Linéariser e en une expression rationnelle linéaire e'. Plus précisément, numéroter les diverses occurences des lettres de e afin d'obtenir une expression rationnelle linéaire. Par exemple, $e = (a + b)^*aba$ se linéarise en $e' = (a_1 + b_1)^*a_2b_2a_3$.
2. Déterminer l'automate local $\mathcal{A}'$ reconnaissant le langage $L(e')$ dénoté par e'.
3. Supprimer les indices des étiquettes des transitions de $\mathcal{A}'$ pour obtenir un automate (non déterministe) $\mathcal{A}$ qui reconnaît donc le langage $L(e)$.

L'automate $\mathcal{A}$ peut ensuite, bien entendu, être déterminisé par la méthode des sous-ensembles.

III.5 Un exemple

Prenons par exemple $e = a(a + b)^*$. L'expression rationnelle linéarisée est $e' = a_1(a_2 + b_1)^*$. On a facilement $\Lambda(e') = \emptyset$, $P(e') = \{a_1\}$, $S(e') = \{a_1, a_2, b_1\}$

et $F(e') = \{a_1a_2, a_1b_1, a_2a_2, a_2b_1, b_1a_2, b_1b_1\}$. On en déduit l'automate local reconnaissant $L(e')$:

et un automate (déterministe, par chance) $\mathcal{A}$ reconnaissant $L(e)$.

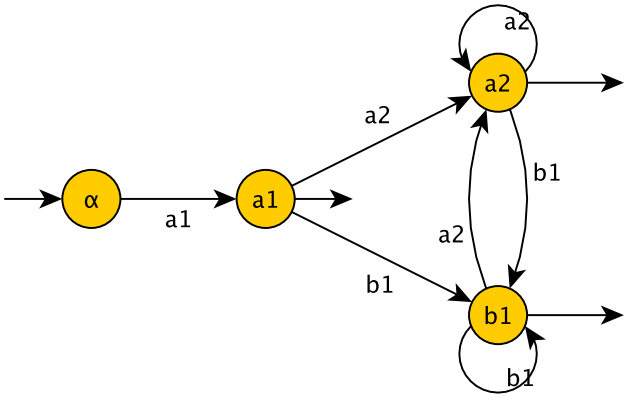


FIGURE 8.3 – L'automate local reconnaissant $L(e')$

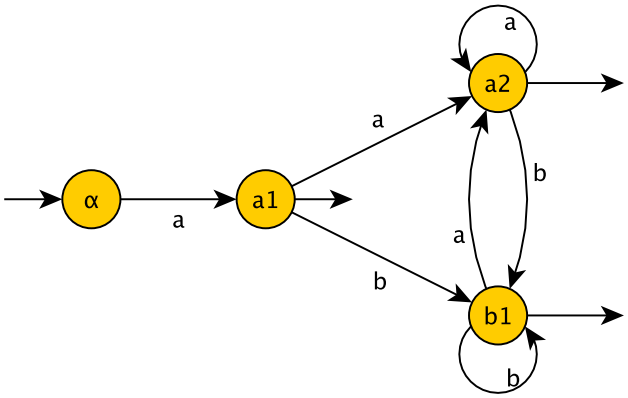


FIGURE 8.4 – Un automate reconnaissant $L(e)$

Chapitre 9

Compléments sur les langages rationnels

I Langages rationnels et automates finis

I.1 Le théorème de Kleene

Théorème 1. [Lemme d'Arden]

Soient K, L deux langages sur un alphabet A. Soit l'équation

$$(\mathcal{E}) X = KX + L$$

dont l'inconnue X est un langage sur A.

- Si $\varepsilon \notin K$, l'équation $(\mathcal{E})$ admet pour unique solution $X = K^*L$.
- Si $\varepsilon \in K$, X est solution de $(\mathcal{E})$ si et seulement si $X = K^*M$, avec $L \subset M$.

Démonstration. Soit X une solution de l'équation. On a

$$X = KX + L = K(KX + L) + L = K^2X + KL + L$$

En reportant $X = KX + L$, on obtient

$$X = K^3X + (K^2 + K + \varepsilon)L$$

Plus généralement, on montre par récurrence sur n que pour tout entier $n \in \mathbb{N}$,

$$X = K^{n+1}X + \left(\sum_{j=0}^{n} K^j\right) L$$

On voit donc que, pour tout entier n, $K^nL \subset X$ (ne pas oublier que la somme est en fait une réunion). Donc : $K^*L \subset X$.

- Supposons que $\varepsilon \notin K$. Tout mot de K est donc de longueur supérieure ou égale à 1, et donc tout mot de K^n est de longueur supérieure ou égale à n. Soit $u \in X$, $|u| = n$. Alors,

  $$u \notin K^{n+1}X$$

  donc

  $$u \in \sum_{j=0}^{n} K^jL$$

  D'où $u \in K^*L$.
  Inversement,

  $$K(K^*L) + L = (KK^* + \varepsilon)L = K^*L$$

  donc K^*L est solution de $(\mathcal{E})$.
- Supposons que $\varepsilon \in K$. On peut écrire

  $$X = (K \setminus \varepsilon)X + X + L$$

  Donc, d'après le premier cas,

  $$X = ((K \setminus \varepsilon)^*(X + L) = K^*M$$

où $L \subset M = X + L$.
Inversement, si $L \subset M = L + Y$, alors

$$K(K^*M) + L = K^+(L + Y) + L = K^+L + K^+Y + L = K^*L + K^+Y$$

Mais le mot vide est dans K, et donc $K^+ = K^*$. Ainsi,

$$K(K^*M) + L = K^*L + K^*Y = K^*M$$

□

Théorème 2. *Soit $n \geq 1$. Soient $A_{ij}, 1 \leq i, j \leq n$ des langages ne contenant pas le mot vide. Soient également des langages $B_i, 1 \leq i \leq n$. Le système*

$$(\mathcal{S})L_i = \sum_{j=1}^{n} A_{ij}L_j + B_i$$

où $1 \leq i \leq n$, dont les inconnues sont les langages $L_i, i = 1, \ldots, n$, admet une unique solution. De plus, les L_i sont rationnels des que les A_{ij} et les B_i le sont.

Démonstration. On fait une récurrence sur n. C'est évident pour $n = 1$ d'après le lemme d'Arden. Supposons la propriété vraie pour $n - 1$. Considérons le système donné par l'énoncé. La $n$ième équation fournit, d'après le lemme d'Arden,

$$L_n = A_{nn}^*(\sum_{j=1}^{n-1} A_{nj}L_j + B_n)$$

En reportant dans les autres équations, on obtient le système de $n - 1$ équations

$$(\mathcal{S}')L_i = \sum_{j=1}^{n-1} A'_{ij}L_j + B'_i$$

où, pour $1 \leq i \leq n - 1$,

$$A'_{ij} = A_{ij} + A_{in}A_{nn}^*A_{nj}$$
$$B'_i = B_i + A_{in}A_{nn}^*B_n$$

On vérifie que les A'_{ij} ne contiennent pas le mot vide. Les langages A'_{ij} et B'_i sont rationnels des que les A_{ij} et les B_i le sont. On termine grâce à l'hypothèse de récurrence. □

Théorème 3. [Kleene]

Soit L un langage sur un alphabet A. Le langage L est rationnel si et seulement si il est reconnaissable par un automate fini.

Démonstration. Soit L un langage reconnu par l'automate fini $\mathcal{A} =< Q, A, E, I, T >$. Pour tout couple d'états (p, q), notons E_{pq} l'ensemble des étiquettes des transitions d'origine p et d'extrémité q. Ce langage est fini (éventuellement vide), donc rationnel. De plus, il ne contient pas le mot vide (vu qu'il ne contient que des mots d'une lettre).

Pour tout état p, notons

$$L_p = \{u \in A^*, \exists t \in T, p \xrightarrow{u} t\}$$

Un mot u est dans L_p si et seulement si il est l'étiquette d'un calcul d'origine p, dont l'extrémité est un état final de l'automate.

Notons enfin $\delta_{p,T} = \varepsilon$ si $p \in T$, et $\emptyset$ sinon. On a alors :

- $L = \sum_{p \in I} L_p$ (évident).
- $L_p = \sum_{q \in Q} E_{p,q} L_q + \delta_{p,T}$ (un peu moins évident).

Il en résulte que les L_p sont solutions d'un système d'équations analogue à celui du théorème précédent. On en déduit que les L_p sont rationnels, et L, réunion de langages rationnels, également. □

I.2 Un exemple complet

On se place sur l'alphabet $A = \{0, 1\}$. Soit L le langage des mots sur A qui sont les représentations des multiples de 3 en base 2. Le langage L est reconnu par l'automate fini de la figure 9.1.

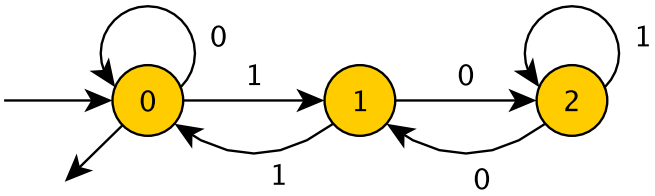


FIGURE 9.1 – Un automate reconnaissant les multiples de 3 en base 2

Pour ne pas s'embrouiller entre les états et l'alphabet, on pose $a = 0$ et $b = 1$. On a alors, avec les notations du théorème, $L = L_0$, et les langages L_0, L_1 et L_2 vérifient le système

$$\begin{cases} L_0 &= aL_0 + bL_1 + \varepsilon \\ L1 &= bL_0 + aL_2 \\ L2 &= aL_1 + bL_2 \end{cases}$$

La troisième équation fournit, par le lemme d'Arden,

$$L_2 = b^* a L_1$$

On reporte dans la deuxième équation, qui devient

$$L_1 = ab^* a L_1 + bL_0$$

D'où, par le lemme d'Arden,

$$L_1 = (ab^* a)^* b L_0$$

On reporte dans la première équation, qui devient

$$L_0 = (a + b(ab^* a)^* b) L_0 + \varepsilon$$

D'où

$$L = (a + b(ab^*a)^*b)^*$$

ou encore, en revenant aux vraies valeurs de a et b :

$$L = (0 + 1(01^*0)^*1)^*$$

II Existence de langages non rationnels

II.1 Facteurs itérants

Définition 1. Soit $u \in L$ où L est un langage sur l'alphabet A. Soient f, g et $h \in A^*$, $g \neq \varepsilon$. On dit que (f, g, h) est un facteur itérant de u dans L lorsque :

- $u = fgh$.
- $fg^*h \subset L$.

Exemple. Soit $L = a^*b^*$. Soit $u = a^3b^2$. Le mot u possède des facteurs itérants, comme par exemple (a, a^2, b^2) ou (a^2, a, b^2). En revanche, (a^2, ab, b) n'est pas un facteur itérant de u.

Exercice. Dans le langage $L = (a + b)^*c(a + b)^*$, le mot $aacb$ possède-t-il des facteurs itérants ? Et le mot c ?

II.2 Lemme de l'étoile « par blocs »

Théorème 4. *Soit L un langage rationnel sur l'alphabet A. Il existe un entier $N \geq 1$ tel que pour tout mot $u \in L$, pour toute factorisation $u = hv_1 \dots v_N k$ où h, k et les v_i sont des mots sur A, et les v_i sont non vides, il existe deux entiers $1 \leq i < j \leq N + 1$ tels que*

$$hv_1 \dots v_{i-1}(v_i \dots v_{j-1})^* v_j \dots v_N k \subset L$$

Remarque 1. On convient dans l'énoncé ci-dessus que $v_1 \dots v_{i-1} = \varepsilon$ si $i = 1$, et de même $v_j \dots v_N = \varepsilon$ si $j = N + 1$.

Démonstration. Soit $\mathcal{A} =< Q, A, E, I, T >$ un automate fini reconnaissant le langage L. Soit N le nombre d'états de l'automate. Soit $u \in L$, factorisé en $u = hv_1 \dots v_N k$ où les v_i sont non vides. Le mot u est l'étiquette d'un calcul réussi :

$$c := q_0 \xrightarrow{h} q_1 \xrightarrow{v_1} q_2 \xrightarrow{v_2} \dots \xrightarrow{v_N} q_{N+1} \xrightarrow{k} q_{N+2}$$

Comme l'automate ne possède que N états, il y a donc deux entiers $1 \leq i < j \leq N + 1$ tels que $q_i = q_j$. Notons $f = hv_1 \dots v_{i-1}$, $g = hv_1 \dots v_{j-1}$ et $l = v_j \dots v_N k$. On a alors le calcul réussi

$$q_0 \xrightarrow{f} q_i \xrightarrow{g} q_i \xrightarrow{l} q_{N+2}$$

Il est alors clair que $fg^n l$ est encore l'étiquette d'un calcul réussi, ceci pour tout entier n (une démonstration rigoureuse se faisant évidemment par récurrence sur n). □

II.3 Conséquences

Le lemme de l'étoile par blocs est souvent trop puissant pour les applications que l'on a à en faire. Donnons deux corollaires plus faibles, mais suffisants en pratique :

Corollaire 5. [Lemme de l'étoile]

Soit L un langage rationnel. Il existe un entier N tel que, pour tout mot $u \in L$, et toute factorisation $u = hvk$ où la longueur de v est supérieure ou égale à N, v peut être factorisé en $g_1 w g_2$, $w \neq \varepsilon$, de sorte que $hg_1 w^* g_2 k \subset L$.

Démonstration. Il suffit de considérer dans le lemme de l'étoile par blocs une factorisation où les v_i sont des lettres, et poser $v = v_1 \dots v_N$. □

Corollaire 6. [Lemme de l'étoile naïf]

Soit L un langage rationnel. Tout mot de L de longueur suffisamment grande possède un facteur itérant.

Démonstration. On prend $h = k = \varepsilon$ dans le corollaire précédent. □

II.4 Quelques exemples

Exemple. Considérons le langage $L = a^*cb^*$. On prend $N = 2$. Soit $u = hv_1v_2k \in L$. Forcément, v_1 où v_2 ne contient que des a ou que des b. Par exemple, v_1 ne contient que des a. Alors, $hv_1^*v_2k \subset L$.

Concernant le corollaire, tout mot de L de longueur supérieure ou égale à 2 contient un a ou un b, donc un facteur itérant.

Exemple. Soit $L = \{a^n b^n, n \geq 0\}$. Alors, L nest pas rationnel. Supposons pour cela qu'il le soit. Soit N l'entier donné par le théorème. Considérons $u = a^N b^N$. Prenons comme factorisation $h = k = \varepsilon$, $v_1 = \dots = v_N = a$. La contradiction est immédiate : la conclusion du théorème ne peut être vraie.

Exercice. Montrer plus simplement que dans le langage $L = \{a^n b^n, n \in \mathbb{N}\}$, il y a des mots de longueur aussi grande qu'on le veut qui n'ont pas de facteur itérant.

II.5 Existence de langages non rationnels

Le lemme de l'étoile est une condition nécessaire (mais pas suffisante, il s'en faut de peu ...) pour qu'un langage soit rationnel. On utilise très souvent ce lemme pour démontrer qu'un langage n'est pas rationnel.

Exemple. Le langage $L = \{a^n b^n, n \geq 0\}$ n'est pas rationnel (voir plus haut).

Exemple. Soit $L = \{a^p, p \text{ premier}\}$. Alors, L n'est pas rationnel. En effet, supposons le contraire. Soit $u = a^q a^r a^s$ factorisé à l'aide d'un fateur itérant (r non nul). On aurait alors, pour tout entier naturel n, $q + nr + s$ premier. En particulier, en prenant $n = q + s$, l'entier $(q + s)(r + 1)$ serait premier. Il y a donc contradiction. On remarque d'ailleurs que dans ce cas, *aucun* mot du langage ne possède de facteur itérant, ce qui est beaucoup plus fort que l'énoncé du corollaire du lemme de l'étoile.

Exemple. Le langage L des mots binaires qui sont l'écriture en base 2 d'un nombre premier est-il rationnel ? Nous allons montrer que non, en admettant une conjecture non prouvée à ce jour (à ma connaissance), à savoir l'existence d'une infinité de nombres premiers de la forme $M_n = 2^n - 1$. Ces nombres sont appelés les nombres de Mersenne. On montre que si M_n est premier, alors n est premier, la réciproque étant fausse. Soit $w = 111...111$ un nombre de Mersenne premier écrit en base 2. Le mot w possède n lettres, et n est premier. Si notre langage est rationnel, alors w possède un facteur itérant (pour n assez grand) : $w = 1^p 1^q 1^r$. Mais alors, $\forall m \in N$, on aurait $1^p 1^{mq} 1^r \in L$, c'est à dire $p + mq + r$ premier. C'est impossible. Il n'y a qu'à prendre $m = p + r$.

II.6 Les langages de programmation sont-ils rationnels ?

Ce parapgraphe peut être sauté en première lecture (voire en deuxième).

Nous allons prouver qu'un langage de programmation contenant le langage des expressions arithmétiques infixées (c'est à dire capable de comprendre des expressions du genre $1+(2*(3-(3-5)))$ ne peut pas être rationnel.

Proposition 7. *Soit L un langage rationnel sur l'alphabet A. Soit a une lettre de A. Alors, le langage obtenu en prenant les mots de L privés de leurs occurences de a est l'image de L par un morphisme de monoïdes.*

Démonstration. Soit $\phi : A^* \to A^*$ définie par

- $\forall x \in A, \phi(x) = x$ si $x \neq a$ et $\phi(a) = \varepsilon$
- ϕ est un morphisme de monoïdes (pour la concaténation).

ϕ étant définie sur les lettres, et tout mot se décomposant de façon unique comme produit de lettres, ϕ est donc bien définie sur A^*. □

Proposition 8. *L'image d'un langage rationnel par un morphisme de monoïdes est encore un langage rationnel.*

Démonstration. Par induction structurelle sur $Rat(A^*)$. Soit $F : A^* \to A^*$ un morphisme de monoïdes.

- Si L est un langage fini, $F(L)$ est fini et donc rationnel.
- Si L_1 et L_2 sont rationnels et tels que $F(L_1)$ et $F(L_2)$ soient rationnels, alors $F(L_1 \cup L_2) = F(L_1) \cup F(L_2)$ est rationnel.
- Si L_1 et L_2 sont rationnels et tels que $F(L_1)$ et $F(L_2)$ soient rationnels, alors $F(L_1.L_2) = F(L_1).F(L_2)$ (F est un morphisme pour la concaténation) est rationnel.

- Si L est rationnel et tel que $F(L)$ soit rationnel, alors $F(L^*) = F(L)^*$ est rationnel.

□

Proposition 9. *Le langage des parenthèses n'est pas rationnel.*

Démonstration. On montre qu'il ne vérifie pas le lemme de l'étoile. Supposons le contraire. Soit N l'entier associé au langage par le lemme. On considère le mot $u = ((((...))))$ possédant N parenthèses ouvrantes et N parenthèses fermantes et on factorise $u = hv_1 \dots v_N k$ en posant $h = \varepsilon$, $v_1 = \dots = v_N = (,$ et $k =)^n$ (désolé pour les notations, ce n'est pas ma faute). La conclusion du lemme de l'étoile est visiblement en défaut, puisque tout mot du langage des parenthèses contient autant de parenthèses ouvrantes que de parenthèses fermantes. □

Théorème 10. *Caml n'est pas rationnel.*

Démonstration. On suppose le contraire. On fabrique alors par suppression de toutes les lettres des programmes sauf les parenthèses (propriété 1) un langage qui est encore rationnel (propriété 2). Or, le langage restant est le langage des parenthèses. Il y a contradiction. □