Arbres

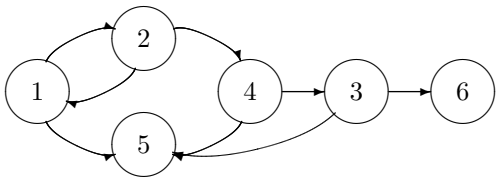
1 Graphes et arbres

1.1 Généralités sur les graphes

1.1.1 Définitions

Définition. Un graphe orienté est la donnée d'un couple (S, A) où S est un ensemble fini (les sommets) et A une partie de $S \times S$ (les arcs).

On représente un graphe par un diagramme (dit sagittal) avec des points (les sommets) et des flèches entre les sommets représentant les arcs.



Définition. Un graphe non orienté est la donnée d'un couple (S, A) où S est un ensemble fini (les sommets) et A un ensemble de parties à 2 éléments de S (les arêtes).

Remarque : Dans un graphe non orienté, on interdit généralement les boucles. A tout graphe orienté, on peut associer un unique graphe non orienté (en enlevant les boucles). En revanche, on peut définir de nombreuses orientations sur un graphe non orienté.
Graphe valué : un graphe est dit valué sur les sommets lorsqu'on dispose en plus d'une application de S dans $\mathbb{R}$, et valué sur les arcs lorsqu'on dispose d'une application de A dans $\mathbb{R}$.
Exemples : une carte géographique avec des distances (ou des temps de parcours) entre deux villes, une carte avec l'altitude (ou la population) de chaque endroit indiqué, un plan des pistes d'un domaine skiable...

Terminologie pour les graphes orientés

Si (a, b) est un arc, on dit que a est son origine et b sa terminaison. Le sommet b est appelé un successeur de a, et le sommet a un prédécesseur de b.
Le degré entrant d'un sommet est le nombre de ses prédécesseurs (noté $d^-(s)$) et le degré sortant le nombre de ses successeurs (noté $d^+(s)$). Le degré total est la somme des deux (lorsqu'il n'y a pas d'arc reliant un sommet à lui-même).
Un **chemin** est une suite de sommets $(a_0, \ldots, a_k)$ telle que pour tout i compris entre 0 et $k-1$, (a_i, a_{i+1}) soit un arc. Sa longueur est égale à k, nombre d'arcs le constituant. Le chemin est dit élémentaire lorsque tous les sommets le constituant sont distincts (à l'exception du premier et du dernier qui peuvent être égaux).
Un **circuit** est un chemin dont l'origine et la terminaison coïncident.
Le sommet a_j est dit accessible à partir du sommet a_i lorsqu'il existe un chemin d'origine a_i et d'extremité a_j.
Une **chaîne** est une suite de sommets $(a_0, \ldots, a_k)$ telle que pour tout i entre 0 et $k-1$, (a_i, a_{i+1}) ou (a_{i+1}, a_i) soit un arc.
Un **cycle** est une chaîne dont le premier et le dernier sommet coïncident.
La relation $\sim$ définie sur S par : "$x \sim y$ si et seulement si $x = y$ ou il existe un chemin reliant x et y et un chemin reliant y et x" est une relation d'équivalence.
Les classes d'équivalence pour cette relation sont appelées composantes connexes par arcs.

Un graphe orienté est **fortement connexe** si deux sommets distincts sont toujours reliés par un chemin.

Terminologie pour les graphes non orientés

Le **degré** d'un sommet est le nombre de ses voisins (i.e les éléments reliés au sommet par une arête).
Une **chaîne** est une suite de sommets $(a_0, \dots, a_k)$ (avec $k \geqslant 2$) telle que pour tout i compris entre 0 et $k-1$, $\{a_i, a_{i+1}\}$ soit une arête. Sa longueur est k, le nombre d'arêtes.
Un **cycle** est une chaîne dont le premier et le dernier élément coïncident.
La relation $\sim$ définie sur S par : "$x \sim y$ si et seulement si $x = y$ ou il existe une chaîne reliant x et y" est une relation d'équivalence.

Un graphe non orienté est **connexe** si deux sommets quelconques sont toujours reliés par une chaîne.
Les **composantes connexes** d'un graphe non orienté sont les parties maximales connexes, c'est-à-dire dont tous les sommets sont reliés entre eux par une chaîne.
Ce sont les classes d'équivalence de la relation définie plus haut. Elles forment une partition de S.

2. Matrice d'adjacence d'un graphe

Soit $G = (S, A)$ un graphe, avec $S = \{x_1, \dots x_n\}$. La matrice d'adjacence de G est la matrice $M = (m_{ij})_{1 \leqslant i,j \leqslant n}$ définie par $m_{ij} = \begin{cases} 1 & \text{si } (x_i, x_j) \in A \\ 0 & \text{sinon} \end{cases}$ (ou VRAI et FAUX si on veut une matrice booléenne)
Pour un graphe non orienté, la matrice d'adjacence est symétrique.

Exercice. Si M est la matrice d'adjacence et $k \in \mathbb{N}$, montrer que $M^k = (m_{ij}^{(k)})_{1 \leqslant i,j \leqslant n}$ où $m_{ij}^{(k)}$ est le nombre de chemins de longueur k allant de x_i à x_j.

Exercice. Soit G un graphe non orienté ayant n sommets, de matrice d'adjacence M.
Montrer que G est connexe si et seulement si $(I_n + M)^{n-1}$ a tous ses coefficients strictement positifs.

3. Implémentation en Caml

Les sommets d'un graphe seront désignés par les entiers de 0 à $n-1$.

- liste des arêtes : `(int*int) list`. Assez peu pratique.
- matrice d'adjacence : `int array array`.

  Le parcours du graphe a un coût de n^2. Cette représentation est plutôt adaptée aux graphes ayant beaucoup d'arcs.
- listes d'adjacence : `(int list) array`.
  `g.(i)` est la liste des successeurs du sommet i. L'espace mémoire utilisé est $n + p$.

Voir dessin.

Exercice. Dessiner le graphe représenté par les listes d'adjacence suivantes :
```
let graphe = [|[2; 5; 6]; [3; 4]; [0; 8]; [1; 4]; [1; 3]; [0; 7]; [0; 12]; [5; 12; 14];
[2; 9; 13; 16]; [8]; [11; 14; 15]; [10]; [6; 7; 14]; [8]; [7; 10; 12]; [10]; [8]|] ;;
```

Exercice. Ecrire une fonction Caml passant de la matrice d'adjacence d'un graphe à ses listes d'adjacence puis la fonction réciproque.

Exercice. Ecrire une fonction testant la connexité d'un graphe non orienté. Etudier sa complexité.

1.2 Arbres

On présente d'abord le point de vue non orienté.
Voir dessin.

Définition. Un arbre est un graphe non orienté connexe sans cycle.

Définition. Un sommet est dit pendant lorsqu'il possède exactement un voisin.

Proposition. *Soit G un arbre.*

- *Il existe des sommets pendants.*
- *Etant donnés deux sommets, il existe une chaîne unique les reliant.*

preuve. Par l'absurde, en partant d'un sommet quelconque, s'il n'y a pas de sommet pendant, on construit par récurrence une suite infinie de sommets distincts en considérant à chaque étape un nouveau voisin, distinct des précédents à cause de l'absence de cycle.
Avec le même argument, on peut toujours relier deux sommets par une chaîne, l'unicité provient de l'absence de cycle. •

Proposition. *Soit $G = (S, A)$ un arbre. Alors* $\operatorname{card} A = \operatorname{card} S - 1$.

preuve. On fait une récurrence sur le nombre de sommets. S'il y a $n+1$ sommets, on retire un sommet pendant et l'unique arête dont il est une extrêmité et on applique l'hypothèse de récurrence. •

Exercice. Soit $G = (S, A)$ un graphe tel que $\operatorname{card} A = \operatorname{card} S - 1$. Si G est connexe ou si G est sans cycle, montrer que G est un arbre.

1.3 Arbres enracinés

Définition. Un arbre enraciné (appelé aussi arborescence) est un graphe orienté sans cycle dans lequel tout sommet sauf un admet un prédécesseur.

Dans un arbre enraciné, les sommets sont appelés des nœuds et le sommet sans prédécesseur s'appelle la racine.
Chaque nœud autre que la racine admet donc un unique prédécesseur.
Si (x, y) est un arc, x est appelé le père de y et y est un fils de x. La relation $\preceq$ définie par : "$x \preceq y$ si et seulement si ($x = y$ ou il existe un chemin de x à y)" est une relation d'ordre.

Proposition. *Dans un arbre orienté, il existe un unique chemin partant de la racine allant jusqu'à un nœud donné.*

preuve. L'existence se démontre en partant du nœud à atteindre et en remontant les prédécesseurs jusqu'à aboutir à la racine, ce qui se produit au bout d'un nombre fini d'étapes car le graphe est fini et ne possède pas de cycle donc on ne peut pas retomber sur un autre nœud que la racine.
L'unicité provient de l'absence de cycle. •

La **profondeur** d'un nœud est la longueur du chemin y menant depuis la racine.
La **hauteur** de l'arbre est la profondeur maximale de ses nœuds.
Le **degré** (ou arité) d'un nœud est cette fois le nombre de ses fils.
Un nœud de degré 0 (c'est-à-dire n'ayant pas de fils) est appelé une **feuille**. Les autres nœuds sont appelés nœuds internes.
Un arbre n-aire est un arbre dont les nœuds sont de degré au plus n. Si $n = 2$, on parlera d'**arbre binaire**.

Proposition. *Si G est un graphe orienté, on note $\widehat{G}$ le graphe non orienté associé.*
La correspondance $T \longmapsto (racine(T), \widehat{T})$ est une bijection de l'ensemble des arbres enracinés dans l'ensemble des couples (r, A) où A est un arbre non orienté et r un sommet de A.

Autrement dit, on peut enraciner un arbre à partir de n'importe lequel de ses sommets. Voir dessin.

Exemples. La structure des dossiers et fichiers dans un système d'exploitation.
Un arbre généalogique.
L'organisation des calculs dans un système de calcul formel.

Sous-arbres

Proposition. *Soient (S, A) un arbre de racine r et $s_1, \ldots, s_p$ les fils de r.*
On pose $S_i = \{y \in S \mid s_i \preceq y\}$ et $A_i = A \cap S_i^2$. Alors pour tout i, (S_i, A_i) est un arbre (appelé sous-arbre de racine s_i), $(S_1, \ldots, S_p, \{r\})$ une partition de S et $(A_1, \ldots, A_p, \{(r, s_k) \mid 1 \leqslant k \leqslant p\})$ une partition de A.

preuve. Tout nœud autre que r est relié à un et un seul des s_i et (S_i, A_i) est un arbre de racine s_i. Il n'y a aucune liaison entre deux nœuds de S_i et S_j pour $i \neq j$. •

2 Structure d'arbre binaire

2.1 Définition récursive d'un arbre binaire

Définition. Un arbre binaire T est une structure de données finie définie sur un ensemble E telle que :
- soit T est vide,
- soit T est constitué d'un nœud étiqueté par un élément de E appelé racine et de deux sous-arbres binaires, appelés sous-arbres gauche et droit.

Un arbre atomique est un arbre réduit à sa racine (i.e dont les sous-arbres gauche et droit sont vides).
On notera $T = \langle r, T_g, T_d \rangle$ un arbre de racine r et de sous-arbres gauche et droit T_g et T_d.
Un nœud dont les deux sous-arbres sont vides est appelé une feuille. Les autres nœuds sont appelés nœuds internes.

Preuve par induction structurelle sur la structure d'arbre binaire.

Théorème. *Soit $\mathcal{P}$ une propriété définie sur un ensemble d'arbres binaires.*
On suppose que $\mathcal{P}$ est vraie pour l'arbre vide (ou pour un arbre atomique), et que si $\mathcal{P}$ est vraie pour deux arbres A_1 et A_2, alors $\mathcal{P}$ est vraie pour tout arbre binaire de sous-arbres gauche et droit A_1 et A_2.
Alors $\mathcal{P}$ est vraie pour tout arbre binaire.

preuve. par l'absurde, en considérant un arbre de hauteur minimale ne vérifiant pas $\mathcal{P}$ et ses sous-arbres. •

2.2 Implémentation Caml

On présente des implémentations persistantes des arbres binaires.

Arbre binaire homogène

Un arbre binaire homogène est un arbre dans lequel tous les nœuds sont du même type.
L'implémentation usuelle est : `type 'a arbre = Nil | Noeud of 'a * 'a arbre * 'a arbre ;;`
Chaque feuille a bien pour sous-arbres deux arbres vides.
Exemple : Voir dessin.

```
let g = Noeud(10,Noeud(15,Noeud(3,Nil,Nil),Noeud(8,Nil,Nil)),Noeud(20,Nil,Nil));;
let d = Noeud(1,Noeud(6,Nil,Noeud(2,Nil,Nil)),Nil);;
let a = Noeud(30,g,d) ;;
```

Arbre binaire hétérogène

Dans cette implémentation, il n'y a pas d'arbre vide. Les feuilles sont du type `'f` et les autres nœuds du type `'n`.

```
type ('f,'n) arbre = Feuille of 'f | Noeud of 'n * ('f,'n) arbre * ('f,'n) arbre ;;
```

Exemples : expression arithmétique (ou booléenne), arbre de décision.

2.3 Fonctions classiques sur les arbres binaires

Soit T un arbre binaire. Si T n'est pas réduit à une feuille, le nombre de nœuds de T est la somme du nombre de nœuds des sous-arbres gauche et droit de T plus 1.
La hauteur de T est le maximum des hauteurs de ses sous-arbres gauche et droit auquel il faut ajouter 1.
Un arbre atomique est de hauteur 0 (on convient parfois que l'arbre vide est de hauteur -1).

Implémentation pour les arbres binaires homogènes

```
let rec taille = function  Nil -> 0
 | Noeud(_,g,d) -> 1 + taille g + taille d
;;
let rec num_feuille = function   Nil -> 0
  | Noeud(_,Nil,Nil) -> 1
  | Noeud(_,g,d) -> num_feuille g + num_feuille d
;;
let rec hauteur = function   Nil -> -1
  | Noeud(_,g,d) -> 1 + max (hauteur g) (hauteur d)
;;
```

La complexité est de l'ordre du nombre de nœuds de l'arbre, puisqu'on les visite tous une et une seule fois.

2.4 Relation entre nombre de nœuds et nombre de feuilles

Définition. Un arbre binaire est localement complet lorsque tout nœud interne possède deux fils.

Voir dessin.

Proposition. *Soient N le nombre de nœuds internes et F le nombre de feuilles d'un arbre binaire.*
Alors $F \leqslant N + 1$.
Il y a égalité si et seulement si l'arbre est localement complet.

preuve. par induction structurelle. Cas de base : arbre vide ou arbre atomique.
Hérédité : Si $a = \langle r, g, d \rangle$, $F = F_g + F_d$, $N = 1 + N_g + N_d$. $F \leqslant N_g + 1 + N_d + 1 = N + 1$.
Il y a égalité si et seulement si il y a égalité pour les sous-arbres gauche et droit, ce qui conduit par récurrence à un arbre localement complet. •

2.5 Parcours en profondeur d'un arbre binaire

On suppose disposer d'un traitement `f` pouvant s'appliquer sur chaque nœud d'un arbre binaire.
Un parcours en profondeur consiste à traiter chaque nœud en parcourant le sous-arbre gauche avant le sous-arbre droit. L'ordre dans lequel on traite la racine détermine le type de parcours de l'arbre.

Le parcours préfixe de l'arbre consiste d'abord à traiter sa racine, puis à parcourir récursivement le sous-arbre gauche et enfin le sous-arbre droit.

```
let rec prefixe f = function   (* f est de type 'a -> unit *)
    Nil -> ()
  | Noeud (r,g,d) -> f r ; prefixe f g ; prefixe f d ;;
```

Le parcours infixe de l'arbre consiste d'abord à parcourir le sous-arbre gauche, puis à traiter la racine, puis à parcourir le sous-arbre droit.

```
let rec infixe f = function
    Nil -> ()
  | Noeud (r,g,d) -> infixe f g ; f r ; infixe f d ;;
```

Le parcours postfixe de l'arbre consiste d'abord à parcourir le sous-arbre gauche, puis le sous-arbre droit, et enfin à traiter la racine.

```
let rec postfixe f = function
    Nil -> ()
  | Noeud (r,g,d) -> postfixe f g ; postfixe f d ; f r ;;
```

La complexité est de l'ordre du nombre de nœuds de l'arbre, puisqu'on les visite tous une et une seule fois.
Voir dessin.

Exemple : pour imprimer avec un espace des étiquettes entières :
`let f x = print_int x ; print_string " " ;;`

2.6 Arbre associé à une expression arithmétique

On définit par induction structurelle l'ensemble des expressions arithmétiques (en abrégé EA) de la façon suivante :
- Une variable x ou une constante c sont des EA,
- Si E est une EA, $(-E)$ est une EA,
- Si E_1 et E_2 sont des EA, $(E_1 + E_2)$, $(E_1 - E_2)$ et $(E_1 \cdot E_2)$ sont des EA.

On peut représenter une EA de deux manières :
- Par une expression linéaire parenthésée (comme ci-dessus). Par exemple, $((x + y) \cdot z)$ est une EA.
- Par un arbre binaire dont les nœuds internes sont étiquetés par les opérateurs unaires ou binaires $+$, $-$, $\times$ et les feuilles par les variables et les constantes. Voir dessin.

L'évaluation d'une EA sur des valeurs données est le résultat du calcul lorsqu'on remplace les variables figurant dans l'EA par les valeurs données.
Par exemple, l'évaluation de $((x+y)\cdot z)$ pour $(x,y,z)=(3,-1,4)$ donne 8.

Représentations en Caml.

Le type `ea` permet de représenter les EA par un arbre et le type `lex list` par une liste parenthésée.

```
type ea = Int of int | Var of string | Plus of ea * ea | Moins of ea * ea | Mult of ea *
ea | Chs of ea ;;
type lex = V of string | C of int | G | D | Pl | Mo | Mu | Ch ;;
```

- La fonction `lex_of_arbre : ea -> lex list` calcule l'EA sous forme de liste de lexèmes à partir de son écriture arborescente.

```
let rec lex_of_arbre = function
 Int n -> [C n]
| Var x -> [V x]
| Plus (x,y) -> G :: (lex_of_arbre x) @ (Pl :: (lex_of_arbre y)) @ [D]
| Moins (x,y) -> G :: (lex_of_arbre x) @ (Mo :: (lex_of_arbre y)) @ [D]
| Mult (x,y) -> G :: (lex_of_arbre x) @ (Mu :: (lex_of_arbre y)) @ [D]
| Chs x -> G :: Ch :: (lex_of_arbre x) @ [D] ;;
```

- La fonction `ecrit_tot_par : ea -> string` écrit une EA comme une chaîne de caractères totalement parenthésée.

```
let rec ecrit_tot_par = function
     Int n -> string_of_int n
   | Var x -> x
   | Plus (x,y) ->  "(" ^ (ecrit_tot_par x) ^ "+" ^ (ecrit_tot_par y) ^ ")"
   | Moins (x,y) -> "(" ^ (ecrit_tot_par x) ^ "-" ^ (ecrit_tot_par y) ^ ")"
   | Mult (x,y) ->  "(" ^ (ecrit_tot_par x) ^ "*" ^ (ecrit_tot_par y) ^ ")"
   | Chs x ->  "(" ^ "-" ^ (ecrit_tot_par x) ^ ")" ;;
```

- Syntaxe et sémantique : les deux arbres associés aux expressions arithmétiques $x\cdot(y+z)$ et $(x\cdot y)+(x\cdot z)$ sont distincts mais ont la même sémantique (même évaluation quelles que soient les valeurs attribuées aux variables).

- Evaluation : soit A un ensemble de variables et soit $v : A \to \mathbb{Z}$ une valuation (c'est-à-dire une affectation de valeur pour chaque variable). La fonction `evalue : ea -> (string * int) list -> int` retourne la valeur d'une expression arithmétique selon une certaine valuation.

```
let rec evalue a l = match a with
   Int n -> n
 | Var x -> List.assoc x l
 | Plus (x,y) -> (evalue x l) + (evalue y l)
 | Moins (x,y) -> (evalue x l) - (evalue y l)
 | Mult (x,y) -> (evalue x l) * (evalue y l)
 | Chs x -> - (evalue x l)
;;
let ea = Mult (Plus (Var "x", Int 5), Plus (Var "y", Chs (Var "z"))) ;;
evalue a1 ["x",1;"y",3;"z",6] ;;
```

Complément : analyse syntaxique d'une expression arithmétique

On se propose d'écrire une fonction `arbre_of_lex` calculant l'arbre associé à une EA écrite sous la forme d'une liste de lexèmes.
Le principe est de décomposer une EA dont l'opérateur principal est binaire sous la forme `E1 op E2`, où `op` est l'opérateur principal et `E1` et `E2` sont les expressions figurant à gauche et à droite de `op`. Si l'expression est sous la forme `-E`, on extrait simplement `E`.
La fonction `aux` réalise cette scission en lisant l'expression privée des parenthèses externes et en s'arrêtant lorsqu'on a obtenu autant de parenthèses ouvrantes que de fermantes.
Par exemple, `aux 0 [G;V "x";Pl;C 5;D;Mu;G;V "y";Pl;G;Ch;V "z";D;D] []` renvoie le couple de listes `([G; V "x"; Pl; C 5; D], [Mu; G; V "y"; Pl; G; Ch; V "z"; D; D])`

Il suffit ensuite d'appliquer récursivement l'analyse syntaxique sur les expressions de gauche et de droite obtenues (ou sur `E` si on part de `-E`).

```
let rec der = function (* renvoie une liste privée de son dernier élément *)
[_] -> []
   | t::q -> t::(der q)
;;
let rec aux k l av = match k,l with
   0,_ when av <> [] -> List.rev av,l
  | _,G::q -> aux (k+1) q (G::av)
  | _,D::q -> aux (k-1) q (D::av)
  | _,a::q -> aux k q (a::av)
;;
let rec arbre_of_lex = function
   [] -> failwith "erreur"
 | [V x] -> Var x
 | [C n] -> Int n
 | G::Ch::l -> Chs(arbre_of_lex (der l))
 | G::l -> let (g,ll) = aux 0 (der l) [] in match ll with
     | Pl::d -> Plus(arbre_of_lex g,arbre_of_lex d)
   | Mo::d -> Moins(arbre_of_lex g,arbre_of_lex d)
   | Mu::d -> Mult(arbre_of_lex g,arbre_of_lex d)
 |_ -> failwith "erreur"
;;
(* Tests *)
let ea1 = [G;G;V "x";Pl;C 5;D;Mu;G;V "y";Pl;G;Ch;V "z";D;D;D];;
let ea3 = [G;C 10;Pl;G;C 5;Pl;G;C 3;Pl;C 1;D;D;D] ;;
arbre_of_lex ea1 ;;
arbre_of_lex ea3 ;;
```