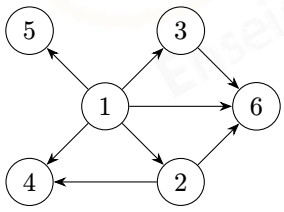
Arbres et graphes

UE Informatique, M1 Mathématiques et applications,
EADS, Université d'Aix-Marseille *

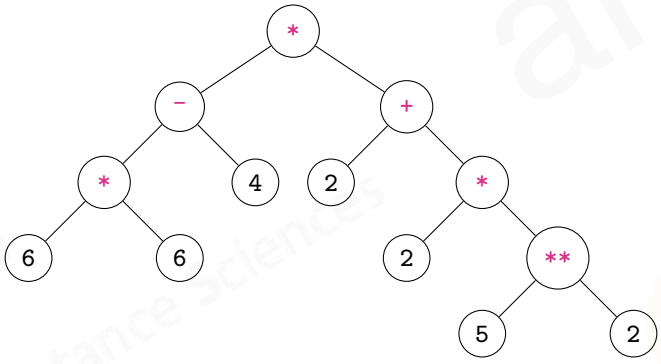
2024–2025

Dans ce chapitre, on aborde des structures de données non séquentielles : les arbres et les graphes. Dans les deux cas, on peut considérer qu'il s'agit de représenter une relation entre les éléments d'un ensemble fini : les *sommets* (ou *nœuds* ou *points*) de l'arbre ou du graphe.

Dans le cas d'un graphe, la relation est définie par les *arcs* (ou *lignes* ou *arêtes* ou *flèches*) : chaque arc met en relation deux sommets. Par exemple, voici le graphe de la relation de divisibilité stricte sur les entiers naturels de 1 à 6 (on a une flèche de i à j quand $i \neq j$ et i divise j) :



Dans le cas d'un arbre, la relation considérée est une relation de filiation : chaque nœud a au plus un *nœud parent*, mais une collection (ordonnée ou non) de *nœuds enfants*. De plus, chaque arbre possède une unique *racine*, c'est-à-dire un nœud sans parent. Par exemple, voici l'arbre syntaxique de l'expression arithmétique Python `((6*6)-4)*(2+(2*5**2))` :



Graphiquement, on n'a pas besoin de faire figurer l'orientation des arcs, qui est implicitement donnée par les positions verticales relatives des nœuds.

Les arbres sont donc des graphes particuliers, mais leurs propriétés, leur représentation informatique, comme l'algorithmique qui les concerne sont tellement spécialisées qu'on les traite séparément.

*Ce support de cours est ©L. Vaux Auclair, amU, 2024–2025, et mis à disposition selon les termes de la licence : Creative Commons Attribution – Pas d'utilisation commerciale – Partage dans les mêmes conditions 4.0 International

1 Arbres

1.1 Définition et représentation

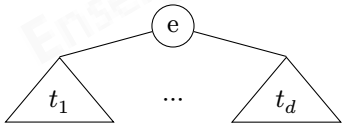
1.1.1. Définition (Arbres ordonnés)**.** Étant donné un ensemble E (l'ensemble des *étiquettes*), un *arbre ordonné* (on dira plus simplement : un *arbre*) à étiquettes dans E est un couple $t = (e, u)$ où $e \in E$ et $u = (t_1, \dots, t_d)$ est, inductivement [1], une liste finie d'arbres à étiquettes dans E. On note $\mathcal{A}(E)$ l'ensemble de ces arbres, et on note $\mathcal{A} = \mathcal{A}(E)$ dans le cas particulier où E est un singleton (dans ce cas, on omet les étiquettes, et l'arbre est réduit à la liste u) : les arbres de $\mathcal{A}$ sont dits *muets*, c'est-à-dire sans étiquette.

Plus explicitement, on peut poser $X_0 = \emptyset$ et $X_{n+1} = E \times X_n^*$ (resp. $X_{n+1} = X_n^*$), et remarquer que la suite d'ensembles (X_n) est croissante pour l'inclusion : alors $\mathcal{A}(E) = \bigcup_{n\in\mathbf{N}} X_n$ (resp. $\mathcal{A} = \bigcup_{n\in\mathbf{N}} X_n$). En d'autres mots, l'ensemble $\mathcal{A}(E)$ (resp. $\mathcal{A}$) est le plus petit ensemble X tel que $E \times X^* \subseteq X$ (resp. $X^* \subseteq X$). En particulier, il contient une copie de E *via* l'injection $e \in E \mapsto (e, \varepsilon) \in \mathcal{A}(E)$ (de même $\varepsilon \in \mathcal{A}$).

À cette construction ensembliste, on peut associer une intuition graphique :

— un arbre $t = (e, u)$ est constitué d'un nœud *racine*, étiqueté par e, qui est le nœud parent de chacun des arbres *enfants*, donnés par le mot $u = t_1 \cdots t_d$:

$(e, t_1 \cdots t_d)$ =



— lorsque u est vide, t est donc réduit à un nœud : c'est une *feuille* ;

— sinon la racine de t est un nœud *interne*.

Notez qu'on a choisi ici de placer la racine en haut, et que les arbres croissent vers le bas. [2] Les définitions de taille et de hauteur d'un arbre sont directement tirées de cette représentation :

1.1.2. Définition (Grandeurs d'un arbre)**.** Le *degré* d'un arbre $t = (e, u)$ est le nombre de ses enfants : $\mathrm{d}(t) = |u|$. On note alors $t[i] = u[i]$ l'enfant i de t, pour $0 \leq i < \mathrm{d}(t)$. La *taille* $\#t$ et la *hauteur* $\mathrm{h}(t)$ sont alors définies inductivement [3] en posant : $\#t = 1 + \sum_{0\leq i<\mathrm{d}(t)} \#t[i]$ et $\mathrm{h}(t) = 1 + \max_{0\leq i<\mathrm{d}(t)} \mathrm{h}(t[i])$.

En particulier, les conditions suivantes sont deux-à-deux équivalentes :

$$t \text{ est réduit à une feuille} ; \qquad \mathrm{d}(t) = 0 ; \qquad \#t = 1 ; \qquad \mathrm{h}(t) = 1.$$

De plus il est évident par définition que, pour $0 \leq i < \mathrm{d}(t)$, on a $\mathrm{h}(t[i]) < \mathrm{h}(t)$; et on peut vérifier que $\mathrm{h}(t)$ est le plus petit $n \in \mathbf{N}$ tel que $t \in X_n$, avec les notations suivant la définition 1.1.1. [4]

1. Dans une définition inductive, on indique comment construire un objet de l'ensemble en cours de définition, en s'autorisant à utiliser des objets du même ensemble, supposés déjà construits. Malgré son apparente circularité, une telle définition est correcte par construction : c'est un type de définition par récurrence (forte), comme le montre la discussion qui suit.

2. C'est bien sûr une pure convention : elle va contre l'habitude « naturelle » (les plantes qui nous entourent poussent plutôt vers le haut…), mais elle est plus pratique sur papier ou écran, où le texte progresse vers le bas.

3. Comme on a des définitions et des preuves par récurrence, on a plus généralement des définitions et des preuves par induction : on définit une fonction (ou on prouve un résultat) sur un arbre, à partir des valeurs de cette fonction (ou de la vérité du résultat) sur les enfants de l'arbre.

4. On pourrait donc remplacer la définition inductive des arbres par une définition par récurrence sur une borne de la hauteur ; et raisonner inductivement sur les arbres revient à raisonner par récurrence sur la hauteur. Vous pouvez garder ces considérations en tête si le vocabulaire « inductif » vous trouble.

©L. Vaux Auclair, amU, 2024–2025

1.1.3. *Exemple* (Arbre d'une expression). L'arbre de l'expression `((6*6)-4)*(2+(2*5**2))`, présenté au début du chapitre, est défini formellement comme :

$$\Big(\texttt{*}, \Big(\texttt{-}, (\texttt{*}, (\texttt{6}, \varepsilon)(\texttt{6}, \varepsilon))(\texttt{4}, \varepsilon)\Big)\Big(\texttt{+}, (\texttt{2}, \varepsilon)(\texttt{*}, (\texttt{2}, \varepsilon)(\texttt{**}, (\texttt{5}, \varepsilon)(\texttt{2}, \varepsilon)))\Big)\Big).$$

C'est évidemment un cauchemar à lire (les habitués du LISP apprécieront), mais on reconnait :

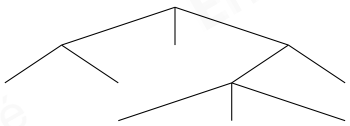
— les feuilles, de la forme (e, ε), qui apparaissent dans l'ordre ;

— la notation polonaise (voir l'exercice sur la notation polonaise inverse, dans le cours précédent), qui s'obtient en oubliant toutes les parenthèses et les ε :

$$\texttt{*}, \texttt{-}, \texttt{*}, \texttt{6}, \texttt{6}, \texttt{4}, \texttt{+}, \texttt{2}, \texttt{*}, \texttt{2}, \texttt{**}, \texttt{5}, \texttt{2}.$$

Cet arbre est de taille 13 (le nombre de nœuds) et de hauteur 5 (le plus grand nombre possible de nœuds sur un chemin menant de la racine à une feuille).

1.1.4. *Exemple* (Arbre muet). En l'absence d'étiquettes, seule compte la structure du parenthésage, et par exemple $\Big(\big(()()\big)\big(\,\big)\big(((()())())()\big)\Big)$, de degré 3, de taille 11 (le nombre de parenthèses ouvrantes) et de hauteur 4 (le niveau maximum d'imbrication des parenthèses), est réprésenté comme :



Les arbres ont une structure intrinsèquement récursive : chaque arbre a des enfants, qui eux-mêmes ont des enfants, qui eux-mêmes… Cette structure est capturée par la notion de sous-arbre.

1.1.5. Définition (Adresses et sous-arbres)**.** L'ensemble des *adresses* de $t = (e, u)$ est l'ensemble de mots $\mathrm{adr}(t) \subset \mathbf{N}^*$ défini inductivement par : $\mathrm{adr}(t) = \{\varepsilon\} \cup \bigcup_{i<\mathrm{d}(t)} i \cdot \mathrm{adr}(t[i])$ (une adresse de t est donc ou bien le mot vide ε, ou bien un mot de la forme ip avec $0 \leq i < \mathrm{d}(t)$ et p une adresse de $t[i]$). Étant donnée une adresse $p \in \mathrm{adr}(t)$, on définit le *sous-arbre de t à l'adresse p* inductivement par : $t[\varepsilon] = t$ et $t[ip'] = t[i][p']$. Si de plus $|p| = n$, on dit que $t[p]$ est *à profondeur n dans t*.

Notez que la notation généralise celle utilisée pour les listes ou les mots : les sous-arbres à profondeur 1 sont les enfants et, si $0 \leq i < \mathrm{d}(t)$, le sous-arbre à l'adresse i (vue comme un mot de longueur 1) est l'enfant $t[i] = t[i][\varepsilon]$.

1.1.6. Proposition. *On a $\mathrm{h}(t) = 1 + \mathrm{h_i}(t)$ où $\mathrm{h_i}(t)$ est la longueur maximale d'une adresse dans t, c'est-à-dire la profondeur maximale des feuilles de t.*

Démonstration. C'est direct par induction sur les arbres. En particulier, si t est réduit à une feuille, $\mathrm{h}(t) = 1$ et $\mathrm{h_i}(t) = 0$. Et pour un nœud interne, l'hypothèse d'induction s'applique directement. □

1.1.7. Exercice (Adresses et sous-arbres)**.**

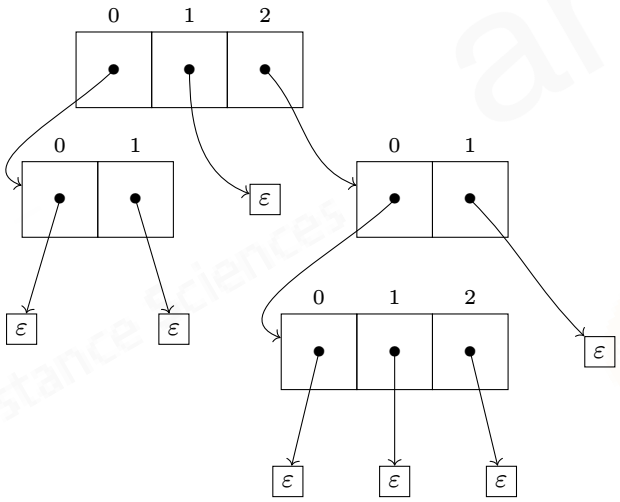
1. Vérifiez que le nombre d'adresses d'un arbre est exactement sa taille.
2. Quel est le sous-arbre à l'adresse 00 de l'arbre de l'exemple 1.1.3 ?
3. Calculez l'ensemble des adresses de l'arbre de l'exemple 1.1.4. ◇

©L. Vaux Auclair, amU, 2024–2025

```python
class Arbre:
    """Une classe pour les arbres étiquetés ou muets."""
    def __init__(self,enfants=(),valeur=None):
        """On crée un arbre à partir de la liste de ses enfants (de n'importe
        quel type séquentiel, vide par défaut), et d'une éventuelle étiquette
        pour la racine."""
        self.valeur = valeur
        self.enfants = tuple(enfants)
    def __getitem__(self,i):
        return self.enfants[i]
    def __len__(self):
        return len(self.enfants)
    def __repr__(self):
        valeur_s = "" if self.valeur is None else repr(self.valeur)
        enfants_s = ",".join(repr(t) for t in self.enfants)
        if enfants_s:
            if valeur_s:
                return f'Arbre([{ enfants_s }],{ valeur_s })'
            return f'Arbre([{ enfants_s }])'
        if valeur_s:
            return f'Arbre(valeur={ valeur_s })'
        return "Arbre()"
```

FIGURE 1 – Une classe Python pour représenter les arbres ordonnés (fichier `arbres_base.py` joint au cours).

La définition des arbres se prête à une représentation directe avec des listes : un arbre, c'est une liste d'arbres (plus, éventuellement, une étiquette). Comme la taille d'un arbre n'est pas bornée, on doit travailler avec des pointeurs. Par exemple, si on suppose disposer d'une représentation des listes comme tableaux de pointeurs vers des objets, on peut représenter en mémoire l'arbre muet de l'exemple 1.1.4 comme

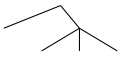


où on retrouve exactement la forme de l'arbre !

1.1.8. *En Python* . On peut directement coder un arbre comme un couple `(etiquette, liste_des_enfants)` (ou juste comme une liste d'enfants, pour un arbre muet), ou bien définir une classe dédiée, par exemple celle de la figure 1, où on prend le parti de donner d'abord la

©L. Vaux Auclair, amU, 2024–2025

liste des enfants, ce qui permet, par le jeu des arguments optionnels, d'avoir une notation concise pour les arbres muets. Elle permet par ailleurs d'obtenir l'enfant `i` de `a` en écrivant `a[i]`, et `len(a)` donne le degré de `a`.

Dans la suite, on se basera sur cette classe, qui permet de représenter l'arbre muet  par `Arbre([ Arbre(), Arbre([Arbre(), Arbre(), Arbre()])])`, et l'arbre de l'expression `2*(5**2)` par

```
Arbre([
    Arbre(valeur=2),
    Arbre([
        Arbre(valeur=5),
        Arbre(valeur=2)
    ], "**")
],"*")
```

1.1.9. Exercice (Arbres en Python).

1. Munissez la classe `Arbre` de méthodes `taille(self)` et `hauteur(self)`, avec le sens évident. Avec `a = Arbre([ Arbre(), Arbre([Arbre(), Arbre(), Arbre()])])`, on devrait avoir `a.taille()== 6` et `a.hauteur()== 3`.
2. Écrivez une méthode `sousarbre(self,adresse)` qui retourne le sous-arbre à l'`adresse` fournie, lorsqu'`adresse` représente un mot qui est bien une adresse de l'`arbre` (sinon, la fonction échoue, par exemple en soulevant l'exception `ValueError`). Vous pourrez alors étendre la méthode `__getitem__` ci-dessus avec

   ```
   def __getitem__(self,i):
       if isinstance(i,int):
           return self.enfants[i]
       return self.sousarbre(i)
   ```

   En particulier, `print(a[(1,2)])` devrait afficher quelque chose comme `Arbre()`.
3. Par contre, on n'a pas `a[(1,2)] == Arbre()` : pourquoi ? ◇

1.2 Parcours

La définition inductive des arbres se prête particulièrement à un traitement récursif : quand on travaille sur un arbre, il est naturel de se ramener à un traitement des sous-arbres. C'est déjà de cette manière qu'on été définies la taille et la hauteur et, plus généralement, un algorithme sur les arbres se présente le plus souvent comme un parcours de la structure *en profondeur*, c'est-à-dire visitant les enfants de l'arbre successivement.

1.2.1. Exercice (Égalité entre arbres). Définissez une méthode `__eq__(self, autre)` qui teste l'égalité entre arbres, en utilisant sa caractérisation inductive : pour $t = (e, u)$ et $t' = (e', u')$, on a $t = t'$ ssi $e = e'$, $\mathrm{d}(t) = \mathrm{d}(t')$ et, pour $0 \leq i < \mathrm{d}(t)$, $t[i] = t'[i]$ (inductivement). Vérifiez qu'on a bien `Arbre()== Arbre()`, ou encore `a[(1,2)] == Arbre()` avec l'exemple précédent. ◇

La terminologie standard distingue deux cas particuliers, suivant quand intervient le traitement de la racine : avant (cas d'un *parcours préfixe*) ou après (cas d'un *parcours suffixe*) celui des enfants.

1.2.2. Exercice (Recherche d'une valeur dans un arbre). Définissez une méthode `trouve(self,valeur)` qui renvoie l'adresse la plus petite pour l'ordre lexicographique à laquelle se trouve la `valeur` donnée (ou None s'il n'y en a pas) : ça se fait naturellement avec un parcours préfixe.

©L. Vaux Auclair, amU, 2024–2025

Par exemple, pour `a=Arbre([Arbre([Arbre(valeur=1),Arbre([Arbre(valeur=1)])])])`, on devrait avoir `a.trouve(1)== (0,0)`. ◇

Cette distinction est en fait un peu réductrice, puisque rien n'interdit d'effectuer un traitement à la fois avant et après la visite des enfants, ou même entre chaque visite d'enfant. Par exemple, dans la méthode `__repr__()` de la classe `Arbre` en figure 1, le fait de calculer `valeur_s` avant `enfants_s` suggère un parcours préfixe, mais on utilise cette chaine après le calcul de `enfants_s` ; de plus, le calcul de `enfants_s` fait intervenir l'expression `",".join(repr(t) for t in self.enfants)`, qui cache un traitement intermédiaire entre deux visites d'arbres enfants, consistant à empiler le résultat de chaque appel récursif pour construire la chaine. La distinction préfixe/suffixe a donc plus à voir avec la forme du résultat qu'avec l'ordre véritable dans lequel on accède aux informations.

1.2.3. Exercice (Affichage d'arbres)**.** Définissez deux nouvelles méthodes `chaine_prefixe(self)` et `chaine_suffixe(self)` qui retournent des chaines représentant l'arbre considéré vu comme un couple `(valeur,enfants)` (cas préfixe) ou `(enfants,valeur)` (cas suffixe). Pour alléger la notation, on peut ignorer la liste (vide) des enfants des feuilles, et ne renvoyer qu'une chaine représentant leur valeur. Par exemple, avec

```
e=Arbre([Arbre(valeur=2),Arbre([Arbre(valeur=5),Arbre(valeur=2)],"**")],"*")
```

`print(e.chaine_prefixe())` devrait afficher quelque chose comme `(*,[2,(**,[5,2])])` et `print(e.chaine_suffixe())` devrait afficher quelque chose comme `([2,([5,2],**)],*)`. ◇

1.2.4. *En Python* (Parcours du système de fichiers). La fonction `walk(dossier)` du module `os` permet de parcourir l'arbre de dossiers défini par le système de fichiers à partir du `dossier` racine (dans l'ordre préfixe par défaut, et dans l'ordre suffixe avec l'argument optionnel `topdown=False`). Le résultat est un générateur (voir le cours précédent) qui, pour chaque sous-arbre (c'est-à-dire chaque sous-dossier) rencontré, retourne un triplet `chemin, dossiers, fichiers` : `chemin` est le chemin du sous-dossier courant, `dossiers` est la liste des noms de ses sous-dossiers enfants, et `fichiers` est la liste des fichiers présents.

Vous pouvez par exemple explorer l'arborescence du dossier courant de l'interpréteur avec :

```
import os
for chemin, dossiers, fichiers in os.walk("."):
    print(chemin,":",str(fichiers))
```

qui affichera, pour chaque sous-dossier du dossier courant (en commençant par celui-là), son chemin et la liste des fichiers qu'il contient.

1.2.5. Exercice (Recherche du fichier le plus volumineux)**.** La fonction `stat(fichier)` du module `os` retourne un objet dont l'attribut `st_size` donne la taille en octets du fichier considéré. En utilisant `os.walk` et `os.stat`, définissez une fonction `gros_fichier(dossier)` qui retourne le chemin du fichier le plus volumineux de l'arborescence de racine `dossier`. ◇

1.3 Arbres binaires

Il est souvent utile de contraindre le degré des arbres. Typiquement, les expressions arithmétiques utilisent généralement des constantes, et des opérations unaires ou binaires : il suffit de considérer des arbres de degré au plus 2. Chaque arbre de degré 2 a un enfant gauche (0) et un enfant droit (1). Si on ne traverse que des sous-arbres de degré 2, une adresse est donc un mot binaire. Ça suggère la définition suivante :

1.3.1. Définition (Arbres binaires)**.** Un *arbre binaire* à valeurs dans V et à feuilles dans F est :

©L. Vaux Auclair, amU, 2024–2025

— ou bien une feuille $f \in F$;
— ou bien un triplet (v, g, d) (un nœud interne) formé d'une valeur $v \in V$, d'un sous-arbre gauche g et d'un sous-arbre droit d.

On note $\mathcal{A}_2(V, F)$ l'ensemble de ces arbres. On note $\mathcal{A}_2(V)$ dans le cas particulier où F est un singleton, et $\mathcal{A}_2$ dans le cas où V et F sont des singletons (et alors on considère un nœud interne comme un couple (g, d)).

La taille et la hauteur des arbres binaires sont définies inductivement par

$$\begin{aligned} \#f &= 1 & \#(v,g,d) &= 1 + \#g + \#d \\ \mathrm{h}(f) &= 1 & \mathrm{h}((v,g,d)) &= 1 + \max(\mathrm{h}(g), \mathrm{h}(d))\,. \end{aligned}$$

L'ensemble des *adresses* d'un arbre t est l'ensemble de mots $\mathrm{adr}(t) \subset \mathbf{B}^*$ défini inductivement par : $\mathrm{adr}(f) = \{\varepsilon\}$ pour $f \in F$, et $\mathrm{adr}((v,g,d)) = 0 \cdot \mathrm{adr}(g) \cup 1 \cdot \mathrm{adr}(d)$. Étant donnée une adresse $p \in \mathrm{adr}(t)$, on définit le *sous-arbre de t à l'adresse p* inductivement par : $t[\varepsilon] = t$, $(v,g,d)[0p] = g[p]$ et $(v,g,d)[1p] = d[p]$.

Notez qu'on a compté les feuilles dans la taille, par souci de cohérence avec les arbres ordonnés généraux. Mais cette information n'est pas vraiment nécessaire :

1.3.2. Proposition. *Pour tout arbre binaire t, on note $\#_\mathrm{f} t$ le nombre de ses feuilles, et $\#_\mathrm{i} t$ le nombre de ses nœuds internes. Alors $\#_\mathrm{f} t = \#_\mathrm{i} t + 1$, et $\#t = \#_\mathrm{f} t + \#_\mathrm{i} t = 2\#_\mathrm{i} t + 1$.*

Démonstration. On précise d'abord les définitions : pour $f \in F$, $\#_\mathrm{f} f = 1$ et $\#_\mathrm{i} f = 0$; de plus, $\#_\mathrm{f}(v,g,d) = \#_\mathrm{f} g + \#_\mathrm{f} d$ et $\#_\mathrm{i}(v,g,d) = 1 + \#_\mathrm{i} g + \#_\mathrm{i} d$. On vérifie alors chacune des égalités directement par induction sur les arbres. □

1.3.3. Exercice. Proposez une classe Python pour les arbres binaires. ◇

La principale qualité des arbres binaires est que les sous-arbres sont adressés par des mots binaires. Une application particulièrement importante est de s'en servir comme représentation efficace pour des collections d'objets sur lesquels on dispose d'un ordre. On impose pour cela deux contraintes.

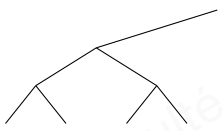
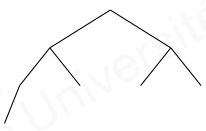
La première consiste à demander que les valeurs de l'arbre gauche (portées par les nœuds internes ou les feuilles) soient toutes plus petites que celles de l'arbre droit. Ainsi, on peut tester la présence d'un élément (ou décider où insérer un élément) en le comparant avec la valeur de la racine pour décider s'il faut aller dans le sous-arbre gauche ou droit. Un arbre qui satisfait cette contrainte est appelé *arbre binaire de recherche*.

1.3.4. Exercice. Donnez un algorithme pour rechercher un élément dans un arbre binaire de recherche, et vérifiez que sa complexité dans le pire cas est de l'ordre de la hauteur de l'arbre. ◇

C'est cette propriété qui justifie la deuxième contrainte : la hauteur de l'arbre doit être minimale relativement à sa taille.

1.3.5. Exercice (Relation entre taille et hauteur d'un arbre binaire)**.** Montrez que, pour tout arbre binaire t, on a $\mathrm{h}(t) \leq \#t < 2^{\mathrm{h}(t)}$. ◇

On obtient $\mathrm{h}(t) > \log_2(\#t)$. Dans le meilleur des cas, on a donc $\mathrm{h}(t) = \lceil \log_2(\#t+1) \rceil$. On vise donc une hauteur logarithmique en la taille. Énoncée de cette manière, la contrainte est globale mais pas très robuste. Voilà par exemple deux arbres de taille 8 et hauteur 4 :

 et  .

©L. Vaux Auclair, amU, 2024–2025

1.3.6. Définition (Arbres équilibrés)**.** Un arbre binaire est *équilibré* lorsque, pour tout nœud interne (v, g, d), $|\mathrm{h}(g) - \mathrm{h}(d)| \leq 1$.

1.3.7. Exercice (Taille des arbres équilibrés)**.** Notons T_h la taille minimale d'un arbre équilibré de hauteur h. On a $T_1 = 1$ et $T_2 = 2$.

1. Vérifiez qu'on a $T_{h+2} = 1 + T_{h+1} + T_h$ pour tout $h \geq 1$.

On note $(F_n)_{n\in\mathbb{N}}$ la suite de Fibonacci de premiers termes $F_0 = 0$ et $F_1 = 1$.

2. Vérifiez qu'on a $T_h = F_{h+2} - 1$ pour tout $h \geq 1$.
3. Déduisez-en une borne asymptotique logarithmique en n pour la hauteur des arbres équilibrés de taille n. ◇

La structure d'arbre binaire de recherche équilibré est ainsi classiquement utilisée pour représenter de manière efficace les ensembles finis d'objets. Tout l'enjeu est de maintenir cette structure lors des ajouts et suppressions d'éléments. Il y a de nombreuses variations sur ce thème, qu'on ne détaillera pas : arbres AVL,[5] arbres bicolores,[6] B-arbres,[7] *etc.* Celles-ci permettent de garantir la complexité dans le pire cas des opérations de recherche, ajout ou suppression d'un élément. Au-delà des ensembles, c'est aussi une solution pour les tableaux associatifs : il suffit de considérer des étiquettes de la forme `(clé,valeur)`, et de ne considérer que la clé dans les opérations sur les arbres. En pratique, ces structures sont utilisées dans les bases de données (par exemple pour les index de PostgreSQL[8]), dans les bibliothèques standard de certains langages de programmation (c'est le choix par défaut pour les tableaux associatifs de C++[9] et l'un des choix possibles pour ceux de Java[10]), et dans la quasi-totalité des systèmes de fichiers.

1.3.8. *En Python* (Tables de hachage pour les ensembles et les tableaux associatifs). Les tableaux associatifs de Python *n'utilisent pas les arbres équilibrés* ! Ils utilisent le concept de table de hachage[11] qu'on ne détaille pas mais qu'on peut décrire rapidement : il s'agit de construire un tableau `t`, de taille raisonnable, et de mettre chaque élément `x` à la case `hash(x)%len(t)` où `hash` est une fonction qui calcule un entier. Lorsque `x` est déjà dans le tableau et `y` est un nouvel élément à insérer, différent de `x` mais avec `hash(x)%len(t)==hash(y)%len(t)`, on utilise simplement la première case libre qui suit, ce qui casse la garantie d'un accès en temps constant. De plus, quand le tableau est plein, il faut l'agrandir, ce qui se fait en temps linéaire en `len(t)` en général.

Si les choses sont bien faites, toutes les opérations se font en temps constant *en moyenne*, mais le prix à payer est que, de temps en temps, certaines opérations se font en temps linéaire en la taille du tableau (à comparer avec le temps logarithmique garanti des arbres binaires de recherche équilibrés).

Outre les tableaux associatifs, c'est aussi la solution retenue pour les ensembles finis, de type `set`.[12] L'ensemble vide s'écrit `set()`. Les ensembles non vides peuvent s'écrire comme `{0,1,2}`, ou bien en appliquant `set` à un iterable : on a `{0,1,2}==set(range(3))`.

5. https://fr.wikipedia.org/wiki/Arbre_AVL[W].
6. https://fr.wikipedia.org/wiki/Arbre_bicolore[W].
7. https://fr.wikipedia.org/wiki/Arbre_B[W].
8. https://docs.postgresql.fr/17/indexes-types.html
9. https://cplusplus.com/reference/map/map/
10. https://docs.oracle.com/javase/8/docs/api/java/util/SortedMap.html.
11. https://fr.wikipedia.org/wiki/Table_de_hachage[W]
12. Voir la documentation complète : https://docs.python.org/fr/3.13/library/stdtypes.html#set.

©L. Vaux Auclair, amU, 2024–2025

2 Graphes

La notion de graphe est si centrale en mathématiques discrètes et en informatique qu'il en existe des variantes de toutes sortes : à chaque ouvrage, chaque article, sa définition, souvent partielle voire omise. On va commencer par une version très générale : on impose seulement que chaque arc relie exactement deux sommets. [13]

2.1 Graphes orientés

2.1.1. Définition (Graphe orienté)**.** Un *graphe orienté* est un quadruplet $G = (S, A, s, t)$ où :

— S et A sont des ensembles (généralement supposés dénombrables) : S est l'ensemble des *sommets*, A est l'ensemble des *arcs* ;

— s et t sont des fonctions $A \to S$: $s(a)$ est la *source* de l'arc a et $t(a)$ est son *but*.

On note alors $S = S_G$, $A = A_G$, *etc.* Une *boucle* de G est un arc $a \in A_G$ tel que $s(a) = t(a)$. Un *sous-graphe* de G est un graphe H tel que $S_H \subseteq S_G$, $A_H \subseteq A_G$ et, pour tout $a \in A_H$, $s_H(a) = s_G(a)$ et $t_H(a) = t_G(a)$. Un graphe G est dit *fini* quand S_G et A_G sont finis.

Le graphe présenté en début de chapitre est donc un sous-graphe fini et sans boucles du graphe de divisibilité sur $\mathbf{N}$, défini en posant $S = \mathbf{N}$, $A = \{(i, j) \in \mathbf{N}^2 \mid i \text{ divise } j\}$, $s((i, j)) = i$ et $t((i, j)) = j$. C'est un cas particulier de la construction suivante :

2.1.2. Définition (Graphe d'une relation binaire)**.** À toute relation binaire $R \subseteq X^2$ sur un ensemble X, on associe le graphe (X, R, s, t) défini en posant $s((i, j)) = i$ et $t((i, j)) = j$ pour tout $(i, j) \in R$.

Notez que dans ce cas, pour tout couple $(i, j) \in X^2$, il y a au plus un arc de i à j. C'est un graphe simple :

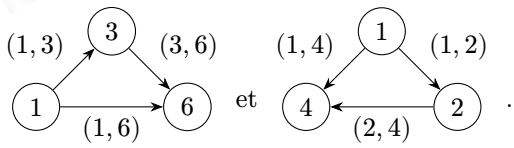
2.1.3. Définition (Graphe orienté simple)**.** Un graphe orienté G est dit *simple* si, pour tout couple de sommets $(i, j) \in S_G^2$ il y a au plus un arc $a \in A_G$ tel que $s(a) = i$ et $t(a) = j$. Autrement dit, G est simple lorsque la fonction $a \in A_G \mapsto (s(a), t(a)) \in S_G^2$ est injective.

Assez souvent, la position des sommets et la nature des arcs importe peu : c'est la relation entre arcs et sommets qui nous intéresse. C'est-à-dire qu'on considèrera parfois les graphes à isomorphisme près :

2.1.4. Définition (Morphisme de graphes)**.** Un *morphisme* du graphe G vers le graphe H est un couple de fonctions (f_S, f_A), avec $f_S : S_G \to S_H$ et $f_A : A_G \to A_H$, telles que $s(f_A(a)) = f_S(s(a))$ et $t(f_A(a)) = f_S(t(a))$ pour chaque $a \in A_G$. Ce morphisme est un *plongement* lorsque f_S et f_A sont injectives, et un *isomorphisme* lorsqu'elles sont bijectives.

2.1.5. Exercice (Graphes et morphismes)**.**

1. Vérifiez que les graphes suivants sont isomorphes :



2. Vérifiez que les plongements du graphe $(s) \xrightarrow{a} (t)$ vers un graphe G sont en bijection avec les arcs de G qui ne sont pas des boucles.

13. Sans cette contrainte, on parle d'*hypergraphe* : `https://fr.wikipedia.org/wiki/Hypergraphe`$^{\mathbb{W}}$.

©L. Vaux Auclair, amU, 2024–2025

3. Vérifiez qu'un graphe est simple ssi il est isomorphe au graphe d'une relation. De plus, dans ce cas, le graphe est sans boucle ssi la relation est irréflexive. ◇

Malheureusement, tester si deux graphes sont isomorphes est un problème compliqué. En effet, contrairement à ce qui se passe pour les arbres, où la racine est connue et où tout se fait par induction sur les sous-arbres, quand on veut mettre en correspondance deux graphes, on ne sait pas trop par où commencer. On ne sait pas vraiment faire mieux que tester toutes les bijections possibles : si on en trouve une qui est un morphisme, alors les graphes sont isomorphes ; sinon, ils ne le sont pas. À l'heure actuelle, la complexité du problème d'isomorphisme de graphes est inconnue : on ne connait pas de solution en temps polynomial en la taille du graphe, mais on ne sait pas non plus si le problème est NP-complet.

En général, les problèmes et les algorithmes sur les graphes s'expriment à partir de la structure locale : la relation entre sommets et arcs. Celle-ci mérite encore un peu de vocabulaire.

2.1.6. Définition (Adjacence, degré, *etc.*)**.** On dit qu'un sommet x et un arc a sont *adjacents* si $x = s(a)$ ou $x = t(a)$. Les *arcs sortants* (resp. les *arcs entrants*) d'un sommet x sont les arcs a tels que $x = s(a)$ (resp. $x = t(a)$). Le *degré entrant* (resp. *degré sortant*) d'un sommet est le nombre de ses arcs entrants (resp. sortants), et son *degré total* est la somme de ses degrés entrant et sortant. Un *successeur* (resp. *prédécesseur*) du sommet x, c'est un sommet y tel qu'il existe un arc a avec $s(a) = x$ et $t(a) = y$ (resp. $t(a) = x$ et $s(a) = y$). On dit que deux sommets x et y sont *voisins* si x est un successeur ou un prédécesseur de y.

2.1.7. *En Python* 🐍. Pour représenter un graphe fini, on doit représenter les deux ensembles S et A, et les fonctions s et t. On veut également pouvoir accéder de manière efficace aux informations d'adjacence : étant donné un sommet, on veut connaître l'ensemble de ses arcs entrants et sortants. La classe définie en figure 2 assure que toutes ces informations sont directement disponibles *via* des dictionnaires.

Le graphe

s'écrit alors `GrapheOriente({0,1,2},{"a":(0,1),"b":(0,1),"c":(2,2)})`.

On peut par exemple calculer le degré sortant d'un sommet `x` dans un graphe `G` avec `len(G.sortants(x))`.

2.1.8. Exercice.

1. Ajoutez à la classe `GrapheOriente` des méthodes `successeurs(self,x)`, `predecesseurs(self,x)` et `voisins(self,x)`, qui retournent des itérables énumérant respectivement l'ensemble des successeurs du sommet `x`, celui de ses prédécesseurs et celui de ses voisins.
2. Ajoutez des méthodes `supprime_arc(self,a)`, et `supprime_sommet(self,x)`, avec le sens évident. Attention : quand on supprime un sommet, il faut d'abord supprimer tous les arcs adjacents ; et quand on supprime un arc, il faut mettre à jour les informations sur les arcs entrants et sortants de sa source et de son but. ◇

2.2 Chemins

Une notion cruciale en théorie des graphes est celle de chemin :

2.2.1. Définition (Chemins orientés)**.** Un *chemin orienté* (on dira simplement *chemin*) dans un graphe orienté G est une suite finie $\gamma = (x_0, a_1, x_1, a_2, \dots, x_{l-1}, a_l, x_l)$ telle que les x_i sont des

©L. Vaux Auclair, amU, 2024–2025

```python
class GrapheOriente:

    """Une classe pour les graphes orientés.

    L'attribut `sommets` est un dictionnaire dont les clés sont les sommets.
    Pour chaque sommet `x`, `sommets[x]` est un couple `(entrants,sortants)`
    où `entrants` est 'lensembles des arcs entrants de `x`, et
    `sortants` 'lensemble des arcs sortants.

    L'attribut `arcs` est un dictionnaire dont les clés sont les arcs.
    Pour chaque arc `a`, `arcs[a]` est un couple `(source,but)`, où
    `source` et `but` sont les sommets source et but de `a`.
    """

    def __init__(self, S=set(), A={}):

        """Crée un nouveau graphe à partir d'un ensemble de sommets `S`
        (vide par défaut), et d'un ensemble d'arcs `A` (vide par défaut)
        chacun muni d'un sommet source et d'un sommet but.

        Formellement : `S` est un itérable et `A` est un tableau clé-valeur:
        les clés de `A` sont les arcs, et la valeur associée à un arc est
        le couple formé de son sommet source et son sommet but."""

        self.sommets = {}
        self.arcs = {}
        for x in S:
            self.ajoute_sommet(x)
        for a in A:
            s,t = A[a]
            self.ajoute_arc(a, s, t)

    def ajoute_sommet(self, x):
        if x in self.sommets:
            raise ValueError("le sommet existe déjà")
        self.sommets[x] = (set(), set())

    def ajoute_arc(self, a, s, t):
        if a in self.arcs:
            raise ValueError("l'arc existe déjà")
        self.arcs[a] = (s,t)
        self.sommets[s][1].add(a) # ajoute `a` aux arcs sortants de `s`
        self.sommets[t][0].add(a) # ajoute `a` aux arcs entrants de `t`

    def source(self, a):
        return self.arcs[a][0]

    def but(self, a):
        return self.arcs[a][1]

    def entrants(self, x):
        return self.sommets[x][0]

    def sortants(self, x):
        return self.sommets[x][1]
```

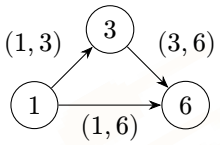
FIGURE 2 – Une classe Python pour représenter les arbres ordonnés (fichier `grapheoriente_base.py` joint au cours).

©L. Vaux Auclair, amU, 2024–2025

sommets de G, les a_j sont des arêtes de G, et on a $x_{i-1} = s(a_i)$ et $x_i = t(a_i)$ pour $1 \leq i \leq l$. La *source* de γ est $s(\gamma) = x_0$, son *but* est $t(\gamma) = x_l$, et sa *longueur* est $|\gamma| = l$. Si γ et δ sont deux chemins *composables*, c'est-à-dire avec $t(\gamma) = s(\delta)$, on note $\gamma \cdot \delta$ la concaténation de ces chemins, de source $s(\gamma)$ et de but $t(\delta)$. On dit qu'un sommet y est *accessible* depuis un sommet x s'il existe un chemin de source x et de but y.

De manière équivalente, un chemin γ est entièrement caractérisé par sa source $s(\gamma)$ et la suite de ses arcs $(a_1, a_2, \dots)$, qui vérifie $s(a_1) = s(\gamma)$ (si $l > 0$) et $t(a_i) = s(a_{i+1})$ pour $1 \leq i \leq |\gamma|$. En particulier, chaque sommet x de G est associé à un *chemin vide* ε_x, de longueur 0 et de source et but x.

2.2.2. *Exemple.* Dans le graphe



il y a 3 chemins vides, 3 chemins de longueur 1, un unique chemin de longueur 2, et aucun chemin plus long. Par contre dans un graphe réduit à une boucle 

, il y exactement un chemin de chaque longueur.

2.2.3. Exercice. À tout ensemble fini X on associe un graphe X^* avec $S_{X^*} = \{0\}$ et $A_{X^*} = X$ (et, forcément, $s_{X^*}(x) = t_{X^*}(x) = 0$ pour tout $x \in X$: il y a une boucle pour chaque élément de X). Vérifiez que les chemins de X^* sont exactement les mots sur X (d'où la notation), et que la concaténation des chemins revient à la concaténation des mots. ◇

On s'intéressera souvent à des chemins particuliers :

2.2.4. Définition. Un chemin γ est dit *fermé* si c'est un chemin fini dont le but et la source coïncident : $s(\gamma) = t(\gamma)$. Un chemin est dit *simple* si ses sommets sont deux-à-deux distincts, sauf éventuellement sa source et son but (et alors il fermé). Un *cycle* est un chemin fermé simple non vide.

Dès qu'un graphe admet un chemin fermé γ non vide, il admet des chemins de longueur arbitraire (et même un chemin infini, qu'on définit comme ci-dessus, mais avec une suite infinie d'arcs). Par contre, la longueur d'un chemin simple est bornée par le nombre de sommets du graphe.

2.2.5. Exercice.

1. Vérifiez qu'un chemin qui utilise deux fois le même arc ne peut pas être simple.
2. Montrez que si un sommet y est accessible depuis un sommet x, alors il existe un chemin simple de source x et de but y.
3. Déduisez de la question précédente que si un graphe orienté contient un chemin fermé non vide, alors il contient un cycle. ◇

2.3 Parcours de graphes

Très souvent, les algorithmes sur les graphes impliquent un parcours consistant à explorer le graphe depuis un sommet donné. Ce parcours peut se faire suivant au moins deux modes : en profondeur ou en largeur.

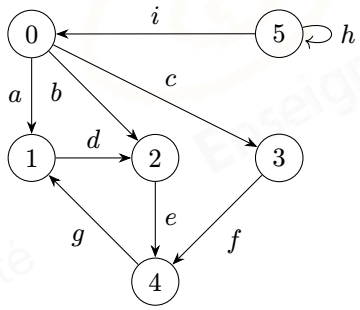
©L. Vaux Auclair, amU, 2024–2025

Dans un *parcours en profondeur*, pour chaque successeur y du nœud courant x, on effectue récursivement un parcours en partant de y, suivant un algorithme ressemblant à : [14]

```
def parcours_profondeur(G, x, visites, aller, retour, trace, termine):
    for a in G.sortants(x):
        if a not in visites:
            visites.add(a)
            trace = aller(G, a, trace)
            trace = parcours_profondeur(G, y, visites, aller, retour, trace)
            trace = retour(G, a, trace)
    return trace
```

où `aller` et `retour` sont les opérations à effectuer respectivement avant et après le traitement des successeurs, `visites` est l'ensemble des arcs déjà suivis, et `trace` représente la propagation de l'information au fil de l'exécution. La mémorisation des arcs (ou sommets) déjà visités est cruciale pour éviter les boucles infinies en présence de cycles.

Par exemple dans un graphe G :



lors d'un parcours démarrant au sommet 0, on aura la trace d'exécution :

étape	traitement	sommet	arc	arcs visités
1	`aller`	0	a	a
2	`aller`	1	d	a, d
3	`aller`	2	e	a, d, e
4	`aller`	4	g	a, d, e, g
5	`retour`	4	g	a, d, e, g
6	`retour`	2	e	a, d, e, g
7	`retour`	1	d	a, d, e, g
8	`retour`	0	a	a, d, e, g
9	`aller`	0	b	a, d, e, g, b
10	`retour`	0	b	a, d, e, g, b
11	`aller`	0	c	a, d, e, g, b, c
12	`aller`	3	f	a, d, e, g, b, c, f
13	`retour`	3	f	a, d, e, g, b, c, f
14	`retour`	0	c	a, d, e, g, b, c, f

où chaque arc est emprunté deux fois (une fois dans le sens aller, une fois dans le sens retour). Par exemple, à l'étape 4, l'arc g mène au sommet 1 dont on a déjà visité tous les arcs sortants

14. Ce n'est bien sûr qu'un exemple générique, plutôt qu'une fonction à utiliser de manière systématique. La plupart des algorithmes de parcours ont des comportements plus spécifiques : on peut ne s'intéresser qu'aux sommets déjà visités plutôt qu'aux arcs ; on peut vouloir s'arrêter dès qu'une certaine information a été trouvée plutôt que de parcourir tous les sommets accessibles ; *etc.*

©L. Vaux Auclair, amU, 2024–2025

(il n'y en a qu'un), donc on rebrousse chemin ; et à l'étape 14, on revient au sommet 0 en ayant exploré récursivement tous ses arcs sortants, ce qui termine le parcours. Notez que ni l'arc h ni l'arc i ne sont visités.

2.3.1. Exercice. Ajoutez à la classe `GrapheOriente` de la figure 2 une méthode `accessibles(self,x)` qui renvoie un itérable dont les éléments sont les sommets accessibles depuis `x`. C'est facile avec un parcours en profondeur. ◇

Dans un *parcours en largeur*, on traite immédiatement les arcs sortants (ou les sommets successeurs), avant d'explorer le reste du graphe. Ça demande de maintenir une mémoire des sommets dont la visite est à venir, sous la forme d'une file. L'algorithme de parcours est donc de la forme suivante : [15]

```
def parcours_largeur(G, x, traitement, trace):
    prochains = File()
    marques = set()
    prochains.push(x)
    marques.add(x)
    while len(prochains):
        x = prochains.pop()
        trace = traitement(G, x, trace)
        for y in G.successeurs(x):
            if y not in marques:
                prochains.push(y)
                marques.add(y)
    return trace
```

où `traitement` est le traitement à effectuer pour chaque sommet et `trace` joue le même rôle que précédemment. L'algorithme utilise à la fois une file `prochains` (on suppose disposer d'une classe `File` pour ça) pour mémoriser les sommets restant à visiter (en respectant l'ordre de visite) et un ensemble `marques` qui permet de tester efficacement si on a déjà visité ou programmé la visite d'un sommet.

Par exemple avec le graphe G ci-dessus, si on démarre un parcours en largeur à partir du sommet 0, on obtient l'exécution :

étape	sommet courant	file des sommets à visiter	sommets marqués
1		0	0
2	0	1,2,3	0,1,2,3
3	1	2,3	0,1,2,3
4	2	3,4	0,1,2,3,4
5	3	4	0,1,2,3,4
6	4		0,1,2,3,4

les sommets étant visités dans l'ordre $0, 1, 2, 3, 4$ et non $0, 1, 2, 4, 3$ comme précédemment.

2.3.2. Exercice. Ajoutez à la classe `GrapheOriente` de la figure 2 une méthode `distance(self,x,y)` qui renvoie la longueur minimale d'un chemin de source `x` et de but `y` (et échoue ou renvoie `None` si `y` n'est pas accessible à partir de `x`). C'est facile avec un parcours en largeur. ◇

Un problème classique est de rechercher un chemin le plus court possible en associant une distance (un nombre positif) à chaque arc. La solution classique de ce problème est l'algorithme de Dijkstra, qui n'est rien d'autre qu'une généralisation du précédent. [16]

15. Là encore, ce n'est qu'un exemple générique. Ici on choisit de se concentrer sur les sommets plutôt que sur les arcs.

16. `https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra`[W].

©L. Vaux Auclair, amU, 2024–2025

2.3.3. Exercice (*facultatif*)**.** Réalisez l'algorithme de Dijkstra comme une méthode `dijkstra(self,x,y,poids)` : en supposant que `poids` est une fonction telle que `poids(a)` retourne un nombre positif ou nul pour chaque arc `a`, `dijkstra(self,x,y,poids)` retourne un chemin du sommet `x` au sommet `y`, qui minimise la somme des poids des arcs (ou soulève une exception s'il n'y a pas de chemin). ◇

2.4 Graphes acycliques et circuits booléens

À ce stade, vous avez sans doute constaté que la présence de cycles complique sensiblement l'algorithmique sur les graphes : il faut chaque fois prendre soin de mémoriser par quels sommets on est déjà passés, sans quoi on risque de boucler indéfiniment. Les graphes sans cycles forment naturellement un classe de graphes dont le rôle est important dans une grande variété d'applications.

2.4.1. Définition (Graphes acycliques)**.** Un graphe orienté sans cycle (c'est-à-dire sans chemin fermé non vide) est appelé *graphe acyclique*.

2.4.2. Exercice.

1. Vérifiez que dans un graphe acyclique, il existe au moins un sommet de degré entrant nul, et même mieux : tout sommet est accessible depuis un sommet de degré entrant nul.
2. Proposez un algorithme pour tester si un graphe est acyclique. Quelle est sa complexité ?
3. Réalisez cet algorithme en ajoutant une méthode `est_acyclique(self)` qui renvoie un booléen. ◇

Un exemple d'application est la modélisation théorique de circuits de calcul, par exemple les circuits booléens : [17]

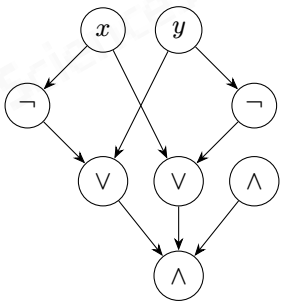
2.4.3. Définition. Soit E un ensemble fini. Un *circuit booléen* d'entrées E est un graphe acyclique C muni d'un étiquetage des sommets par les éléments de $E \cup \{\wedge, \vee, \neg\}$ (l'union étant supposée disjointe), tel que :

— les sommets munis d'une étiquette dans E sont de degré entrant nul : ce sont les *entrées du circuit* ;

— les sommets munis d'une étiquette $\neg$ sont de degré entrant 1.

Les sommets de degré sortant nul sont les *sorties du circuit*.

2.4.4. *Exemple.* Voici un circuit booléen à deux entrées et une sortie.



17. On adapte la définition du livre de Sylvain Périfel, *Complexité algorithmique*, paru chez Ellipses, et disponible directement sur son site `https://www.irif.fr/users/sperifel/livre_complexite`.

©L. Vaux Auclair, amU, 2024–2025

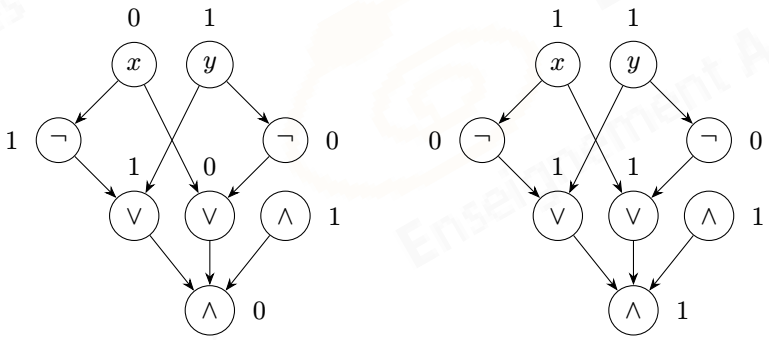
On calcule avec un circuit booléen en affectant des valeurs à ses entrées, et en calculant les valeurs obtenues pour ses sorties. Pour que ce calcul ait du sens, il faut qu'il soit défini de manière univoque. Formellement calculer revient à appliquer l'opération prévue à chaque sommet :

2.4.5. Définition. Une *solution* d'un circuit booléen C est une fonction $\varphi : S_C \to \mathbf{B}$ telle que :

— pour tout sommet x étiqueté par $\neg$, si y est l'unique prédécesseur de x, $\varphi(x) = \neg\varphi(y)$;

— pour tout sommet x étiqueté par $\wedge$ (resp. $\vee$), si $\{y_1, \dots, y_k\}$ est l'ensemble des prédécesseurs de x, alors $\varphi(x) = \min\{\varphi(y_1), \dots, \varphi(y_k)\}$ (resp. $\varphi(x) = \max\{\varphi(y_1), \dots, \varphi(y_k)\}$) — si $k = 0$, $\varphi(x) = 1$ (resp. $\varphi(x) = 0$).

2.4.6. *Exemple.* Voilà deux solutions pour le circuit de l'exemple 2.4.4 (on écrit la valeur associée à un sommet à proximité de ce sommet) :



2.4.7. Exercice. Montrez que pour tout circuit booléen C d'entrées E et toute fonction $f : E \to \mathbf{B}$, il existe une unique solution φ de C telle que $\varphi(x) = f(e)$ chaque fois que x est une entrée étiquetée par e. ◇

2.4.8. Définition. Soit un circuit C à k entrées $x_1, \dots, x_k$ et n sorties $y_1, \dots, y_n$. La *fonction calculée par* C est la fonction booléenne $f : \mathbf{B}^k \to \mathbf{B}^n$ telle que, pour tout mot $u \in \mathbf{B}^k$, l'unique solution φ de C telle que $\varphi(x_i) = u[i]$ pour $i \in [\![k]\!]$ vérifie $\varphi(y_j) = f(u)[j]$ pour $j \in [\![n]\!]$.

2.4.9. Exercice. Vérifiez que le circuit de l'exemple 2.4.4 calcule l'équivalence : $xy \mapsto x \Leftrightarrow y$. ◇

2.5 Les arbres comme graphes

À tout arbre ordonné t (au sens de la section précédente), on peut associer le graphe $G(t)$ de sa relation de filiation : les sommets de $G(t)$ sont les adresses de t, et on a un arc de source p et de but pi pour toute adresse $p \in \mathrm{adr}(t)$ et tout $i \in [\![\mathrm{d}(t[p])]\!]$.

2.5.1. Proposition. *Le graphe $G(t)$ d'un arbre t est acyclique.*

Démonstration. C'est direct par induction sur t. En effet, si t est réduit à une feuille, il n'y a pas de chemin non vide dans $G(t)$. Sinon, considérons un chemin fermé non vidé γ dans $G(t)$. Ce chemin ne peut pas passer par la racine (d'adresse ε), puisqu'elle n'a que des arcs sortants, donc γ est entièrement dans un des enfants $t[i]$ et donc dans $G(t[i])$: par hypothèse d'induction, on obtient une contradiction. □

On peut capturer la structure graphique correspondante de la manière suivante :

©L. Vaux Auclair, amU, 2024–2025

2.5.2. Définition (Forêts et arbres)**.** On dit qu'un graphe acyclique est une *forêt*, si tout sommet est de degré entrant au plus 1. Une forêt est un *arbre* (disons un *arbre-graphe* pour éviter la confusion avec la section précédente) si de plus elle admet une unique *racine*, c'est-à-dire un unique sommet de degré entrant 0.

Dans une forêt, chaque sommet a au plus un prédécesseur (son parent) ; et dans un arbre, seule la racine n'a pas de prédécesseur.

2.5.3. Proposition. *Tout arbre-graphe H est isomorphe au graphe $G(t)$ d'un arbre t.*

Démonstration. On raisonne par récurrence sur le nombre de sommets de H : si H n'a pas de sommets, il ne peut pas être un arbre puisqu'il n'a pas de racine. Sinon, soient x la racine de H, et $y_0, \dots, y_{n-1}$ ses successeurs. Par construction, pour chaque $i \in [\![n]\!]$, le sous-graphe H_i de H constitué des sommets accessibles depuis y_i est un arbre-graphe, avec strictement moins de sommets de que H : par hypothèse de récurrence, on obtient un arbre t_i tel que H_i soit isomorphe à $G(t_i)$, *via* une bijection φ_i entre les sommets de H_i et les adresses de t_i. On construit alors $t = (t_0, \dots, t_{n-1})$, et on pose $\varphi(x) = \varepsilon$ et, pour $i \in [\![n]\!]$, $\varphi(y) = i\varphi_i(y)$ pour chaque sommet y de H_i. On vérifie alors que φ est un isomorphisme (il suffit de le définir sur les sommets parce qu'un arbre-graphe est évidemment un graphe simple). □

Notez toutefois que, l'ordre des enfants étant perdu, deux arbres distincts peuvent correspondre à des graphes-abres isomorphes : c'est précisément pour cette raison que les théoriciens des graphes appellent *arbre ordonné* un arbre-graphe muni d'un ordre total sur les successeurs de chaque sommet. On a alors une notion équivalente (*à isomorphisme près*) à celle de la section précédente : ici la notion d'isomorphisme est moins dangereuse que pour un graphe général, car tester l'isomorphisme d'arbres-graphes ordonnés se fait en temps linéaire en le nombre de sommets.

2.5.4. Exercice. Proposez un algorithme pour tester si un graphe est un arbre. Quelle est sa complexité ? ◇

2.6 Graphes symétriques et chemins non orientés

Très souvent, on s'intéresse à des graphes pour lesquels l'orientation des arcs n'est pas pertinente : on veut seulement savoir si un arc relie deux sommets, sans que l'un soit distingué comme sa source et l'autre comme son but. C'est la notion de graphe non-orienté, ou graphe symétrique :

2.6.1. Définition (Graphe symétrique)**.** Un *graphe symétrique* est un triplet $G = (S, A, t)$ où :

- S et A sont des ensembles (généralement supposés dénombrables) : S est l'ensemble des *sommets*, A est l'ensemble des *arcs* ;
- t est une fonction $A \to \mathcal{P}_{1,2}(S)$ où $\mathcal{P}_{1,2}(S) = \{\{x, y\} \mid x, y \in S\}$ est l'ensemble des parties de S à 1 ou 2 éléments : $t(a)$ est l'ensemble des *extrémités* de l'arc a.

Une *boucle* de G est un arc $a \in A_G$ tel que $\#t(a) = 1$. Un *sous-graphe* de G est un graphe H tel que $S_H \subseteq S_G$, $A_H \subseteq A_G$ et, pour tout $a \in A_H$, $t_H(a) = t_G(a)$. Un graphe G est dit *fini* quand S_G et A_G sont finis.

C'est donc exactement la même définition que pour les graphes orientés, sauf qu'au lieu d'associer un couple $(x, y) = (s(a), t(a))$ de sommets adjacents à chaque arc, on associe un ensemble $\{x, y\} = t(a)$. À chaque graphe orienté G, on peut associer un graphe symétrique $\widetilde{G}$, avec les mêmes ensembles de sommets et d'arcs, en posant $t_{\widetilde{G}}(a) = \{s_G(a), t_G(a)\}$. Il est alors direct que tout graphe symétrique est de la forme $\widetilde{G}$ où G est un graphe orienté.

©L. Vaux Auclair, amU, 2024–2025

On peut reproduire la plupart des définitions dans un cadre non-orienté. Par exemple un *voisin* d'un sommet, c'est un sommet adjacent à un arc commun ; et le *degré* d'un sommet, c'est le nombre de ses voisins. On a aussi une notion non orientée de chemin :

2.6.2. Définition (Chemins dans un graphe symétrique)**.** Un *chemin dans un graphe symétrique* G est une suite finie $\gamma = (x_0, a_1, x_1, a_2, \dots, x_{l-1}, a_l, x_l)$ telle que les x_i sont des sommets de G, les a_j sont des arêtes de G, et on a $\{x_{i-1}, x_i\} = t(a_i)$ pour $1 \leq i \leq l$.

Suivre un chemin dans $\widetilde{G}$, c'est s'autoriser à suivre les arcs de G dans le sens prescrit par l'orientation (de la source vers le but) mais aussi dans le sens inverse.

Dans la plupart des cas, on peut donc ramener l'étude des graphes symétriques à celle des graphes orientés, quitte à considérer des chemins non orientés :

2.6.3. Définition (Chemins non orientés)**.** Un *chemin non orienté* (on dit aussi une *chaîne*) dans un graphe orienté G est une suite finie $\gamma = (x_0, a_1, x_1, a_2, \dots, x_{l-1}, a_l, x_l)$ telle que les x_i sont des sommets de G, les a_j sont des arêtes de G, et on a $\{x_{i-1}, x_i\} = \{s(a_i), t(a_i)\}$ pour $1 \leq i \leq l$.

On a alors une correspondance bijective immédiate entre les chemins de $\widetilde{G}$ et les chaînes de G. La seule subtilité à considérer est que la notion de chemin simple (et donc de cycle) change : dans un graphe orienté contenant au moins un arc a, on a toujours une chaîne fermée $s(a), a, t(a), a, s(a)$, qu'on n'a pas envie de considérer comme un cycle. Or cette chaîne utilise deux fois l'arc a : dans le cas orienté, il suffit d'exiger qu'un chemin simple ne passe pas deux fois par le même sommet (sauf, peut-être, à sa source et son but), pour éviter cette situation (voir l'exercice 2.2.5). Ici, il faut l'imposer directement.

2.6.4. Définition. Un chemin γ dans un graphe symétrique est dit *fermé* si son but et sa source coïncident. Un chemin est *simple* si ses sommets sont deux-à-deux distincts, sauf éventuellement sa source et son but (et alors elle est fermée), et si ses arcs sont également deux-à-deux distincts. Un *cycle* dans un graphe symétrique est un chemin simple fermé non vide.

2.6.5. Exercice. Vérifiez qu'un chemin dans un graphe symétrique simple ssi :

— ses sommets sont deux-à-deux distincts, sauf éventuellement sa source et son but,

— et il n'est pas de la forme x, a, y, a, x avec $t(a) = \{x, y\}$.

Autrement dit, l'exemple précédent est le seul cas où il faut faire attention aux arcs traversés. ◇

Parmi les graphes symétriques, on s'intéresse particulièrement aux graphes connexes :

2.6.6. Définition. Un graphe symétrique est dit *connexe* si, pour tous sommets x et y, y est accessible depuis x, c'est-à-dire qu'il existe un chemin de source x et de but y. Un graphe connexe est dit *1-connexe* si de plus il est non vide. La *composante connexe* d'un sommet x dans un graphe symétrique est l'ensemble des sommets accessibles depuis y.

La relation d'accessibilité dans un graphe symétrique est évidemment une relation d'équivalence, et la composante connexe d'un sommet n'est rien d'autre que sa classe d'équivalence pour cette relation. Ces notions portant sur les graphes symétriques se traduisent directement sur les graphes orientés : par exemple la composante connexe d'un sommet dans un graphe orienté G, c'est sa composante connexe dans $\widetilde{G}$.

2.6.7. Exercice. Proposez un algorithme pour calculer la composante connexe d'un sommet dans un graphe. ◇

Suivant les textes, les théoriciens des graphes appellent *arbre* n'importe quel graphe symétrique acyclique et 1-connexe.

2.6.8. Exercice. Soit G un graphe symétrique acyclique et 1-connexe.

©L. Vaux Auclair, amU, 2024–2025

1. Montrez que G contient au moins un sommet de degré 1.

2. Montrez que le nombre de sommets de G est égal au nombre d'arcs de G, plus 1.

3. Montrez que $G = \widetilde{H}$ avec H un arbre-graphe. ◇

©L. Vaux Auclair, amU, 2024–2025