Graphes et outils logiques

Thibaut Balabonski @ Université Paris-Sud

Deuxième partie, 31 janvier 2020

Table des matières

3 Relations d'équivalence, classes, composantes connexes (06/02)

3.1 Relations d'équivalence

On revient sur les relations binaires sur un seul ensemble E, c'est-à-dire les relations sur $E \times E$. Ces relations sont dites ***homogènes***.

Une relation binaire homogène $\mathscr{R}$ sur un ensemble E est :

— ***réflexive*** si tout élément est en relation avec lui-même

$$\forall x \in E,\ x \mathscr{R} x$$

— ***symétrique*** si la relation ne dépend pas de l'ordre des éléments

$$\forall x_1, x_2 \in E,\ x_1 \mathscr{R} x_2 \Longrightarrow x_2 \mathscr{R} x_1$$

(alternativement : $\mathscr{R} = \mathscr{R}^{-1}$)

— ***anti-symétrique*** si deux éléments différents ne peuvent être en relation dans les deux sens à la fois

$$\forall x_1, x_2 \in E,\ (x_1 \mathscr{R} x_2 \wedge x_2 \mathscr{R} x_1) \Longrightarrow x_1 = x_2$$

— ***transitive*** si la relation se propage

$$\forall x_1, x_2, x_3 \in E,\ (x_1 \mathscr{R} x_2 \wedge x_2 \mathscr{R} x_3) \Longrightarrow x_1 \mathscr{R} x_3$$

Une ***relation d'équivalence*** est une relation réflexive, symétrique et transitive. La notion de relation d'équivalence peut être vue comme une généralisation de la notion d'égalité.

Exercices

— L'égalité sur $\mathbb{N}$ est une relation d'équivalence.

— La relation « est isomorphe à » sur les graphes est une relation d'équivalence.

3.2 Classes d'équivalence

Étant donnés un ensemble E, une relation d'équivalence $\sim$ sur E et un élément $e \in E$, la ***classe d'équivalence*** de e, notée $[e]$, est l'ensemble des éléments de E équivalent à e :

$$[e] = \{e' \in E \mid e \sim e'\}$$

L'élément e est appelé un ***représentant*** de la classe d'équivalence $[e]$.

L'ensemble $\mathscr{C} = \{C_1, C_2, \ldots\}$ des classes d'équivalence d'une relation d'équivalence $\sim$ sur E forme une ***partition*** de E, c'est-à-dire un ensemble de parties de E qui couvrent tout E tout en étant deux à deux disjointes.

$$E \subseteq C_1 \cup C_2 \cup \ldots \quad \wedge \quad \forall i, j,\ C_i \cap C_j = \emptyset$$

La partie $E \subseteq C_1 \cup C_2 \cup \ldots$ est évidente car tout élément e appartient à sa propre classe d'équivalence ($e \in [e]$).

Caractérisation alternative de la partition Tous deux ensembles $[e_1]$ et $[e_2]$ sont soit disjoints soit égaux.

$$\forall e_1, e_2 \in E,\ [e_1] \cap [e_2] = \emptyset \ \vee \ [e_1] = [e_2]$$

Pour la preuve de cette dernière propriété on commence par le lemme suivant :
Lemme : deux éléments équivalents définissent la même classe.

$$\forall e_1, e_2 \in E,\ e_1 \sim e_2 \Longrightarrow [e_1] = [e_2]$$

Preuve du lemme.
Soient e_1 et e_2 deux éléments de E tels que $e_1 \sim e_2$. Montrons que $[e_1] \subseteq [e_2]$.
Soit $x \in [e_1]$. Par définition $e_1 \sim x$, d'où par symétrie $x \sim e_1$ et par transitivité $x \sim e_2$. Ainsi par symétrie à nouveau $e_2 \sim x$, d'où par définition $x \in [e_2]$.
On montrerait de même que $[e_2] \subseteq [e_1]$, donc $[e_1] = [e_2]$.

Preuve de la propriété : $\forall e_1, e_2 \in E,\ [e_1] \cap [e_2] = \emptyset \ \vee \ [e_1] = [e_2]$.
Soient e_1 et e_2 deux éléments de E.

- Si $[e_1] \cap [e_2] = \emptyset$, alors la conclusion est immédiate.
- Sinon $[e_1] \cap [e_2] \neq \emptyset$, et il existe $e \in [e_1] \cap [e_2]$. Par définition de l'intersection $e \in [e_1]$ et $e \in [e_2]$, d'où par définition $e_1 \sim e$ et $e_2 \sim e$. Par symétrie on a donc $e \sim e_2$ et par transitivité $e_1 \sim e_2$. Alors le lemme permet de conclure $[e_1] = [e_2]$.

Caractérisation alternative des classes d'équivalence Étant donnés un ensemble E et une relation d'équivalence $\sim$ sur E, les classes d'équivalence de $\sim$ sont les ensembles maximaux (pour l'inclusion) d'éléments de E deux à deux équivalents.
Autrement dit, pour tout élément $e \in E$ on a :

- tous les éléments de $[e]$ sont deux à deux en relation, et
- tout ensemble C d'éléments deux à deux en relation tel que $[e] \subseteq C$ vérifie $C = [e]$.

(et réciproquement, tout ensemble C ayant ces propriétés est la classe d'équivalence d'un élément de e).
Preuve.

- Soient $e_1, e_2 \in [e]$. Par définition $e \sim e_1$ et $e \sim e_2$, par symétrie $e_1 \sim e$ et par transitivité $e_1 \sim e_2$.
- Soit $C \subseteq E$ tel que $[e] \subseteq C$ et $\forall c_1, c_2 \in C,\ c_1 \sim c_2$.
  Montrons que $C \subseteq [e]$ *:* Soit $e' \in C$, comme $e \in [e]$ et $[e] \subseteq C$ on déduit que $e \in C$. Donc par hypothèse $e \sim e'$, et par définition $e' \in [e]$, donc $C \subseteq [e]$.
  Comme de plus par hypothèse $[e] \subseteq C$, on déduit $C = [e]$.

Exercice

- Soit $G = (S, A, \epsilon)$ un graphe et $\sim$ la relation définie par $s_1 \sim s_2$ si et seulement si il existe des chemins de s_1 à s_2 et de s_2 à s_1. La relation $\sim$ est une relation d'équivalence, dont les classes d'équivalence sont les composantes fortement connexes de G.

3.3 Fonctions compatibles

Étant donné un E muni d'une relation d'équivalence $\sim$, une fonction $f : E \longrightarrow F$ est ***compatible*** avec $\sim$ si elle donne le même résultat pour des

entrées équivalentes :

$$\forall e_1, e_2 \in E,\ e_1 \sim e_2 \Longrightarrow f(e_1) = f(e_2)$$

Quelques conséquences :

- Du point de vue de f, l'équivalence $\sim$ est comme une égalité.
- Chaque classe d'équivalence a une unique image.
- Il suffit de caractériser f sur un représentant quelconque de chaque classe d'équivalence pour l'avoir définie totalement.

Espace quotient On appelle ***espace quotient*** d'un ensemble E par une relation d'équivalence $\sim$ sur E l'espace obtenu en considérant comme égaux tous les éléments de chaque classe d'équivalence.

Alternativement, cet espace quotient $E_{/\sim}$ peut être défini comme l'ensemble dont les éléments sont les classes d'équivalences de E.

Une fonction $f : E \longrightarrow F$ compatible avec $\sim$ définit une fonction $f_\sim : E_{/\sim} \longrightarrow F$, qui à une classe d'équivalence $C \in E_{/\sim}$ associe l'unique image des éléments de cette classe.

3.4 Structures de données et classes d'équivalence

Il est courant qu'un algorithme soit conçu à l'aide de structures de haut niveau dont la représentation en mémoire n'est pas unique. On se retrouve alors dans un cas pratique de classes d'équivalences, issues d'une relation d'équivalence « représenter la même structure abstraite » sur l'ensemble des structures concrètes. Les programmes doivent alors être compatibles avec cette représentation non unique des données.

Exemple : représenter des ensembles Supposons que l'on souhaite manipuler des ensembles d'entiers. Une réalisation rudimentaire consiste à utiliser un tableau contenant tous les éléments de l'ensemble représenté. Ainsi l'ensemble à quatre éléments $\{2,3,5,7\}$ peut être représenté par le tableau `[2; 3; 5; 7]`. Cependant, d'autres tableaux représentent ce même ensemble, comme `[3; 5; 2; 7]` ou encore `[2; 3; 5; 3; 7; 7]`.

Dans la conception et la réalisation d'un algorithme manipulant de tels ensembles on travaille donc sur deux niveaux :

- au niveau abstrait d'un algorithme, avec la notion mathématique d'ensemble,
- au niveau concret d'un programme, avec les tableaux d'entiers du langage utilisé.

L'ensemble des tableaux concrets est muni d'une relation d'équivalence « représenter le même ensemble mathématique », dont les classes d'équivalence sont les ensembles abstraits manipulés par l'algorithme.

Les fonctions écrites dans le programme doivent *a priori* être compatibles avec cette relation d'équivalence. Une fonction `appartient(elt, ens)` de test d'appartenance d'un élément `elt` à un ensemble `ens` doit renvoyer le même résultat quel que soit le tableau donné en argument représentant un ensemble mathématique donné. Une fonction `union(ens_1, ens_2)` d'union de deux ensembles doit, quels que soient les deux représentants de deux ensembles donnés auxquels on l'applique, renvoyer un représentant du même ensemble.

Représentants canoniques Pour s'économiser l'écriture d'un programme compatible avec toutes les représentations possible d'une structure abstraite donnée on peut aussi choisir un représentant unique de chaque

classe d'équivalence, appelé ***représentant canonique***, et s'interdire de manipuler toute autre représentation.

Ainsi dans l'exemple précédent on pourrait représenter un ensemble à l'aide d'un tableau dont les éléments devront nécessairement être triés et aucun élément ne devant apparaître deux fois. Cela aurait les conséquences suivantes :

- la fonction `appartient` peut tirer parti de l'ordre imposé des éléments et être plus efficace que la recherche linéaire naïve,
- la fonction `union` doit maintenant nécessairement produire le représentant canonique de l'union des deux ensembles donnés en arguments : elle ne peut se contenter de concaténer les deux tableaux mais doit à la place réaliser une fusion de tableaux triés.

Autres exemples Les graphes sont des structures riches, qui ont naturellement un grand nombre de représentants :

- dans sa représentation par matrice d'adjacence, un graphe peut avoir plusieurs représentations concrètes en fonction de l'indice donné à chaque sommet,
- dans sa représentation par liste d'adjacence, un graphe peut avoir plusieurs représentations concrètes en fonction de l'ordre des voisins dans chaque liste,
- dans sa représentation par dictionnaire d'adjacence, un graphe peut avoir plusieurs représentations concrètes liées aux multiples représentations concrètes d'un dictionnaire donné.

3.5 Représentation des classes d'équivalence : structure *Union-Find*

Étant donnés un ensemble E et une relation d'équivalence $\sim$ sur E, il peut être intéressant de savoir identifier efficacement les éléments de E qui sont ou ne sont pas équivalents. La structure *Union-Find* répond à ce problème en fournissant deux opérations principales :

- find, qui prend en paramètre un élément $e \in E$ et renvoie un représentant canonique r de sa classe d'équivalence $[e]$ (on dit que le représentant r est « canonique » car pour tous les élements de $[e]$ l'opération find renvoie cet unique représentant r), et
- union, qui prend en paramètre deux éléments $e_1, e_2 \in E$ et modifie la structure pour y fusionner les classes d'équivalence $[e_1]$ et $[e_2]$.

L'opération find peut en particulier être utilisée pour tester l'appartenance de deux éléments à la même classe (et donc tester l'équivalence de ces deux éléments), tandis que l'opération union sert à construire la structure *Union-Find* elle-même en indiquant quels éléments doivent être considérés comme équivalents.

Construction. Pour construire une telle structure à partir d'une liste de paires d'éléments équivalents on peut procéder ainsi :

1. Considérer d'abord que tout élément e a sa classe d'équivalence $[e]$ réduite à $\{e\}$, puis
2. pour chaque paire d'éléments e_1, e_2 équivalents, fusionner les classes $[e_1]$ et $[e_2]$.

4 Arbres (13/02)

4.1 Arbres

Un ***arbre*** est un graphe non orienté $G = (S, A, \epsilon)$ non vide, acyclique et connexe (on parle de ***forêt*** quand on a l'acyclicité mais pas la connexité). Un arbre non dégénéré contient un certain nombre de sommets de degré 1, ces sommets sont appelés des ***feuilles*** (dans le cas d'un arbre contenant un unique sommet, ce sommet est également appelé une feuille).

Caractérisations alternatives

- G connexe et $|S| = |A| + 1$.
- G acyclique et $|S| = |A| + 1$.
- G acyclique et ajouter n'importe quelle arête crée un cycle.
- Toute paire de sommets de G est reliée par un unique chemin élémentaire :

$$\begin{aligned}\forall s_1, s_2 \in S,\ (\exists \rho \in \mathsf{chemins}(G),\ & \sigma(\rho) = s_1 \wedge \sigma(\rho) = s_2) \\ \wedge\ (\forall \rho_1, \rho_2 \in \mathsf{chemins}(G),\ & (\sigma(\rho_1) = s_1 \wedge \tau(\rho_1) = s_2 \\ & \wedge \sigma(\rho_2) = s_1 \wedge \tau(\rho_2) = s_2 \\ & \wedge \mathsf{elementaire}(\rho_1) \wedge \mathsf{elementaire}(\rho_2)) \\ & \Longrightarrow \rho_1 = \rho_2)\end{aligned}$$

Exercice

- Justifier ces équivalences.

Caractérisation par construction Les arbres sont les graphes qui peuvent être construits de la manière suivante :

1. démarrer avec un seul sommet et aucune arête,
2. répéter *ad lib.*
   (a) ajouter un nouveau sommet, relié par une arête à l'un quelconque des sommets déjà présents.

On reviendra plus tard dans le semestre sur ce type de constructions.

Exercices

- Cette construction ne produit que des arbres [facile].
- Tous les arbres peuvent être construits de cette manière [plus délicat].

4.2 Arbre couvrant

Un ***arbre couvrant*** d'un graphe non orienté $G = (S, A, \epsilon)$ est un arbre (S', A', ϵ') pour lequel :

- $S' = S$,
- $A' \subseteq A$,
- ϵ' est la restriction de ϵ à A'.

Autrement dit, un arbre couvrant de G est un arbre formé avec les arêtes de G et contenant tous les sommets de G.

Théorème Tout graphe non orienté connexe admet un arbre couvrant.

Une manière de prouver un tel théorème d'existence consiste à définir un algorithme de construction d'un arbre couvrant, et à démontrer que cet algorithme est correct, c'est-à-dire que dans tous les cas le graphe G' produit par cet algorithme est bien un arbre couvrant de G.

Algorithme associé à la preuve

1. On définit $G' = (S, \emptyset, \emptyset)$.
2. Pour chaque arête $a \in A$,
   (a) si $G' \cup \{a\}$ admet un cycle,
   (b) alors ne rien faire
   (c) sinon $G' := G' \cup \{a\}$

Pour prouver la correction d'un algorithme contenant une boucle, on a besoin d'énoncer des propriétés qui rendent compte de la progression de l'algorithme, qu'on appelle des ***invariants***. En l'occurrence :

1. G' acyclique, et
2. tous deux sommets reliés par un chemin dans l'ensemble des arêtes considérées jusque là sont reliés par un chemin dans G'.

Prouver un invariant consiste à montrer que la propriété est valide avant le premier tour de boucle, et que sa validité est préservée par la boucle (c'est-à-dire qu'en prenant comme hypothèse que l'invariant est vrai au début d'un tour boucle, on peut justifier qu'il est encore vrai après un tour de boucle). On en déduit par récurrence que l'invariant est encore vrai après n'importe quel nombre de tours de boucle, et en particulier après le dernier tour.

Correction de l'algorithme Pour tout graphe non orienté $G = (S, A)$ connexe donné en entrée, l'algorithme précédent produit un graphe G' qui est un arbre couvrant de G.
Preuve. On vérifie chacun des invariants.

1. G' acyclique.
   — Initialisation : $(S, \emptyset, \emptyset)$ ne contient pas d'arêtes, et est *a fortiori* acyclique.
   — Préservation : supposons avoir obtenu après n tours de boucle un graphe G' acyclique, et considérons la prochaine arête a.
      — Si $G' \cup \{a\}$ contient un cycle, alors G' n'est pas modifié par l'algorithme, et à la fin du tour G' est donc toujours acyclique.
      — Sinon ($G' \cup \{a\}$ acyclique), G' est modifié en $G' \cup \{a\}$, qui est justement acyclique.

   Donc la propriété « G' est acyclique » est bien un invariant de l'algorithme.
2. Tous deux sommets reliés par un chemin dans l'ensemble des arêtes considérées jusque là sont reliés par un chemin dans G'. Notons A_n l'ensemble des arêtes considérées lors des n premiers tours de boucle ($A_n \subseteq A$).
   — Initialisation : au début on considère $A_0 = \emptyset$. Aucune paire de sommets distincts n'est reliée par un chemin dans le graphe $G_0 = (S, A_0)$ donc il n'y a rien à prouver.
   — Préservation : supposons avoir obtenu après n tours de boucle un graphe $G' = (S, A')$ tel que si deux sommets sont reliés dans $G_n = (S, A_n)$ alors il sont aussi reliés dans G'. Soit a la prochaine arête considérée par l'algorithme, et soient s_1 et s_2 deux sommets reliés par un chemin ρ dans $G_{n+1} = (S, A_n \cup \{a\})$.
      — Si ρ ne contient pas l'arête a, alors ρ est un chemin de s_1 à s_2 dans G_n, et par hypothèse s_1 et s_2 sont reliés par un chemin dans G' (ce qui restera le cas à la fin du tour de boucle).

— Si ρ contient l'arête a, alors il existe un chemin simple ρ' de s_1 à s_2 contenant une seule fois l'arête a, et on peut découper ρ' en $\rho'_1 \cdot a \cdot \rho'_2$, avec ρ'_1 un chemin de s_1 à $\sigma(a)$ et ρ'_2 un chemin de $\tau(a)$ à s_2, tous deux dans G_n.
Par hypothèse il existe deux chemins $\rho_1 : s_1 \longrightarrow \sigma(a)$ et $\rho_2 : \tau(a) \longrightarrow s_2$ dans G' au début du tour de boucle. À la fin du tour de boucle :
  — si $G' \cup \{a\}$ est acyclique alors a est ajoutée à G', et on a alors un chemin $\rho_1 \cdot a \cdot \rho_2$ dans $G' \cup \{a\}$,
  — sinon $G' \cup \{a\}$ admet un cycle, mais G' est acyclique, donc $G' \cup \{a\}$ admet un cycle utilisant l'arête a, donc G' admet un chemin $\rho_3 : \sigma(a) \longrightarrow \tau(a)$, et on a un chemin $\rho_1 \cdot \rho_3 \cdot \rho_2$ de s_1 à s_2 dans G'.

4.3 Arbres enracinés

Un ***arbre enraciné*** est une paire (G, r) où G est un arbre et r un sommet de G qu'on appellera la racine.

Dans un arbre enraciné (G, r), on appelle :
— ***parent*** d'un sommet s le sommet adjacent à s sur le chemin entre s et r,
— ***fils*** d'un sommet s les autres sommets adjacents à s,
— ***ascendants*** d'un sommet s les sommets du chemin entre s et r (s et r compris),
— ***descendants*** d'un sommet s les sommets dont s est un ascendant,
— ***frères*** de s les autres fils du parent de s.

La ***profondeur*** d'un sommet s dans un arbre (G, r) est la longueur (nombre d'arêtes) de l'unique chemin entre r et s. La ***hauteur*** d'un arbre (G, r) est égale à la profondeur maximale de ses sommets.

Le choix d'une racine définit une orientation naturelle pour les arêtes de G, qu'on décrira informellement comme « de la racine vers les feuilles ». On utilise aussi parfois une orientation anti-naturelle, des feuilles vers la racine.

Caractérisation des arbres enracinés avec orientation naturelle
— tous les sommets ont un degré entrant de 1,
— *sauf un* qui a un degré entrant de 0 (la racine).

4.4 Réalisation de l'*Union-Find*

Modélisation avec des arbres. La structure *Union-Find* et ses opérations peuvent être réalisées efficacement grâce à une structure de forêt.

L'ensemble des éléments E est pris comme ensemble de sommets d'un graphe dont chaque composante connexe correspond à une classe d'équivalence pour la relation $\sim$. Plus précisément, chaque composante connexe a la forme d'un arbre orienté dont la racine est prise comme représentant canonique de la classe d'équivalence correspondante.

Les opérations sont alors réalisées comme suit :
— À l'initialisation, on prend le graphe $G = (E, \emptyset)$ dans lequel les composantes connexes sont les sommets $e \in E$ (chacun étant isolé).
— L'opération $\mathsf{find}(e)$ renvoie la racine de l'arbre auquel appartient e.
— L'opération $\mathsf{union}(e_1, e_2)$ combine les arbres de e_1 et de e_2 en un seul, à condition que ces deux sommets ne soient pas déjà dans la même composante. La combinaison des deux arbres est réalisée en

faisant de la racine $r_2 = \text{find}(e_2)$ de l'arbre de e_2 un fils de la racine $r_1 = \text{find}(e_1)$ de l'arbre de e_1 (ou inversement).

Pour une réalisation efficace de l'opération find, on opte pour une orientation anti-naturelle des arbres, en donnant à chaque sommet un pointeur vers son parent.

Réalisation de la structure à base d'arbres avec des tableaux. Comme on utilise une orientation anti-naturelle des arbres, il n'y a qu'une seule information à mémoriser pour chaque sommet : l'identité de son parent (qui sera indéfinie pour les racines).

En numérotant les sommets avec les nombres de l'ensemble $[1, n]$ on peut donc représenter une structure *Union-Find* par un tableau uf de taille $n+1$, dans lequel uf(e) contient le numéro du parent de e (et la case d'indice 0 est ignorée). Pour une racine r, on peut par exemple définir aussi uf(r) = None.

Les opérations peuvent alors être écrites en Python de la manière suivante :

```
# Initialisation pour un ensemble de n éléments
def init(n):
  return [None for _ in range(n+1)]

# Recherche du représentant de la composante de l'élément e
def find(uf, e):
  current = e
  while uf[current] != None
    current = uf[current]
  return current

# Fusion des composantes des éléments e1 et e2
def union(uf, e1, e2):
  r1 = find(e1)
  r2 = find(e2)
  if  r1 != r2:
    uf[r2] = r1
```

La complexité potentielle de l'algorithme se trouve dans la boucle while lors d'un appel à find(e), qui prendra d'autant plus de temps que la profondeur de l'élément e dans son arbre sera importante. Cette idée de base est donc améliorée en faisant en sorte que cette profondeur soit en moyenne aussi petite que possible. En l'occurrence, la version normale de l'*Union-Find* a les deux améliorations suivantes :

— Union par rang : à chaque sommet on ajoute une information appelée « rang », qui donne une approximation de la profondeur de l'arbre dont ce sommet est la racine (cette information n'est plus utile pour les sommets qui ne sont plus des racines). Alors dans l'opération union, au lieu de systématiquement faire de r2 le fils de r1, on prend la racine de plus petit rang pour en faire le fils de l'autre (et on met à jour le rang de la nouvelle racine si besoin). Ainsi on fait la fusion de sorte à minimiser la hauteur de l'arbre obtenu.

— Compression de chemin : lors de chaque opération find, chaque sommet vu lors de la remontée vers la racine est directement rattaché à la racine. Ainsi le chemin de e à find(e) dans le graphe ne sera plus jamais parcouru à nouveau, car il a été remplacé par une unique arête.

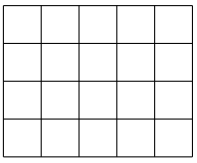
Exercices

— Écrire une variante de l'opération find utilisant une fonction récursive plutôt qu'une boucle while.

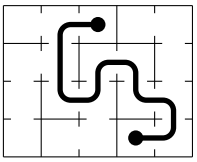
— Coder les optimisations d'union par rang et de compression de chemin.

4.5 Labyrinthe

Énoncé du problème Partons d'un terrain rectangulaire, formé d'une multitude de petites salles carrées séparées par des cloisons.



On souhaite ouvrir des portes dans certaines des cloisons, de sorte à former un *labyrinthe parfait* : on veut que quel que soit le choix d'une salle de départ et d'une salle d'arrivée il existe un unique chemin élémentaire permettant d'aller de l'une à l'autre.



Analyse La condition décrivant le labyrinthe parfait est l'une des caractérisations des arbres. On cherche donc à former un arbre dont les sommets sont les salles et les arêtes sont les portes entre les salles. On cherche de plus à ce que les sommets de l'arbre contiennent toutes les salles : on veut donc un arbre couvrant.

Ainsi, si on considère le graphe G dont les sommets sont les salles et les arêtes sont les cloisons séparant les salles, notre problème revient à construire un arbre couvrant de G.

Résolution On applique l'algorithme de construction d'arbre couvrant vu plus tôt dans ce chapitre en considérant les arêtes dans un ordre aléatoire, et en représentant les ensembles de salles connectées avec une structure d'*Union-Find*.

5 Parcours de graphes (27/02)

5.1 Parcours de structures séquentielles

Tableaux Pour parcourir un tableau t dont chaque élément est accessible par son indice :

1. Soit n la taille du tableau
2. Pour chaque indice de 0 à $n-1$:
   (a) traiter l'élément $t[i]$

```
def print_tab(t):
  for i in range(len(t)):
    print(t[i])
```

Listes chaînées Pour parcourir une liste l où chaque élément est associé à un pointeur vers la suite de la liste :

1. Si la liste est finie :
   (a) ne rien faire.
2. Sinon :
   (a) traiter l'élément courant,
   (b) puis parcourir la suite de la liste.

Dans l'exemple en Python on considère qu'une cellule de la liste est une paire, dont le premier élément est le contenu et le deuxième élément est un pointeur vers la cellule suivante.

```
# Version itérative
def print_list(l):
  current = l
  while current != []:
    print(current[0])
    current = current[1]

# Version récursive
def print_list(l):
  if  l != []:
    print(l[0])
    print_list(l[1])
```

5.2 Parcours d'arbres enracinés

Dans la structure d'arbre il n'y a pas d'indices comme dans les tableaux, mais plutôt des pointeurs comme dans les listes chaînées. Différence en revanche : on n'a plus unicité du pointeur vers « la suite », puisque que chaque sommet a potentiellement plusieurs fils.

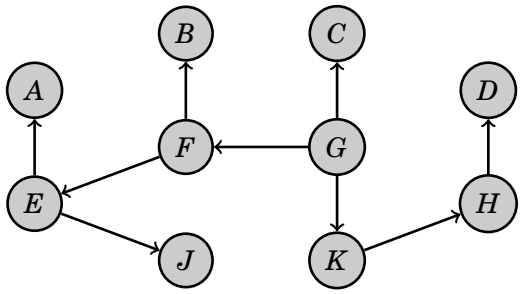
Version 1 : arbres irréguliers Pour parcourir un arbre a où chaque sommet contient un élément et une collection de fils :

1. Traiter l'élément courant
2. pour chaque fils s
   (a) parcourir le sous-arbre à partir de s

Dans l'exemple en Python on considère que chaque sommet de l'arbre est représenté par une paire, dont le premier élement est le contenu et dont le deuxième élément contient la collection des fils (par exemple sous forme de tableau, ou d'ensemble). Dans le cas d'une feuille cette collection de fils est vide.

```
# Version récursive
def print_tree(t):
  print(t[0])
  for  s in t[1]:
    print_tree(s)
```

Dans l'arbre suivant de racine G,



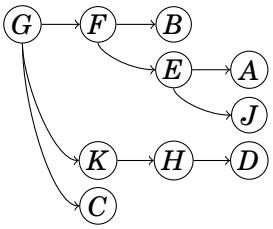
les sommets pourront être traités dans l'ordre

$$G \quad F \quad B \quad E \quad A \quad J \quad K \quad H \quad D \quad C$$

avec un processus qui serait dans le détail :

1. Traiter G, puis F, puis B.
2. Cul-de-sac, revenir au sommet précédent (F) et choisir un autre fils (E). Traiter E, puis A.
3. Cul-de-sac, revenir au sommet précédent (E) et choisir un autre fils (J). Traiter J.
4. Cul-de-sac, revenir au sommet précédent (E). Pas d'autre fils disponible, revenir encore au précédent (F). Pas d'autre fils disponible, revenir encore au précédent (G) et choisir un autre fils (K). Traiter K, puis H, puis D.
5. Cul-de-sac, revenir au sommet précédent (H). Pas d'autre fils disponible, revenir encore au précédent (K). Pas d'autre fils disponible, revenir encore au précédent (G) et choisir un autre fils (C). Traiter C.
6. Cul-de-sac, revenir au sommet précédent (G). Pas d'autre fils disponible, pas de sommet précédent où revenir. Arrêt.

Cet ordre correspond à une lecture de gauche à droite et de haut en bas dans l'arbre redessiné de la manière suivante.



Version 2 : arbres k-aires À partir de maintenant, sauf mention contraire on considérera des ***arbres positionnels***, c'est-à-dire des arbres orientés avec orientation naturelle dans lesquels les fils de chaque sommet sont numérotés (deux fils d'un même sommet ne pouvant porter le même numéro). On appelle ***arbre k-aire*** un arbre positionnel dans lequel les fils de chaque sommet prennent leur numéro dans l'intervalle $[1,k]$. Un cas particulier courant est celui des arbres 2-aires, dits arbres ***binaires***, dans lesquels chaque sommet a au plus deux fils, un appelé ***fils gauche*** portant le numéro 1 et un appelé ***fils droit*** portant le numéro 2 (s'il n'y a qu'un fils, il peut être à gauche ou à droite et cela compte comme deux arbres différents).

Un sommet d'un arbre k-aire peut être représenté comme une paire d'un élément et d'un tableau de k cases contenant les fils de numéros 1 à k (ou un élément vide s'il n'y a pas de fils à l'un des numéros). Le parcours d'un tel arbre devient donc :

1. Si l'arbre est vide :
   (a) ne rien faire.
2. Sinon :
   (a) traiter l'élément courant
   (b) pour chaque entier entre 1 et k
      i. parcourir le sous-arbre de numéro k

```
# Version récursive
def print_tree(t):
  if t != []:
    print(t[0])
    for s in t[1]:
      print_tree(s)
```

Parcours préfixes, suffixes, infixes, mixfixes On a quelques variantes de formes autour de cette première version, en fonction de l'ordre dans lequel sont traités un sommet donné et ses différents fils.

— Un parcours ***préfixe*** traite chaque sommet avant d'explorer ses fils (c'est la situation déjà décrite).
— Un parcours ***postfixe*** explore les sous-arbres fils d'un sommet avant de traiter le sommet.
— Dans le cas d'un arbre binaire, un parcours ***infixe*** explore le sous-arbre gauche d'un sommet, puis traite le sommet, et enfin explore le sous-arbre droit.
— On appelle parcours ***mixfixes*** les autres variantes imaginables.

5.3 Calculs sur des arbres

Les parcours vus ici permettent notamment d'effectuer différents calculs sur les arbres, en collectant des résultats intermédiaires au niveau de chaque sommet.

```
def count_leaves(t):
  if t == []:
    return 1
  else:
    nb = 0
    for s in t[1]:
      nb += count_leaves(s)
    return nb
```

Application : positions gagnantes Pour examiner les différentes séquences de coups possibles dans un jeu (à un ou plusieurs joueurs), on peut explorer l'arbre des coups. On explore dans ce cas un arbre irrégulier donné implicitement, dont les sommets sont des configurations du jeu, et dans lequel les fils d'un sommet s sont les configurations atteintes après avoir joué chaque coup légal à partir de s.

5.4 Parcours d'arbres, version itérative

Lors d'un parcours d'arbre, on peut avoir à un instant donné plusieurs sommets à explorer en attente (en particulier des frères du sommet en train d'être exploré, et des frères de ses ancêtres). Ceci est géré naturellement avec les algorithmes récursif, car chaque nouvel appel correspond

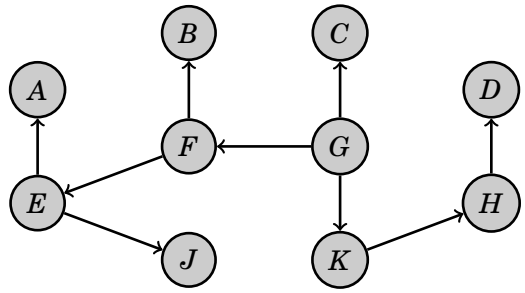
à ouvrir une parenthèse dans l'exploration courante, exploration qui reprendra naturellement dès la parenthèse fermée. Pour obtenir une version itérative de l'algorithme de parcours d'arbre, il faut maintenir explicitement à jour l'ensemble des sommets en attente d'être traités. On peut pour cela utiliser une structure de pile (ou *lifo* pour *last in first out*, « le dernier entré sera le premier sorti »).

— Quand on découvre un nouveau sommet à parcourir on le place sur la pile.
— Quand on a fini de traiter un sommet, on continue avec celui placé en haut de la pile (et on le retire au passage), ou on s'arrête si la pile est vide.

On peut réaliser ceci en Python avec le code suivant :

```
# Version itérative
def print_tree(t):
  stack = [t]
  while stack != []:
    current = stack.pop()
    print(current[0])
    for  s in current[1]:
      stack.append(s)
```

Sur le même arbre que précédemment, de racine G



le processus sera le suivant, avec dans la pile une mémoire explicite des sommets auxquels revenir et dans quel ordre.

Sommet exploré	G	F	B	E	A	J	K	H	D	C
Successeurs ajoutés	C,K,F	E,B	$\emptyset$	J,A	$\emptyset$	$\emptyset$	H	D	$\emptyset$	$\emptyset$
		B		A						
Pile	F	E	E	J	J					
	K	K	K	K	K	K	H	D		
	C	C	C	C	C	C	C	C	C	$\emptyset$

5.5 Parcours de graphe en profondeur

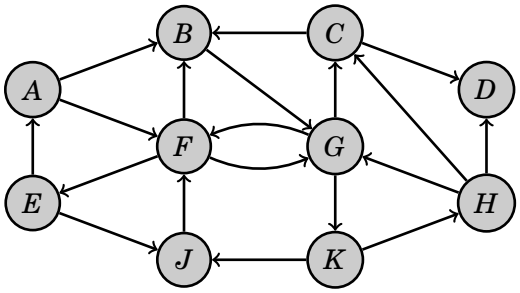
On cherche maintenant à explorer un graphe (potentiellement orienté) à partir d'un de ses sommets, pour trouver les autres sommets accessibles. L'idée de départ consiste, partant d'un sommet s, à considérer tour à tour toutes les arêtes partant de s, et à explorer récursivement à partir de la cible de chacune de ces arêtes, exactement de la même façon que cela était fait pour explorer des arbres.

Différence : il est maintenant possible de retourner à un sommet à partir duquel l'exploration a déjà été faite (voire de retourner au point de départ), et on voudra en général dans ce cas ne pas refaire l'exploration correspondante. Pour cela on maintient une collection des sommets déjà vus, et on ne place dans la pile de sommets en attente que des sommets pas encore vus.

```
# Version itérative
def print_graph(g):
  stack = [g]
  seen = [g]
  while stack != []:
    current = stack.pop()
    print(current[0])
    for  s in current[1]:
      if  not s in seen:
        seen.append(s)
        stack.append(s)
```

Ce mode d'exploration est appelé ***parcours en profondeur***, et est caractérisé comme suit : on s'avance aussi loin que possible sur un chemin donné, pour ne revenir sur nos pas qu'une fois un cul-de-sac atteint. Ceci est lié à la structure de pile qui fait que l'exploration repart toujours des sommets les plus récemment découverts.

À noter : ce mode de parcours correspond à parcourir un arbre formé par certaines des arêtes du graphe. Chaque chemin de l'arbre correspond à un chemin du graphe.



Sommet exploré	G	F	B	E	A	J	K	H	D	C
Successeurs	C,K,F	E,B,G	G	J,A	B,F	F	J,H	G,C,D	$\emptyset$	D
dont déjà vus	$\emptyset$	G	G	$\emptyset$	B,F	F	J	G,C	$\emptyset$	D
Successeurs ajoutés	C,K,F	E,B	$\emptyset$	J,A	$\emptyset$	$\emptyset$	H	D	$\emptyset$	$\emptyset$
		B		A						
Pile	F	E	E	J	J					
	K	K	K	K	K	K	H	D		
	C	C	C	C	C	C	C	C	C	$\emptyset$

5.6 Parcours de graphe en largeur

Alternativement on peut utiliser une structure de file (ou *fifo* pour *first in first out*, « le premier arrivé est le premier servi ») pour enregistrer les sommets en attente. Dans ce cas, l'exploration est toujours faite au niveau des plus anciens sommets découverts et pas encore explorés. Cela revient à procéder en cercles concentriques autour du point de départ.

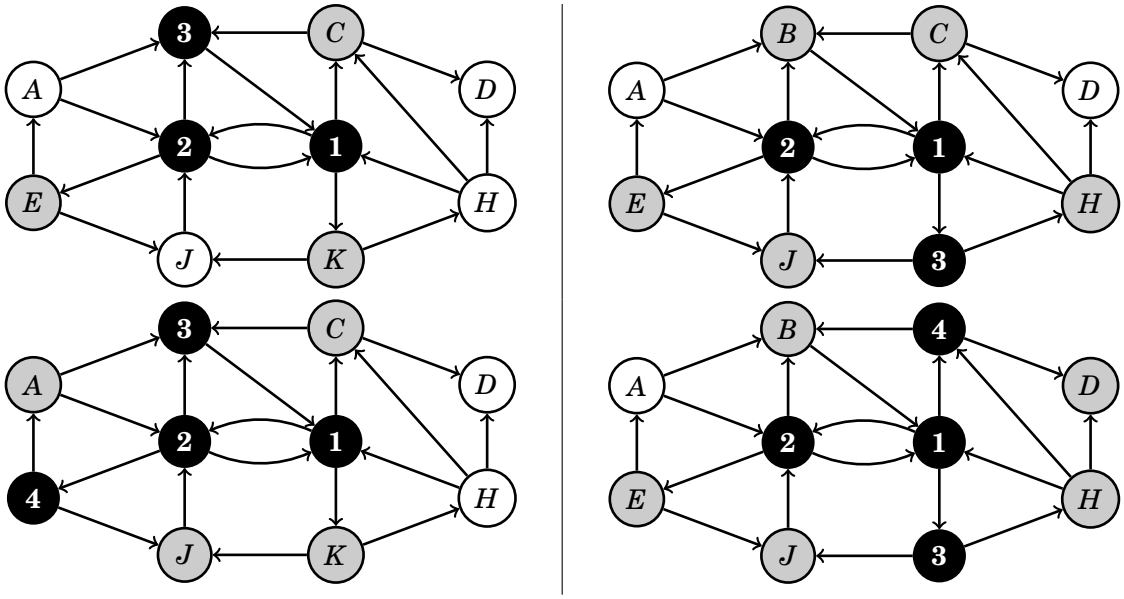
Dans l'exemple suivant (sur le même graphe que précédemment), la file apparaît comme une colonne dans laquelle les éléments sont ajoutés

par le haut et retirés par le bas.

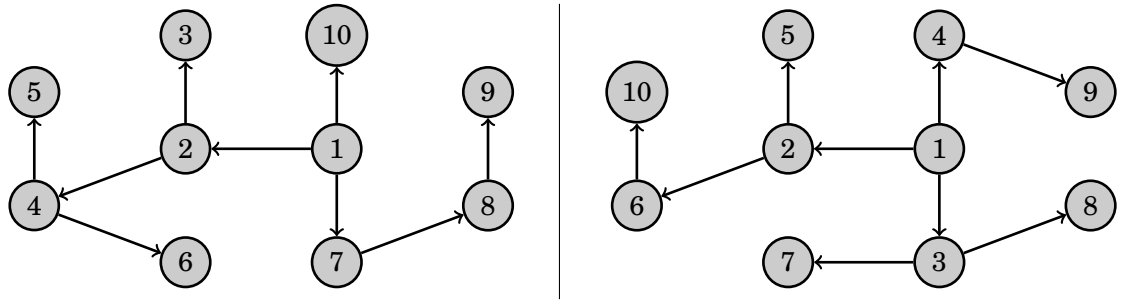
Sommet exploré	G	F	K	C	B	E	J	H	D	A
Successeurs	F,K,C	B,E,G	J,H	B,D	G	A,J	F	G,C,D	$\emptyset$	B,F
dont déjà vus	$\emptyset$	G	$\emptyset$	B	G	J	F	G,C,D	$\emptyset$	B,F
Successeurs ajoutés	F,K,C	E,B	J,H	D	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
			H	D						
		E	J	H	D	A				
File	C	B	E	J	H	D	A			
	K	C	B	E	J	H	D	A		
	F	K	C	B	E	J	H	D	A	$\emptyset$

5.7 Comparaison des deux parcours

Sur ces schémas, on représente en noir les sommets déjà explorés, en gris les sommets déjà vus mais pas encore explorés, et en blanc et les sommets par encore vus. Les schémas de gauche correspondent à l'état du parcours en profondeur et ceux de droite à l'état du parcours en largeur, après 3 puis 4 sommets explorés. Les nombres en blanc dans les sommets noirs indiquent l'ordre dans lequel les sommets ont été explorés.



À chacun des parcours on peut également associer un arbre orienté, dans lequel la racine est le sommet de départ et chaque sommet a pour fils les autres sommets qu'il est le premier à découvrir. Les arbres obtenus sont les suivants, avec les sommets numérotés dans l'ordre de leur exploration.



6 Arbres de recherche (05/03)

6.1 Recherche dichotomique dans un tableau trié

Recherche linéaire dans un tableau quelconque. Pour tester si un élément e apparaît dans un tableau t, dans le cas où ne connaît aucune propriété particulière sur t, il faut parcourir le tableau jusqu'à trouver éventuellement l'élément ou jusqu'à avoir tout inspecté.

```
def search(e,t):
  for i in range(len(t)):
    if t[i] == e:
      return true
  return false
```

L'invariant de la boucle de recherche est : « aucun élément avant l'indice i n'est égal à e ». Autrement dit à partir du i^{e} tour de boucle, si l'élément e est présent dans le tableau alors il se trouve à un indice de l'intervalle $[i, \mathsf{len}(t)[$, ce qui signifie qu'on peut restreindre la zone de recherche à cette tranche du tableau. La complexité de cette recherche est linéaire en moyenne.

Recherche linéaire dans un tableau trié. Si on sait que les éléments du tableau t sont triés par ordre croissant, on peut facilement intégrer à l'algorithme précédent un autre critère de restriction de la zone de recherche : dès qu'on trouve un élément $t[i]$ dans t plus grand strictement que l'élément e cherché, on sait que tous les éléments suivant seront de même strictement plus grands que e. Une optimisation facile consiste à arrêter la recherche dès que les éléments inspectés sont plus grands que l'élément cherché.

```
def search(e,t):
  for i in range(len(t)):
    if t[i] == e:
      return true
    elif t[i] > e:
      return false
  return false
```

L'invariant de la boucle est le même que précédemment : à partir du i^{e} tour de boucle aucun élément avant l'indice i n'est égal à e. Le test supplémentaire revient à déterminer si l'on peut en plus écarter de la zone de recherche toute la fin du tableau à partir de l'indice courant, réduisant ainsi à néant la zone dans laquelle l'élément e est susceptible d'être trouvé.

Si le test est négatif en revanche on n'aura pas fait mieux que l'algorithme précédent. L'ensemble reste de complexité linéaire. La recherche est susceptible de s'arrêter plus tôt lorsque l'élément cherché n'est pas présent dans le tableau mais on a ajouté des tests, le gain de performance n'est pas garanti.

Recherche dichotomique dans un tableau trié. Si on sait que les éléments du tableau t sont triés par ordre croissant, la technique de recherche dichotomique permet avec chaque test d'écarter à coup sûr environ la moitié du tableau. La complexité de la recherche devient alors logarithmique. Pour cela on désigne explicitement une zone du tableau à laquelle la recherche est restreinte, zone qui est d'abord le tableau entier et qui du début à la fin aura la forme d'un intervalle (c'est-à-dire une zone

contiguë). Pour chercher un élément e dans une zone $t[l:u]$ (l pour *lower born* et u pour *upper born*), on compare e à l'élément du milieu de l'intervalle. S'il s'agit de l'élément cherché on s'arrête, et sinon on continue la recherche dans la moitié inférieure ou supérieure en fonction du résultat de la comparaison.

```
def search(e, t):
  l = 0
  u = len(t)
  while l < u:
    m = (l+u)/2
    if  t[m] == e:
      return true
    elif  t[m] < e:
      l = m+1
    else:
      u = m
  return false
```

Au cours de la boucle la recherche a lieu entre l'indice l inclus et l'indice u exclu. L'invariant est cette fois qu'à aucun moment l'élément cherché e ne peut se trouver à une position avant l'indice l (exclu) ou à partir de l'indice u (inclus). Autrement dit : il n'est pas certain que l'élément cherché e apparaisse dans le tableau, mais s'il y est présent il se trouve nécessairement dans la zone de recherche.

Pour justifier que l'algorithme fonctionne il faut de plus s'assurer que l'indice m utilisé comme « milieu » est bien toujours dans l'intervalle $[l,u[$ et que la taille de l'intervalle de recherche réduit à chaque tour. Pour assurer que l'algorithme s'arrête au plus tard après un nombre de tours de boucle logarithmique en la taille du tableau, il faut de plus justifier que la « moitié » de la zone à laquelle est restreinte la recherche porte bien son nom et a bien une taille proche à un arrondi près de la moitié de la taille de la zone précédente.

Bilan : cette structure permet une recherche en temps logarithmique. Cependant l'insertion d'un nouvel élément peut être très coûteuse : l'élément doit être placé au bon endroit pour respecter la propriété selon laquelle le tableau est trié, et cela implique de décaler tous les éléments suivants. On a donc une complexité linéaire.

Exercices

- En notant l, u les valeurs des variables `l` et `u` au début d'un tour de boucle et l', u' les valeurs de ces mêmes variables après ce tour de boucle, on a $u'-l' \leq (u-l)/2$.
- Pour un tableau de départ dont la taille est dans l'intervalle $[2^k, 2^{k+1}[$ l'algorithme exécute au plus $k+1$ tours de boucle.

6.2 Arbres k-aires

Rappel. Un arbre ***k-aire*** est un arbre enraciné avec orientation naturelle des arêtes (de la racine vers les feuilles), dans lequel chaque sommet possède entre 0 et k fils portant chacun un numéro différent pris dans l'intervalle $[1,k]$. Dans un tel arbre, la ***profondeur*** d'un sommet s est la longueur de l'unique chemin entre r et s (comptée en nombre d'arêtes), et la ***hauteur*** d'un arbre est égale à la profondeur maximale atteinte par ses sommets. Un arbre ***binaire*** est un arbre 2-aire.

Arbres k-aires particuliers. Un arbre k-aire est ***plein*** si chacun de ses sommets a soit 0 fils soit k fils, autrement dit si chaque sommet qui n'est pas une feuille a exactement k fils. Un arbre k-aire ***parfait*** est un arbre k-aire plein dont toutes les feuilles ont la même profondeur. On parle parfois aussi d'arbre ***quasi-parfait*** pour désigner un arbre k-aire parfait auquel il manque seulement un certain nombre de feuilles, les feuilles manquantes devant être « les plus à droite ».

Un arbre k-aire est ***équilibré*** si pour chaque sommet s de cet arbre les hauteurs des sous-arbres définis par les k fils de s ne diffèrent pas de plus de 1. Dans le cas d'un arbre qui n'est pas plein, les fils manquants sont considérés comme des sous-arbres de hauteur -1 : les autres fils d'un tel sommet non plein doivent être des feuilles pour que l'arbre soit équilibré.

Exercices

— Un arbre binaire parfait de hauteur h a 2^h feuilles, et $2^{h+1}-1$ sommets au total (se généralise à k^h et $k^{h+1}-1$ pour les arbres k-aires parfaits).

— Un arbre binaire plein de hauteur h a au minimum $2h+1$ sommets (se généralise à $kh+1$ pour les arbres k-aires pleins).

— Un arbre binaire plein avec n sommets a $(n+1)/2$ feuilles et $(n-1)/2$ sommets qui ne sont pas des feuilles, et on sait en outre que n est impair.

— Un arbre binaire avec n sommets a une hauteur comprise entre $\log_2(n)$ et $n-1$ (et ces deux bornes peuvent être atteintes).

— Un arbre binaire équilibré de hauteur h a au moins $F_{h+3}-1$ sommets, où F_k est la suite de Fibonacci ($F_0=0$, $F_1=1$, $F_{k+2}=F_{k+1}+F_k$). *La suite de Fibonacci ayant une croissance exponentielle, on en déduit qu'un arbre binaire équilibré a une hauteur logarithmique en le nombre de sommets qu'il possède.*

6.3 Structures de données arborescentes

Arbres annotés. On peut associer des informations à chaque sommet d'un arbre, pour :

— représenter des choses intrinsèquement arborescentes (voir chapitre 9), ou

— organiser une structure de données sous la forme d'un arbre (voir tout de suite).

Pour donner une structure souple à une collection d'éléments, on peut faire de chaque élément un sommet d'un arbre (éventuellement en se restreignant aux feuilles).

Abstraitement, on a un arbre dans lequel à chaque sommet (ou chaque feuille) est associé une donnée. Concrètement on dispose d'au moins deux techniques :

— Une technique « universelle », adaptée à tous les arbres k-aire : chaque sommet est une paire d'un élément et d'un tableau. Comme vu précédemment, le tableau contient les fils du sommet (ou plutôt des pointeurs vers les fils), et une valeur nulle pour les fils qui ne sont pas présents. Une structure classique : arbre binaire de recherche.

— Le cas particulier d'une collection d'éléments arrangés dans un arbre binaire quasi-parfait peut être représenté par un unique tableau contenant tous les éléments, dans lequel la structure de

l'arbre (c'est-à-dire l'ensemble des arêtes parent/fils) est implicite : les fils du sommet d'indice i sont les sommets d'indices $2i+1$ et $2i+2$. Une structure classique : tas binaire.

6.4 Arbres binaires de recherche

On cherche une structure d'arbre dans laquelle la recherche d'un élément peut être similaire à la recherche dichotomique :

— On considère d'abord un élément médian m (et on s'arrête s'il est exactement l'élément e cherché) ;
— si l'élément cherché e est inférieur à l'élément médian m, alors on poursuit la recherche « à gauche »,
— sinon (e supérieur à m) on poursuit la recherche « à droite ».

Avec une structure d'arbre binaire, on peut prendre la racine comme élément médian, et ses sous-arbres gauche et droit comme parties gauche et droite où poursuivre la recherche.

L'invariant important des tableaux triés permettant la recherche dichotomique était : tous les éléments de la partie gauche sont inférieurs à l'élément médian, et tous les éléments de la partie droite sont supérieurs à l'élément médian. Ainsi par exemple, si on cherchait un élément plus petit que l'élément médian il était garanti qu'il ne se trouvait pas dans la partie droite, qui pouvait être ignorée dans la suite de la recherche. On peut transposer cette propriété dans le cadre des arbres.

Un ***arbre binaire de recherche*** (ABR) est un arbre binaire annoté dans lequel, pour tout sommet s portant un élément m, tous les éléments présents dans le sous-arbre gauche de s sont inférieurs à m et tous les éléments présents dans le sous-arbre droit de s sont supérieurs à m.

Dans de tels arbres, on peut transposer l'algorithme de recherche dichotomique. La condition d'arrêt selon laquelle on abandonne la recherche et on déclare que l'élément n'appartient pas à la collection lorsque la zone de recherche est vide correspond ici à tester la vacuité de l'arbre.

```
# Version impérative
def search(e, a):
  courant = a
  while courant != []:
    if  e == a[0]:
      return true
    elif  e < a[0]:
      courant = a[1]
    else:  # e > a[0]
      courant = a[2]
  return false

# Version récursive
def search(e, a):
  if  a == []:
    return false
  if  e == a[0]:
    return true
  elif  e < a[0]:
    return search(e, a[1])
  else:  # e > a[0]
    return search(e, a[2])
```

Cet algorithme répond dans le pire cas en un temps proportionnel à la hauteur de l'arbre. Son efficacité est donc garantie lorsque les arbres ont

une faible hauteur (par exemple : arbres équilibrés), mais est aussi potentiellement mauvaise pour des arbres de grande hauteur.

Pour ajouter un élément dans un arbre binaire de recherche tout en préservant la propriété des ABR, il suffit de commencer par utiliser l'algorithme de recherche, puis :

— si l'élément a été trouvé, ne rien faire,

— sinon, on a atteint un sous-arbre vide, qu'on remplace par une feuille contenant l'élément à ajouter.

```
# Version impérative
def add(e, a):
  # Si l'arbre était vide, on en renvoie un nouveau
  if a is None: return [e, None, None]
  # Sinon on le parcourt jusqu'à trouver la bonne position
  courant = a # Sommet en cours d'examen
  parent = None # Predecesseur
  position = None # Numero de [courant] dans les fils de [parent]
  while courant is not None:
    # Rien à faire si l'élément est deja là
    if e == courant[0]: return a
    # Sinon on avance dans l'arbre
    parent = courant
    if e < courant[0]:
      position = 1
      courant = courant[1]
    else: # e > courant[0]
      position = 2
      courant = courant[2]
  # À la fin on insère [e,None,None] du bon côté du dernier sommet
  parent[position] = [e, None, None]
  return a

# Version récursive
def add(e, a):
  if a == []: return [e,[],[]]
  if e < a[0]:
    a[1] = add(e, a[1])
  elif e > a[0]:
    a[2] = add(e, a[2])
  return a
```

Cet algorithme d'insertion ne garantit pas que l'arbre formé sera équilibré.

Exercices

— Un même ensemble d'éléments peut être représenté par plusieurs arbres de recherche différents.

— L'insertion successive de n éléments dans un arbre vide au départ peut amener à un arbre de recherche de profondeur proportionnelle à n.

— Donner un algorithme pour trouver le plus petit élément dans un arbre binaire de recherche.

— Donner un algorithme pour supprimer un élément d'un arbre binaire de recherche (le résultat doit toujours être un ABR).
Indication : retirer une feuille ne pose pas de problème. Si l'élément à retirer est un nœud interne en revanche, pour préserver la structure d'arbre il faut ramener à sa place un autre élément bien choisi de l'arbre.

6.5 Arbres de recherche équilibrés

Il est possible d'ajuster les opérations d'insertion et de suppression pour que les arbres de recherche formés soient toujours équilibrés.

Exercices

— Pour tout arbre binaire de recherche A, il existe un arbre binaire de recherche équilibré contenant les mêmes éléments que A.

— L'arbre binaire ci-dessous à gauche est un arbre de recherche si et seulement si l'arbre à droite l'est également (e_1 et e_2 sont des éléments, et a_0, a_1 et a_2 sont des arbres).