Outils logiques et algorithmiques

Florent Hivert d'après Thibaut Balabonski @ Université Paris-Saclay
Édition 2026.

Deuxième partie

Graphes

4 Gérer des conflits

4.1 Problème : allocation de ressources

Panique à l'université Saris-Paclay. Chaque filière a, chacune dans son coin, fixé son emploi du temps. Il y a chaque semaine des milliers de cours prévus, et il faut maintenant trouver une salle à chacun. Comme vous pouvez vous en douter, il y a nettement moins de salles disponibles que de cours à y loger quotidiennement : plusieurs devront certainement se succéder dans une même salle le même jour. Mais, évidemment, deux cours ne peuvent avoir lieu dans la même salle que si leurs horaires ne se recouvrent pas[1]. Nous avons donc :

- un ensemble de cours, dont certains peuvent se tenir dans une même salle (car leurs horaires sont disjoints) mais d'autres ne le peuvent pas (car leurs horaires se recouvrent),
- et un certain nombre de salles où loger nos cours.

L'objectif est d'affecter une salle à chaque cours en respectant ces contraintes.

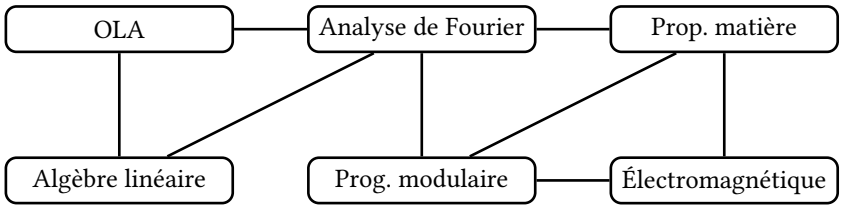
Échauffement. Voici un extrait des cours du lundi matin :

- Algèbre linéaire : 8h15–10h15
- Analyse de Fourier pour la physique : 8h15–10h45
- Outils logiques et algorithmiques : 8h45–10h15
- Transformations et propriétés de la matière : 10h30–12h00
- Programmation modulaire : 10h30–12h30
- Électromagnétique : 11h–12h45

De combien d'amphithéâtres avez-vous besoin au minimum pour organiser ces six cours ?

4.2 Modélisation : graphes non orientés

Points et traits. La structure centrale de notre problème est un ensemble d'éléments (ici : des cours), dont certains sont en relation l'un avec l'autre (ici : compatibles ou incompatibles). On peut résumer cette structure par un schéma dans lequel chaque cours occupe un point de l'espace, et où certains cours sont reliés par des traits, en fonction de leur relation. Voici notre exemple précédent, où on a dessiné un lien entre les cours qui sont incompatibles.



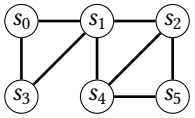
On appelle une telle structure un *graphe*.

1. Pour les besoins du scénario, on néglige les questions de capacité et de nature des salles : on suppose que toute salle peut accueillir n'importe quel cours.

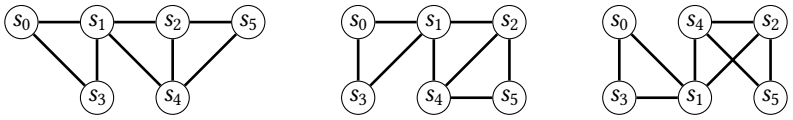
Définitions. Un ***graphe*** est formé par une paire (S, A) où :

- S est un ensemble d'éléments appelés ***sommets*** (ou *nœuds*),
- A est un ensemble d'éléments appelés ***arêtes*** (ou *arcs*, ou *flèches*),
- chaque arête $a \in A$ a deux extrémités $s, t \in S$.

On peut « dessiner » un graphe en représentant chaque sommet par un point du plan et chaque arête par un trait liant ses deux extrémités. Ci-dessous, un graphe à six sommets et huit arêtes. Il y a une arête entre les sommets s_0 et s_1, mais pas entre les sommets s_0 et s_4.

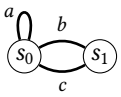


Dans un tel dessin, les positions exactes des différents sommets n'ont aucune importance, seules comptent les relations qu'entretiennent entre eux les sommets. Les dessins ci-dessous représentent tous le même graphe.



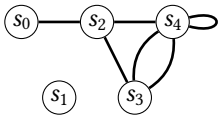
Deux sommets liés par une arête sont ***adjacents*** (ou *voisins*). Une arête a ayant un sommet s parmi ses extrémités est ***incidente*** à s.

On appelle ***boucle*** une arête dont les deux extrémités sont identiques (arête a ci-dessous). On parle d'***arêtes multiples*** (ou *parallèles*) lorsque plusieurs arêtes ont les mêmes deux extrémités (arêtes b et c ci-dessous).



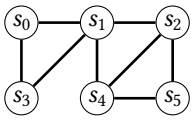
Un ***graphe simple*** est un graphe qui n'a ni boucles, ni arêtes multiples.

Le ***degré*** d'un sommet s d'un graphe simple, noté $\delta(s)$, est le nombre de voisins de ce sommet, équivalent au nombre d'arêtes ayant s pour extrémité. On généralise la notion aux graphes quelconques en comptant le nombre d'extrémités d'arêtes touchant s : des arêtes parallèles comptent chacune pour 1, et une boucle compte pour 2.



$$\begin{array}{lcllcl} \delta(s_0) & = & 1 & \delta(s_3) & = & 3 \\ \delta(s_1) & = & 0 & \delta(s_4) & = & 5 \\ \delta(s_2) & = & 3 & & & \end{array}$$

Problème du coloriage. Le problème du ***coloriage*** d'un graphe consiste à donner à chaque sommet une *couleur*, de sorte que les sommets adjacents aient toujours des couleurs différentes.



0	orange
1	violet
2	orange
3	turquoise
4	turquoise
5	violet

Notre problème d'allocation d'amphithéâtres peut se ramener au coloriage d'un graphe :

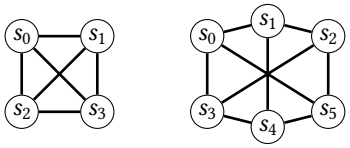
1. créer un graphe dont les sommets sont les cours auxquels affecter des salles, avec une arête entre deux sommets lorsque les cours correspondants sont incompatibles,
2. colorier le graphe en donnant des couleurs différentes aux sommets adjacents,
3. chaque couleur représente un amphithéâtre !

Remarquez que d'autres solutions sont possibles.

Si on reprend notre sélection de cours, son graphe, et le coloriage exemple précédent, il suffit donc de trois amphithéâtres :

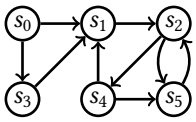
- un pour OLA et Propriétés de la matière,
- un pour Analyse de Fourier et Électromagnétisme,
- un pour Algèbre linéaire et Prog. modulaire.

Apparté : graphes planaires. Lorsque l'on peut dessiner un graphe de manière à ce que ses arêtes ne se croisent pas, on dit que ce graphe est ***planaire***. Le graphe dessiné ci-dessous à gauche est planaire : même si deux arêtes se croisent sur le dessin, il est possible d'éviter le croisement en plaçant différemment un sommet, ou en tordant une arête. Celui de droite en revanche, n'est pas planaire.



Un théorème célèbre, le *théorème des quatre couleurs*, affirme que quatre couleurs suffisent toujours pour colorier un graphe planaire.

Apparté : graphes orientés. Dans un ***graphe orienté***, chaque arête a une « direction ». On distingue ainsi pour chacune une extrémité de ***départ*** (ou *source*) et une extrémité d'***arrivée*** (ou *cible*). Dans le dessin d'un tel graphe, on représente une arête par une flèche allant du sommet de départ au sommet d'arrivée.



Pour un sommet s d'un tel graphe, le degré $\delta(s)$ se décompose en :

— un ***degré entrant*** : nombre d'arêtes dont s est l'arrivée,

— un ***degré sortant*** : nombre d'arêtes dont s est le départ.

Nous reviendrons sur la notion de graphe orienté au prochain chapitre.

4.3 Structure de données : graphe

Pour travailler avec des graphes simples non orientés, on utilisera principalement les opérations suivantes :

— énumérer les sommets,

— étant donné un sommet s, énumérer les voisins de s.

On peut par exemple imaginer le procédé suivant pour connaître le nombre total d'arêtes d'un graphe : pour chaque sommet s, compter les voisins de s, et à la fin diviser le total par 2.

Question : pourquoi divise-t-on la somme obtenue par 2 ?

Interface. Pour simplifier le code, on suppose toujours manipuler un graphe dont les sommets sont numérotés à partir de zéro. Ainsi, pour un graphe à n sommets, on désignera chaque sommet par un nombre pris dans l'intervalle $[0, n[$. Énumérer les sommets d'un graphe ou les voisins d'un sommet consiste donc à énumérer une liste d'entiers. Pour représenter un graphe, on veut donc essentiellement réaliser l'interface suivante, où `Iterable<Integer>` est le type des énumérations d'entiers pouvant être parcourues à l'aide d'une boucle *for each*.

```
interface Graphe {
    Iterable<Integer> sommets();
    Iterable<Integer> voisins(int s);
}
```

Notre fonction comptant le nombre d'arêtes d'un graphe réalisant cette interface s'écrirait ainsi.

```
static int nombreAretes(Graphe g) {
    int d = 0;
    for (int s : g.sommets()) {
        for (int v : g.voisins(s))
            d += 1;
    }
    return d/2;
}
```

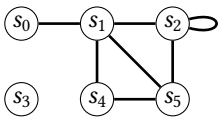
Ces deux opérations d'énumération des sommets et des voisins permettent également facilement de réaliser des fonctions renvoyant le nombre de sommets d'un graphe, ou le degré d'un sommet. On supposera par la suite que ces méthodes existent bien.

Faites-le.

Réalisation 1 : matrice d'adjacence. Un graphe G à n sommets $\{s_0, \dots, s_{n-1}\}$ et sans arêtes parallèles peut être représenté par une matrice M, carrée, d'ordre n, telle que :

- $M[i, j]$ = vrai s'il existe une arête entre s_i et s_j,
- $M[i, j]$ = faux sinon.

On appelle M la ***matrice d'adjacence*** de G. Remarque : dans les graphes *non orientés* que nous considérons ici, une arête entre s_i et s_j est aussi une arête entre s_j et s_i. Ainsi, chaque arête qui n'est pas une boucle donne deux cases à vrai dans la matrice, qui est symétrique.



	0	1	2	3	4	5
0		✓				
1	✓		✓		✓	✓
2		✓	✓			✓
3						
4		✓				✓
5		✓	✓		✓	

En java, on peut représenter une telle matrice par un simple tableau à deux dimensions. Voici le début du code. Pour créer un graphe, on se donne un constructeur prenant en paramètre le nombre de sommets et initialisant la matrice, et une méthode ajoutant un lien entre deux sommets.

```
class GrapheMat implements Graphe {
    public final int taille;
    private boolean[][] adj;

    public GrapheMat(int taille) {
        this.taille = taille;
        this.adj = new boolean[taille][taille];
    }

    public void ajouteArete(int s, int t) {
        adj[s][t] = true;
        adj[t][s] = true;
    }
```

Question : dans quel ordre sont renvoyés les voisins ici ?

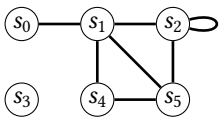
Pour énumérer les voisins d'un sommet s_i, on construit et on renvoie la liste des j tels que $M[i, j]$ vaut vrai.

```
    public Iterable<Integer> voisins(int s) {
        ArrayList<Integer> voisins = new ArrayList<>();
        for (int v=0; v<taille; v++) {
            if (adj[s][v]) voisins.add(v);
        }
        return voisins;
    }
```

Pour énumérer les sommets, c'est-à-dire les entiers de 0 à $n-1$, on pourrait de même construire une liste à la main et la renvoyer. Voici une variante dans laquelle on se passe de construire la liste. À la place, on concrétise la classe abstraite `AbstractList` en indiquant que l'élément d'indice i dans cette énumération est précisément i lui-même.

```
    public Iterable<Integer> sommets() {
        return new AbstractList<Integer>() {
            public Integer get(int index) { return index; }
            public int size() { return taille; }
        };
    }
}
```

Réalisation 2 : listes d'adjacence. Un graphe G à n sommets $\{s_0, \dots, s_{n-1}\}$ et sans arêtes parallèles peut également être représenté par un tableau A de taille n tel que $A[i]$ contient la liste des [numéros des] voisins du sommet s_i.



0	1			
1	0	2	4	5
2	1	2	5	
3				
4	1	5		
5	1	2	4	

Voici un code java pour cette seconde réalisation. Chaque liste de voisins est du type `ArrayList<Integer>`. Pour le tableau A, on utilise un nouveau `ArrayList` contenant ces listes de voisins [2].

```
class GrapheAdj implements Graphe {
    public final int taille;
    private ArrayList<ArrayList<Integer>> adj;

    public GrapheAdj(int taille) {
        this.taille = taille;
        this.adj = new ArrayList<>(taille);
        for (int s=0; s<taille; s++) adj.add(new ArrayList<>());
    }
    public void ajouteArete(int s, int t) {
        adj.get(s).add(t);
        adj.get(t).add(s);
    }
    public Iterable<Integer> voisins(int s) {
        return adj.get(s);
    }
    public Iterable<Integer> sommets() { /* identique à GrapheMat.sommets() */ }
}
```

Coût des graphes. Les deux propositions de réalisation des graphes ont des coûts différents. Si l'on considère un graphe à n sommets et k arêtes, voici les ordres de grandeur.

- Une matrice d'adjacence nécessite en mémoire un tableau de n^2 booléens. L'énumération des voisins d'un sommet a une complexité proportionnelle à n.
- Un tableau de listes d'adjacence nécessite en mémoire un tableau de n références, plus n listes contenant chacune n entiers au maximum. Plus précisément, ces listes réunies contiennent environ $2k$ éléments. L'énumération des voisins d'un sommet a une complexité proportionnelle au nombre de voisins.

Bilan : le plus souvent, la représentation par listes d'adjacence est plus efficace, autant pour l'utilisation de mémoire (car $n + k \leq n^2$) que pour le coût de l'énumération des voisins (car le nombre de voisins d'un sommet est toujours inférieur à n).

La représentation par matrice d'adjacence reste cependant à privilégier dans le cas d'un graphe *dense*, c'est-à-dire dans lequel la majorité des arêtes possibles sont présentes : la structure, plus simple, est alors plus efficace en pratique. Regardons précisément les constantes associées aux ordres de grandeur $\Theta(n^2)$ et $\Theta(n + k)$ pour la représentation en mémoire.

- Chaque booléen de la matrice d'adjacence occupe un octet, d'où un nombre d'octets $\sim n^2$ pour une matrice d'adjacence.
- Les listes d'adjacence contiennent des références vers des objets `Integer`. Chaque référence occupe huit octets, et chaque objet en java occupe en mémoire au moins 12 octets de plus que la place utilisée pour la donnée elle-même (pour `Integer`, contenant des entiers **`int`** de 4 octets, on a donc $12 + 4 = 16$ octets). En outre, chaque tableau redimensionnable `ArrayList` peut avoir une taille double du nombre d'éléments effectivement contenus. Le nombre d'octets pour un tableau de listes d'adjacence est donc compris entre $\sim 24(k + n)$ et $\sim 32(k + n)$.

Bilan : pour un graphe non orienté contenant un nombre d'arêtes proche du maximum $\sim n^2/2$, la matrice d'adjacence est plus compacte.

En outre, une matrice d'adjacence dense a un meilleur comportement vis-à-vis du cache, d'où un possible gain de vitesse.

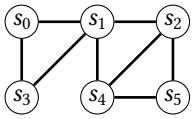
2. On pourrait vouloir utiliser un tableau primitif pour A au lieu d'un `ArrayList`, mais on ne le fait pas ici pour des raisons techniques liées au système de types de java : les tableaux primitifs ne peuvent pas, du moins en théorie, contenir des types génériques comme `ArrayList<T>`.

4.4 Algorithme : coloriage glouton

On va numéroter nos « couleurs » par des nombres entiers à partir de zéro. On considère donc l'ensemble infini de couleurs $\{c_0, c_1, c_2, ...\}$. Produire une coloration pour un graphe de sommets $\{s_0, \dots, s_{n-1}\}$, c'est donc produire un tableau C de n entiers, tel que $C[i]$ est le numéro de la couleur affectée au sommet s_i.

On cherche à produire un coloriage utilisant un nombre de couleurs aussi petit que possible. Un algorithme de coloriage ***glouton*** consiste à considérer les sommets un à un, et à choisir pour chacun à son tour la couleur qui paraît la plus adaptée pour lui. On considérera comme « couleur la plus adaptée » pour un sommet s, la plus petite des couleurs qui n'a pas encore été utilisée pour un des voisins de s.

Reprenons notre exemple, et considérons les sommets dans l'ordre donné par leurs numéros.



Sommet	Couleurs voisines	Couleur choisie
s_0	aucune	c_0
s_1	$c_0(s_0)$	c_1
s_2	$c_1(s_1)$	c_0
s_3	$c_0(s_0), c_1(s_1)$	c_2
s_4	$c_0(s_2), c_1(s_1)$	c_2
s_5	$c_0(s_2), c_2(s_4)$	c_1

En définissant c_0 comme orange, c_1 comme violet et c_2 comme turquoise, on obtient exactement le coloriage proposé plus tôt.

Code java. Dans le code, on initialise un tableau de couleurs avec le numéro `-1` pour les sommets pas encore coloriés. Lors du traitement d'un sommet, on enregistre les couleurs des sommets voisins dans un tableau de booléens. Il ne reste ensuite plus qu'à considérer chaque nombre entier à partir de `0` jusqu'à en trouver un qui n'a pas été coché dans ce tableau de booléens.

```
static int[] colorie(Graphe g) {
    // initialisation du tableau des couleurs
    int[] couleurs = new int[g.taille()];
    for (int s : g.sommets()) couleurs[s] = -1;
    // coloriage de chaque sommet
    for (int s : g.sommets()) {
        // énumération des couleurs des voisins
        int d = g.degre(s);
        boolean[] couleursVoisines = new boolean[d+1];
        for (int v : g.voisins(s)) {
            int c = couleurs[v];
            if (0 <= c && c <= d) couleursVoisines[c] = true;
        }
        // recherche de la première couleur disponible
        for (int c=0;; c++) {
            if (!couleursVoisines[c]) {
                couleurs[s] = c;
                break;
            }
        }
    }
    return couleurs;
}
```

Note : l'ensemble des couleurs voisines contient également la « fausse » couleur `-1` si l'un des voisins n'a pas encore été colorié. Cela n'a aucun impact sur le déroulement de l'algorithme.
Note : on suppose également que l'interface des graphes contient des méthodes `int taille()` et `int degre(int s)` renvoyant respectivement le nombre de sommets du graphe et le degré d'un sommet.
Remarque : dans ce code, on définit la taille du tableau de booléens des couleurs voisines en fonction du degré du sommet. Pourquoi ?

4.5 Approfondissement : analyse du coloriage glouton.

Évaluons la complexité de notre algorithme de coloriage, et la qualité du coloriage produit.

Complexité temporelle. La fonction `colorie` est la succession de deux boucles.

- La première boucle, initialisant le tableau `couleurs`, a une complexité proportionnelle au nombre n de sommets du graphe.
- La deuxième boucle, qui est la principale, réalise les opérations suivantes pour chaque sommet s.
  1. Énumération des voisins de s, pour remplir `couleursVoisines` : complexité proportionnelle au degré $\delta(s)$ de s.
  2. Recherche d'une couleur disponible : complexité encore proportionnelle à $\delta(s)$.

  D'où complexité totale de la deuxième boucle proportionnelle à la somme des degrés des sommets, c'est-à-dire proportionnelle au nombre k d'arêtes du graphe.

D'où complexité totale $\Theta(n + k)$.

Qualité du coloriage. L'algorithme glouton ne trouve pas systématiquement le plus petit nombre de couleurs possible pour le graphe auquel on l'applique. On peut cependant démontrer que le nombre χ de couleurs utilisées est borné par les degrés des sommets du graphe. Plus précisément, si on note $\Delta_{max} = \delta(s_{max})$ le degré d'un sommet s_{max} ayant le plus grand degré dans notre graphe, alors on a $\chi \leq \Delta_{max} + 1$.

Savez-vous trouver un contre-exemple ?

Démonstration. Considérons le choix d'une couleur pour un sommet s quelconque du graphe. Par définition, ce sommet a $\delta(s)$ voisins. Ces voisins utilisent donc au maximum $\delta(s)$ couleurs différentes. Ainsi, il existe au moins un nombre non utilisé dans l'intervalle $[0, \delta(s)]$: la couleur choisie pour s sera nécessairement dans cette intervalle, et en particulier inférieure ou égale à Δ_{max}. Finalement, l'algorithme glouton ne choisit que des couleurs appartenant à l'intervalle $[0, \Delta_{max}]$, et leur nombre est donc inférieur ou égal à $\Delta_{max} + 1$.

4.6 Approfondissement : relations binaires

Une *relation binaire* entre les éléments de deux ensembles A et B est un ensemble d'associations entre un élément de A et un élément de B, caractérisant des éléments qui sont « en relation » l'un avec l'autre. La nature de cette « relation » peut couvrir des situations extrêmement variées. Par exemple :

- similarité entre objets, comme l'égalité $=$,
- hiérarchie entre un tout et ses parties, comme l'appartenance $\in$,
- comparaison de grandeurs, comme la comparaison $\leq$,
- antécédents et images d'une fonction,
- dépendance entre deux événements,
- incompatibilité ou interférence entre deux faits,
- accessibilité d'un point d'arrivée depuis un point de départ...

Certains de ces exemples correspondent a des types de relations importantes en mathématiques, à savoir les relations fonctionnelles, les relations d'ordre et les relations d'équivalence, que nous aborderons progressivement. Certains aussi correspondent directement à des problèmes que l'on peut modéliser et résoudre à l'aide de graphes.

Relations binaires. Une ***relation binaire*** $\mathcal{R}$ entre deux ensembles A et B est un sous-ensemble du produit cartésien $A \times B$. Dans ce contexte, on note couramment $a\mathcal{R}b$ ou $\mathcal{R}(a, b)$ pour $(a, b) \in \mathcal{R}$. On parle de relation binaire ***homogène*** lorsque les ensembles A et B sont égaux.

Relations fonctionnelles. Une relation fonctionnelle est une relation binaire décrivant le lien entre les entrées et les sorties d'une fonction. Une telle relation $\mathcal{R}_f$ pour une fonction $f : A \to B$ contient donc les paires (a, b) telles que $f(a) = b$.

La caractéristique principale d'une telle relation, qui définit le concept de fonction, est que la sortie $f(a)$ est uniquement déterminée par l'entrée a. Autrement dit, il ne peut pas y

avoir deux images associées au même antécédent. Une relation $\mathcal{R}_f \subseteq A \times B$ est ***fonctionnelle*** si pour tout $a \in A$ il existe au plus un $b \in B$ tel que $\mathcal{R}_f(a, b)$.

$$\forall a \in A,\ \forall b_1, b_2 \in B,\ a\mathcal{R}b_1 \wedge a\mathcal{R}b_2 \Rightarrow b_1 = b_2$$

Notez qu'avec cette définition, une fonction $f : A \to B$ peut n'être que *partielle*, c'est-à-dire ne pas avoir de valeur $f(a)$ définie pour certaines entrées $a \in A$.
Une fonction $f : A \to B$, définie par une relation fonctionnelle $\mathcal{R}_f \subseteq A \times B$, est :

- ***totale*** si tout élément de A a une image

  $$\forall a \in A,\ \exists b \in B,\ a\mathcal{R}_f b$$

- ***surjective*** si tout élément de B a au moins un antécédent

  $$\forall b \in B,\ \exists a \in A,\ a\mathcal{R}_f b$$

- ***injective*** si aucun élément de B n'a plus d'un antécédent

  $$\forall b \in B,\ \forall a_1, a_2 \in A,\ a_1\mathcal{R}_f b \wedge a_2\mathcal{R}_f b \Rightarrow a_1 = a_2$$

- ***bijective*** si elle est totale, surjective et injective.

Exemple : la fonction $f : \mathbb{N} \to \mathbb{N}$ définie par $f(n) = n^2$ est donnée par la relation binaire $\mathcal{R}_f = \{(n, n^2) \mid n \in \mathbb{N}\}$. Cette fonction est totale et injective. La même fonction étendue au domaine $\mathbb{Z} \to \mathbb{N}$ serait toujours totale mais plus injective, puisque $f(1) = f(-1) = 1$.

Relations binaires homogènes et graphes. Une relation binaire homogène $\mathcal{R} \subseteq E \times E$ est :

- ***symétrique*** si le fait pour deux éléments $e_1, e_2 \in E$ d'être en relation est indépendant de l'ordre dans lequel on les considère

  $$\forall e_1, e_2 \in E,\ e_1\mathcal{R}e_2 \Rightarrow e_2\mathcal{R}e_1$$

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

  $$\forall e \in E, e\mathcal{R}e$$

- ***irréflexive*** si un élément n'est jamais en relation avec lui-même

  $$\forall e \in E, \neg e\mathcal{R}e$$

Que se passe-t-il si le graphe contient des boucles ? Et des arêtes parallèles ?

Un graphe simple (S, A) définit une relation binaire homogène $\mathcal{R}$ sur $S \times S$ par la condition « $s_1\mathcal{R}s_2$ s'il existe une arête $a \in A$ entre s_1 et s_2 ». Cette relation est *symétrique* et *irréflexive*. Réciproquement, toute relation binaire homogène $\mathcal{R}$ sur un ensemble E qui est symétrique et irréflexive définit un graphe simple ayant E pour ensemble de sommets et ayant une arête entre e_1 et e_2 si et seulement si $e_1\mathcal{R}e_2$.

5 Mettre de l'ordre

5.1 Problème : ordonnancement de tâches interdépendantes

L'université Saris-Paclay a encore besoin d'aide. L'enseignant d'un cours connu sous le nom de code « OLA » a fait la liste des outils et algorithmes qu'il allait présenter au cours du semestre, et s'est rendu compte que les dépendances entre ces différents points étaient sournoisement emmêlées. Il faut maintenant trouver un ordre dans lequel organiser les séances, de sorte que chacune ne dépende que de ce qui a déjà été vu avant.

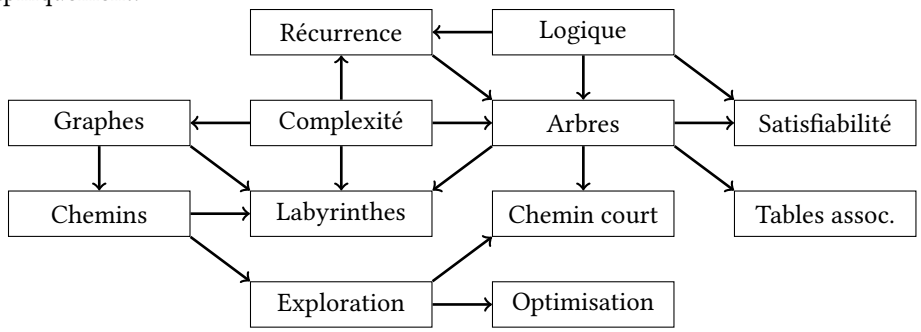
Voici un résumé de la liste, et des dépendances :

Sujet	**Dépendances** (doivent être traitées d'abord)
Graphes	Complexité
Chemins	Graphes
Récurrence	Logique, complexité
Complexité	
Labyrinthes	Graphes, arbres, chemins, complexité
Exploration	Chemins
Logique	
Arbres	Récurrence, complexité, logique
Plus court chemin	Exploration, arbres
Optimisation	Exploration
Satisfiabilité	Logique, arbres
Tables associatives	Arbres

Pouvez-vous trouver un ordre de présentation des sujets qui fasse en sorte qu'aucun ne soit abordé avant que toutes ses dépendances aient été elles-mêmes traitées ?

5.2 Modélisation : graphes orientés

On peut reprendre l'idée d'une modélisation du problème par un graphe : chaque sujet à traiter devient un sommet, et les arêtes matérialisent la relation de dépendance entre deux sujets. Différence par rapport aux graphes déjà vus : la relation de dépendance est ***orientée***. Lorsque l'on dit qu'un cours *A dépend* d'un cours *B*, cela signifie que *B* doit être programmé *avant A* : on fixe un *ordre* entre les éléments. On utilise des flèches pour représenter cela graphiquement.



Graphe orienté. La notion de ***graphe orienté*** permet d'exprimer ceci en donnant une *direction* à chaque arête. Au lieu de deux extrémités équivalentes, on a maintenant un sommet de ***départ*** (ou *source*) et un sommet d'***arrivée*** (ou *cible*). On dessine une telle arête sous la forme d'une flèche.

Pour un sommet s d'un tel graphe, le degré $\delta(s)$ se décompose en :

— un ***degré entrant*** : nombre d'arêtes dont s est l'arrivée,

— un ***degré sortant*** : nombre d'arêtes dont s est le départ.

Le degré sortant d'un sommet s correspond également au nombre d'éléments renvoyés par `voisins(`s`)`.

Chemins et cycles. Un ***chemin*** dans un graphe est une séquence C de sommets liés par des arêtes. C'est-à-dire : une séquence $C[0] \to C[1] \to C[2] \to \ldots \to C[k]$ où les $C[i]$ sont des sommets du graphe tels que pour tout $i \in [0, k[$, on a dans le graphe une arête *de* $C[i]$ *vers* $C[i+1]$. Dans un tel chemin, $C[0]$ est le ***départ***, et $C[k]$ l'***arrivée***. Le nombre k d'arêtes

Question : à quoi ressemble un chemin de longueur zéro ?

empruntées est la ***longueur*** du chemin. Note : dans un graphe orienté, les chemins doivent respecter l'orientation de chaque arête empruntée. Un ***cycle*** est un chemin dont le sommet d'arrivée est égal au sommet de départ. Un graphe ***acyclique*** est un graphe ne possédant aucun cycle.
Dans notre graphe des sujets, on a par exemple un chemin

Logique → Récurrence → Arbres → Labyrinthes

mais on n'a en revanche aucun cycle. En revanche, si l'arête entre « Complexité » et « Labyrinthes » était dans *l'autre sens*, c'est-à-dire de « Labyrinthes » vers « Complexité », alors on aurait un cycle.

Structure de données. Les structures de données que nous avons déjà vues pour les graphes sont tout à fait adaptées aux graphes orientés. Aussi bien dans une matrice d'adjacence que dans un tableau de listes d'adjacence, chaque arête non orientée entre deux sommets s et t est représentée par deux éléments : l'indication que t est un voisin de s, et celle que s est un voisin de t. Pour une arête orientée il suffit de ne garder qu'un de ces deux éléments, choisi selon la direction de l'arête. Les seules adaptations sont donc les suivantes.

— Dans l'interface `Graphe`, on précise la signification de la méthode `voisins` : un appel à `voisins(s)` énumère les sommets t pour lesquels il existe une arête orientée *de s vers t*.

— Dans la classe `GrapheMat`, on ajoute la méthode suivante, version simplifiée de `ajouteArete`.

```
public void ajouteAreteOrientee(int s, int t) { adj[s][t] = true; }
```

— Dans la classe `GrapheAdj`, on ajoute de même la méthode suivante.

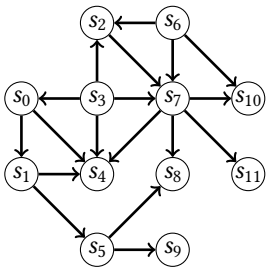
```
public void ajouteAreteOrientee(int s, int t) { adj.get(s).add(t); }
```

5.3 Algorithme : tri topologique

Le problème du ***tri topologique***, ou ***ordonnancement séquentiel***, d'un graphe orienté consiste à déterminer une séquence T contenant tous les sommets, dans un ordre tel que s'il existe une arête de s_i vers s_j, alors s_j apparaît après s_i dans T. Autrement dit, on cherche une permutation $\sigma \in \mathfrak{S}_n$ telle que s'il existe une arête de s_i vers s_j, alors $\sigma(i) < \sigma(j)$.

Principe de l'algorithme. Considérons un sommet s de degré entrant zéro, c'est-à-dire tel qu'il n'existe aucun t avec une arête $t \to s$. Il n'y a donc aucun sommet qui doive nécessairement apparaître avant s : on peut choisir de sélectionner s en premier dans notre tri topologique. Une fois s sélectionné en premier, on peut sélectionner en deuxième tout autre sommet de degré entrant nul, *ou tout sommet dont la seule arête entrante vient de s*. Autrement dit, on peut sélectionner en deuxième tout sommet de degré entrant nul dans le graphe qu'on obtiendrait en supprimant s (et les arêtes incidentes à s).

On continue ainsi à sélectionner les sommets de degré entrant nul dans des graphes de plus en plus réduits, car on ne tient plus compte des sommets qui ont déjà été sélectionnés. Dans l'exemple ci-dessous, on appelle *sommets disponibles* ces sommets qui sont prêts à être sélectionnés, car il n'ont aucune arête entrante venant d'un sommet qui n'aurait pas déjà été sélectionné.



Étape	Disponibles	Sélectionné
1	s_3, s_6	s_6
2	s_3, s_2	s_3
3	s_0, s_2	s_2
4	s_0, s_7	s_0
5	s_1, s_7	s_1
6	s_5, s_7	s_5
7	s_7, s_9	s_7
8	s_4, s_8, s_9, s_{10}, s_{11}	s_{10}
9	s_4, s_8, s_9, s_{11}	s_{11}
10	s_4, s_8, s_9	s_8
11	s_4, s_9	s_9
12	s_4	s_4

Note : à chaque étape, le choix parmi les sommets disponibles est arbitraire. L'ordre donné ici en exemple correspond au plan réel du semestre.

Code java. Dans le code java suivant, on ne modifie pas le graphe lui-même pour supprimer les sommets sélectionnés (ce serait potentiellement coûteux). À la place, on crée un tableau `degreEntrant` qui renseigne le degré entrant de chaque sommet, et on décrémente les valeurs de ce tableau pour qu'elles ne comptent plus les sommets déjà sélectionnés. La boucle principale (boucle **`while`**) parcourt le tableau des sommets sélectionnés et réalise deux choses :

— pour chaque sommet s trouvé dans ce tableau, décrémenter le degré entrant des sommets v voisins de s,

— lorsque cette opération annule le degré entrant d'un sommet v, sélection de celui-ci.

On déclare que le tri topologique a réussi lorsque ce processus permet bien de sélectionner *tous* les sommets.

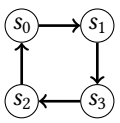
```
static int[] triTopologique(Graphe g) {
    int n = g.taille();
    // calcul du degré entrant de chaque sommet
    int[] degreEntrant = new int[n];
    for (int s : g.sommets())
        for (int v : g.voisins(s))
            degreEntrant[v] += 1;
    // initialisation avec les sommets de degré entrant nul
    int[] ordre = new int[n];
    int j = 0;
    for (int s : g.sommets())
        if (degreEntrant[s] == 0) ordre[j++] = s;
    // énumération des sommets sélectionnés
    int i = 0;
    while (i < j) {
        // degré entrant des voisins décroît, sélection si devient nul
        for (int v : g.voisins(ordre[i])) {
            degreEntrant[v] -= 1;
            if (degreEntrant[v] == 0) ordre[j++] = v;
        }
        i++;
    }
    // échec si on n'a pas sélectionné tous les sommets
    if (j < n) return null;
    return ordre;
}
```

Suivi de l'exécution de la boucle principale, sur notre graphe exemple.

		degreEntrant												
i	s	0	1	2	3	4	5	6	7	8	9	10	11	ordre $[0, j[$
		1	1	2	0	4	1	0	3	2	1	2	1	s_3, s_6
0	s_3	0	1	1	0	3	1	0	2	2	1	2	1	s_3, s_6, s_0
1	s_6	0	1	0	0	3	1	0	1	2	1	1	1	s_3, s_6, s_0, s_2
2	s_0	0	0	0	0	2	1	0	1	2	1	1	1	s_3, s_6, s_0, s_2, s_1
3	s_2	0	0	0	0	2	1	0	0	2	1	1	1	s_3, s_6, s_0, s_2, s_1, s_7
4	s_1	0	0	0	0	1	0	0	0	2	1	1	1	s_3, s_6, s_0, s_2, s_1, s_7, s_5
5	s_7	0	0	0	0	0	0	0	0	1	1	0	0	s_3, s_6, s_0, s_2, s_1, s_7, s_5, s_4, s_{10}, s_{11}
6	s_5	0	0	0	0	0	0	0	0	0	0	0	0	s_3, s_6, s_0, s_2, s_1, s_7, s_5, s_4, s_{10}, s_{11}, s_8, s_9

Note : les cinq dernières étapes ne font plus décroître les degrés, car les sommets restants s_4, s_{10}, s_{11}, s_8 et s_9 n'ont aucune arête sortante.

Cas où le tri topologique est impossible. Pour certains graphes, il n'est pas possible de trouver un tri topologique. C'est le cas notamment dès qu'un graphe contient un cycle.



5.4 Terminaison : technique du variant

La boucle principale de notre algorithme de tri topologique est une boucle **`while`**, comparant deux indices `i` et `j`. Pour que la boucle s'arrête, il faut que l'indice `i` devienne égal (ou supérieur) à l'indice `j`. Il n'est pas évident que cela arrive un jour :

— à chaque tour, `i` est incrémenté de 1,

— à chaque tour, `j` peut rester tel quel ou être incrémenté, de 1 ou plus.

Si l'indice `j` est non nul à l'origine, et est incrémenté au moins de 1 à chaque tour, il ne sera jamais rattrapé par l'indice `i` ! Nous avons besoin d'un argument plus élaboré que la simple croissance de `i` pour assurer que notre algorithme produit bien toujours un résultat en un temps fini.

En plus, si `j` devenait trop grand on échouerait à cause d'accès hors des limites du tableau `ordre`. On aimerait aussi éviter cela.

Argument intuitif : on s'attend à ce que chaque sommet du graphe ne soit ajouté qu'une fois à la liste `ordre`. La longueur de cette liste ne doit donc pas dépasser la taille n du graphe, et la boucle n'est pas censée effectuer plus de n étapes.

Argument de décroissance. Pour garantir qu'un algorithme avec une boucle s'arrête, on identifie un ***variant*** de la boucle, c'est-à-dire un nombre entier calculé en fonction des variables du programme et qui a les deux propriétés suivantes :

Ne pas confondre !
Invariant : *propriété* préservée
Variant : *quantité* décroissante

— il est positif ou nul,

— il décroît strictement à chaque tour.

Comme un nombre entier donné ne peut décroître qu'un nombre fini de fois avant de devenir négatif ou nul, on s'assure qu'une boucle dotée d'un variant ne peut pas « boucler infiniment ».

Pour notre tri topologique, on peut prendre comme variant le nombre suivant :

$$\texttt{j} - \texttt{i} + \sum_{0 \leq k < n} \texttt{degreEntrant}[k]$$

Vérifions les deux propriétés demandées.

— Ce nombre est bien positif pendant toute la durée de la boucle : d'une part la somme $\sum_{0 \leq k < n} \texttt{degreEntrant}[k]$ est une somme de nombres positifs ou nuls, et d'autre part $\texttt{j} - \texttt{i} \geq 0$ car la boucle teste elle-même `i < j`.

— D'un tour de boucle au suivant, la somme $\texttt{j} + \sum_{0 \leq k < n} \texttt{degreEntrant}[k]$ ne croît jamais, car chaque augmentation de `j` est associée à la baisse de l'un des $\texttt{degreEntrant}[k]$. À l'inverse, à chaque tour `i` augmente de 1. La valeur totale décroît donc au moins de 1.

Avec ce variant, on garantit la terminaison de la boucle sans justifier rigoureusement que chaque sommet est ajouté au plus une fois à `ordre`. On se contente ici de garantir que le nombre d'insertions dans `ordre` ne dépasse par le nombre d'arêtes du graphe. On n'a donc pas encore assuré que la taille n donnée au tableau `ordre` était suffisante.

Argument de décroissance, version plus précise. Pour comparer plus précisément `j` et n, on introduit une propriété liant `j` au nombre d'entrées inférieures ou égales à zéro dans `degreEntrant`.

$$\texttt{j} = \mathrm{card}(\{k \in [0, n[\mid \texttt{degreEntrant}[k] \leq 0\})$$

Cette propriété est préservée par la boucle **`for`** `(int v : g.voisins(ordre[i]))`. En effet, considérons un tour de cette boucle pour un sommet v.

— Si au début, $\texttt{degreEntrant}[v] > 1$, alors à la fin $\texttt{degreEntrant}[v] \geq 1$ et aucune des valeurs `j` et $\mathrm{card}(\{k \in [0, n[\mid \texttt{degreEntrant}[k] \leq 0\})$ n'a changé.

— Si au début, $\texttt{degreEntrant}[v] = 1$, alors à la fin $\texttt{degreEntrant}[v] = 0$ et les deux valeurs `j` et $\mathrm{card}(\{k \in [0, n[\mid \texttt{degreEntrant}[k] \leq 0\})$ ont augmenté de 1.

— Si au début, $\texttt{degreEntrant}[v] \leq 0$, alors à la fin $\texttt{degreEntrant}[v] \leq 0$ et aucune des valeurs `j` et $\mathrm{card}(\{k \in [0, n[\mid \texttt{degreEntrant}[k] \leq 0\})$ n'a changé (ce dernier cas ne doit pas arriver si `degreEntrant` a été correctement initialisé, mais il ne poserait pas de problème à cette propriété).

Pour assurer que notre propriété d'égalité est un invariant, il ne reste plus qu'à justifier qu'elle est vraie avant le début de la boucle. Pour cela, on remarque d'abord que l'égalité est également préservée par la boucle `while (i < j)`, puisque cette dernière n'a aucune action sur `j` ou `degreEntrant` autre que celles contenues dans la boucle `for`. Enfin, l'égalité est vraie avant le début de la boucle `while`, car la boucle précédente (`for (int s : g.sommets())`) initialise `j` précisément à la valeur demandée.

Notre invariant sur `j` en implique un autre : $\texttt{j} \leq n$. Comme en outre d'un bout à l'autre de l'algorithme $\texttt{i} \leq \texttt{j}$, on déduit un invariant sur `i` :

$$\texttt{i} \leq n$$

Avec cette propriété, on assure que la valeur $n - \texttt{i}$ est un variant de notre boucle `while` principale, et donc que celle-ci termine. On assure même que cette boucle termine après n tours au maximum.

5.5 Relations d'ordre

Une notion d'*ordre* est une manière de comparer et classer les éléments d'un ensemble.

Définition. Étant donné un ensemble E, une ***relation d'ordre*** sur E est une relation binaire homogène $\mathcal{R}$sur E qui est :

— réflexive : tout élément est comparable à lui-même

$$\forall e \in E,\ e \mathcal{R} e$$

— anti-symétrique : deux éléments distincts ne peuvent pas être comparables à la fois dans un sens et dans l'autre

$$\forall e_1, e_2 \in E,\ (e_1 \mathcal{R} e_2 \wedge e_2 \mathcal{R} e_1) \Rightarrow e_1 = e_2$$

— transitive : la comparabilité se propage de proche en proche

$$\forall e_1, e_2, e_3 \in E,\ (e_1 \mathcal{R} e_2 \wedge e_2 \mathcal{R} e_3) \Rightarrow e_1 \mathcal{R} e_3$$

Un ordre ***total*** est un ordre pour lequel tous deux éléments sont comparables (dans un sens ou dans l'autre).

$$\forall e_1, e_2 \in E,\ e_1 \mathcal{R} e_2 \vee e_2 \mathcal{R} e_1$$

Exemples

— Relation d'ordre usuelle $\leq$ sur un ensemble de nombres.

— Relation d'inclusion $\subseteq$ sur les parties d'un ensemble A.

— Relation de divisibilité $|$ sur les nombres entiers.

Ordre strict. Un ordre $\leq$ sur un ensemble E définit un ***ordre strict*** $<$ par la condition $e_1 < e_2 \iff (e_1 \leq e_2 \wedge e_1 \neq e_2)$. Cette relation est transitive, anti-symétrique et *ir*réflexive.

Plus petit élément, élément minimal. On considère un ensemble E et un ordre $\leq$ sur E. Étant donnés un sous-ensemble $A \subseteq E$ et un élément $x \in A$, on dit que :

— x est le ***plus petit élément*** de A si x est plus petit que tous les éléments de A

$$\forall a \in A,\ x \leq a$$

— x est le ***plus grand élément*** de A si x est plus grand que tous les éléments de A

$$\forall a \in A,\ a \leq x$$

Note : une partie A n'admet pas nécessairement de plus petit élément, mais dans le cas où un tel élément existe il est unique (de même pour le plus grand élément).

Étant donnés un sous-ensemble $A \subseteq E$ et un élément $x \in A$, on dit que :

— x est un élément ***minimal*** de A s'il n'existe pas dans A d'élément plus petit que x

$$\forall a \in A,\ a \leq x \Rightarrow a = x$$

— x est un élément ***maximal*** de A s'il n'existe pas dans A d'élément plus grand que x

$$\forall a \in A,\ x \leq a \Rightarrow a = x$$

Note : minimal n'est pas la même chose que *plus petit*, et un élément minimal de A n'est pas nécessairement unique (de même pour maximal/plus grand).

Quelques propriétés.

— Le plus petit élément, s'il existe, est unique.

— Le plus petit élément, s'il existe, est l'unique élément minimal.

— Si l'ordre $\leq$ est total, les conditions « être le plus petit élément de A » et « être un élément minimal de A » deviennent équivalentes.

Majorants/minorants, bornes. On considère un ensemble E, un ordre $\leq$ sur E et une partie $A \subseteq E$.

— Un élément $x \in E$ est un ***minorant*** de A s'il est plus petit que tous les éléments de A

$$\forall a \in A,\ x \leq a$$

— Un élément $x \in E$ est un ***majorant*** de A s'il est plus grand que tous les éléments de A

$$\forall a \in A,\ a \leq x$$

— La ***borne inférieure*** de A est, s'il existe, le plus grand élément des minorants de A.

— La ***borne supérieure*** de A est, s'il existe, le plus petit élément des majorants de A.

Note : les majorants, minorants et bornes de A n'existent pas forcément, et dans le cas où ils existent n'appartiennent pas nécessairement à A.

5.6 Approfondissement : ordres bien fondés

La notion d'ordre permet de donner du sens à la notion de « progression » évoquée dans notre problème de justification de l'arrêt d'un algorithme : on considérera avoir progressé dès lors que l'on obtiendra quelque chose de *strictement plus petit* vis-à-vis de l'ordre choisi. En revanche, tous les ordres n'empêchent pas une telle progression de se poursuivre indéfiniment. Cette dernière propriété caractérise les ordres dits « bien fondés ».

Ordre bien fondé : définition. Un ordre $\leq$ sur un ensemble E est ***bien fondé*** s'il n'existe pas de suite infinie strictement décroissante pour $\leq$. Autrement dit, en notant $<$ l'ordre strict associé à $\leq$, il ne peut pas exister de suite $(x_k)_{k\in\mathbb{N}}$ telle que $\forall k \in \mathbb{N},\ x_{k+1} < x_k$.

Cette propriété traduit directement la notion d'arrêt cherchée.

Caractérisation alternative Un ordre $\leq$ sur un ensemble E est bien fondé si et seulement toute partie non vide de E admet un élément minimal. Autrement écrit :

$$\forall A \subseteq E,\ \ A \neq \emptyset \Rightarrow (\exists a \in A,\ \forall x \in A,\ x \leq a \Rightarrow x = a)$$

Preuve

— Supposons $(E, \leq)$ bien fondé. Soit A une partie non vide de E.
Raisonnement par l'absurde. Supposons que A n'admette pas d'élément minimal. Autrement dit, $\forall a \in A,\ \exists a' \in A,\ a' < a$. Comme A est non vide, il existe au moins un élément $a_0 \in A$. Comme $\leq$ est bien fondé, il n'existe pas de suite infinie strictement décroissante à partir de a_0. Soit $(a_k)_{k\in[0,N]}$ une suite strictement décroissante d'éléments de A à partir de a_0, qui soit la plus longue possible. Comme $a_N \in A$ et A n'admet pas d'élément minimal, il existe $a_{N+1} \in A$ avec $a_{N+1} < a_N$. Donc $(a_k)_{k\in[0,N+1]}$ est une suite strictement décroissante dans A strictement plus longue que la précédente. Contradiction, donc A doit admettre un élément minimal.

— Supposons que toute partie non vide A de E admette un élément minimal.
Raisonnement par l'absurde. Soit $(x_k)_{k\in\mathbb{N}}$ une suite infinie strictement décroissante dans E. On note A l'ensemble des valeurs de cette suite. Cet ensemble A est non vide (il contient par exemple x_0), et admet donc un élément minimal x_k. Or $x_{k+1} < x_k$ avec $x_{k+1} \in A$, ce qui contredit la minimalité de x_k. Donc il ne peut pas exister de suite infinie strictement décroissante dans E, et l'ordre $\leq$ est bien fondé.

Exemples

- Ordre usuel $\leq$ sur $\mathbb{N}$.
- Divisibilité $|$ sur $\mathbb{Z}$.
- Inclusion $\subseteq$ sur les parties d'un ensemble *fini*.

Contre-exemples

- Ordre usuel $\leq$ sur $\mathbb{Z}$: on peut descendre indéfiniment dans les négatifs.
- Ordre usuel $\leq$ sur $\mathbb{R}^+$: on peut s'approcher indéfiniment de 0 sans jamais l'atteindre.
- Inclusion $\subseteq$ sur les parties d'un ensemble infini : on peut définir une suite infinie d'ensembles qui ont toujours moins d'éléments mais restent infinis, comme la suite $([k, \infty[)_{k\in\mathbb{N}}$.

Récurrence bien fondée On considère un ensemble E, avec un ordre bien fondé $\leq$. En notant $<$ l'ordre strict associé à $\leq$ (par définition, $x < y$ si et seulement si $x \leq y \wedge x \neq y$), on a le nouveau principe de récurrence suivant.

Pour tout prédicat P sur les éléments de E, si

1. pour tout $e \in E$, $(\forall x \in E,\ x < e \Rightarrow P(x))$ implique $P(e)$,

alors pour tout élément $e \in E$ on a $P(e)$.

Autrement dit, si l'on peut déduire $P(e)$ dès lors que l'on suppose la propriété vraie pour tout les élément strictement inférieurs à e, et ceci pour chaque e, alors la propriété P est vraie pour tous les éléments de E. C'est la même idée que le principe de récurrence forte sur les entiers.

Justification du principe de récurrence bien fondée Supposons que pour tout $e \in E$, $(\forall x \in E,\ x < e \Rightarrow P(x))$ implique $P(e)$, notons A l'ensemble des éléments de $a \in E$ tels que $P(a)$ ne soit pas vraie et montrons que cet ensemble est vide en *raisonnant par l'absurde*.

Supposons cet ensemble A non vide. Comme $\leq$ est bien fondé et A non vide, il existe un élément minimal a de A. Soit $x \in E$ tel que $x < a$. Comme a est minimal dans A, nous savons que $x \notin A$, et donc que $P(x)$ est vraie. Ceci étant vrai pour pour tous les $x < a$, on déduit $P(a)$. Contradiction avec l'appartenance de a à A. Donc l'ensemble A doit être vide, et tous les éléments de E vérifient P.

5.7 Approfondissement : combinaison d'ordres

Comment jugeons-nous les propositions suivantes ?

- $(1, 2) < (3, 4)$?
- $(1, 3) < (2, 4)$?
- $(1, 5) < (2, 3)$?
- $(2, 3) < (1, 5)$?

De manière plus générale, étant donnés deux ensembles A et B, chacun avec un ordre (on pourra les noter respectivement $\leq_A$ et $\leq_B$), on cherche à ordonner les paires de $A \times B$.

Ordre produit cartésien L'***ordre produit*** sur $A \times B$ est défini par $(a_1, b_1) \leq (a_2, b_2)$ si et seulement si (a_1, b_1) est plus petit sur les deux composantes à la fois : $a_1 \leq_A a_2 \wedge b_1 \leq_B b_2$.

Par exemple :

- $(1, 2) \leq (3, 4)$
- $(1, 3) \leq (2, 4)$
- $(1, 5)$ et $(2, 3)$ sont incomparables

Note : l'ordre produit n'est donc pas total.

Si $\leq_A$ et $\leq_B$ sont deux ordres bien fondés, alors leur produit est lui aussi bien fondé.

Justification. Si on prend une suite infinie $(a_n, b_n)_{n\in\mathbb{N}}$ strictement décroissante pour l'ordre produit, alors on obtient deux suites infinies décroissantes $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ pour les ordres $\leq_A$ et $\leq_B$. On a plus précisément, pour tout $n \in \mathbb{N}$, d'une part $a_n \geq_A a_{n+1}$ et $b_n \geq_B b_{n+1}$, et d'autre part au moins l'une des deux conditions $a_n \neq a_{n+1}$ ou $b_n \neq b_{n+1}$. Ainsi, au moins l'une des deux suites (a_n) ou (b_n) décroît strictement infiniment souvent. Cela contredit l'hypothèse selon laquelle l'ordre correspondant ($\leq_A$ ou $\leq_B$) est bien fondé.

Ordre produit lexicographique Le ***produit lexicographique*** des deux ordres $\leq_A$ et $\leq_B$ consiste à comparer d'abord la première composante, puis à ne tenir compte de la deuxième composante qu'en cas d'égalité sur la première : on a $(a_1, b_1) \leq (a_2, b_2)$ si et seulement si $a_1 <_A a_2 \vee (a_1 = a_2 \wedge b_1 \leq_B b_2)$.

C'est selon ce principe que l'on compare deux mots dans le dictionnaire (l'ordre lexicographique est aussi appelé ***ordre du dictionnaire***).

- $(1, 2) \leq (3, 4)$
- $(1, 3) \leq (2, 4)$
- $(1, 5) \leq (2, 3)$

Note : l'ordre lexicographique est total.

Si $\leq_A$ et $\leq_B$ sont des ordres bien fondés, alors leur produit lexicographique est bien fondé également.

Justification. On va utiliser la caractérisation alternative des ordres bien fondés : toute partie non vide contient au moins un élément minimal. Soit donc $C \subseteq A \times B$ un ensemble non vide arbitraire de paires d'un élément de A et d'un élément de B. On note C_A l'ensemble des éléments de A apparaissant dans une paire de C. Formellement : $C_A = \{a \in A \mid \exists b \in B,\ (a, b) \in C\}$. Comme C n'est pas vide, C_A contient également au moins un élément. L'ordre $\leq_A$ étant bien fondé on en déduit qu'il existe un élément minimal a_0 pour $\leq_A$ dans C_A. Notons maintenant C_B l'ensemble des éléments de B apparaissant associés à a_0 dans l'ensemble C. Formellement : $C_B = \{b \in B \mid (a_0, b) \in C\}$. Cet ensemble C_B est à nouveau non vide, puisque $a_0 \in C_A$ et par définition de C_A il existe au moins un $b \in B$ tel que $(a_0, b) \in C$. L'ordre $\leq_B$ étant bien fondé on en déduit qu'il existe un élément minimal b_0 pour $\leq_B$ dans C_B. Il ne reste plus qu'à montrer que la paire (a_0, b_0) est un élément minimal de C pour l'ordre lexicographique. Soit donc $(a, b) \in C$, telle que $(a, b) \leq (a_0, b_0)$. Par définition de l'ordre lexicographique $\leq$, on a deux cas.

- Soit $a < a_0$, ce qui contredirait la minimalité de a_0 car $a \in C_A$.
- Soit $a = a_0$ et $b \leq_B b_0$. Alors $b \in C_B$, et par minimalité de b_0 on a donc $b = b_0$.

On a donc nécessairement $(a, b) = (a_0, b_0)$, et la paire (a_0, b_0) est bien minimale. Donc C admet un élément minimal, et ainsi l'ordre lexicographique est bien fondé.

5.8 Approfondissement : critère d'existence d'un tri topologique

On constate que la présence d'un cycle dans le graphe empêche tout tri topologique. Raisonnons par l'absurde : prenons un graphe contenant un cycle

$$t_0 \to t_1 \to t_2 \to \ldots \to t_k \to t_0$$

et supposons que ce graphe admet un tri topologique. Nécessairement, le tri topologique fait intervenir tous les sommets de ce cycle. Notons t_i celui qui apparaît en premier. En particulier, il apparaît *avant* t_{i-1}, ce qui contredit l'existence d'une arête $t_{i-1} \to t_i$ (dans le cas où $i = 0$, on remplace dans ce raisonnement $i - 1$ par k).

Critère d'existence d'un tri topologique. Inversement, on peut démontrer le fait suivant.

Tout graphe sans cycle admet un tri topologique.

On commence par démontrer le lemme suivant : *tout graphe acyclique non vide admet au moins un sommet de degré entrant 0.*

Preuve par l'absurde. Soit G un graphe acyclique non vide tel que tous les sommets de G ont un degré entrant strictement positif. On va démontrer par récurrence que $\forall n \in \mathbb{N}$, le graphe G admet un chemin de longueur n :

- Cas de base : G étant non vide il contient un sommet s, et un chemin de longueur 0 de s à s.
- Itération : Soit $n \in \mathbb{N}$ tel que G admette un chemin de longueur n, et soit $t_0 \to t_1 \to \ldots \to t_n$ un tel chemin de longueur n dans G. Le sommet de départ t_0 de ce chemin a un degré entrant non nul, il existe donc une arête $s \to t_0$ ayant t_0 pour arrivée. On peut donc construire un chemin $s \to t_0 \to t_1 \to \ldots \to t_n$ de longueur $n + 1$ dans G.

Donc $\forall n \in \mathbb{N}$, le graphe G admet un chemin de longueur n. En particulier, en notant N le nombre de sommets de G, on sait que G admet un chemin de longueur $N+1$. Par le principe des tiroirs il existe un sommet s de G par lequel ce chemin passe deux fois. On extrait du chemin la séquence comprise entre les deux premières occurrences de s pour obtenir un chemin de s à s, c'est-à-dire un cycle. Contradiction avec l'hypothèse selon laquelle G est acyclique. Donc G admet nécessairement un sommet de degré entrant 0.

Retour au théorème : *tout graphe orienté acyclique admet un tri topologique.*

Preuve par récurrence sur le nombre de sommets du graphe. On note $P(n)$ la propriété : « tous les graphes acycliques à n sommets admettent un tri topologique ».

- Initialisation (preuve de $P(0)$) : un graphe vide est trié topologiquement par la séquence vide.
- Hérédité (preuve de $\forall n \in \mathbb{N},\ P(n) \Rightarrow P(n+1)$). Soit n tel que $P(n)$ soit vraie, et soit G un graphe acyclique à $n+1$ sommets $\{s_0, \dots, s_n\}$. Par notre lemme, G admet un sommet s_i de degré entrant 0. Le sous-graphe G' obtenu en retirant de G ce sommet s_i et ses arêtes incidentes a n sommets, et est de plus acyclique (supposons qu'il existe un cycle dans G', alors ce cycle existerait également dans G ; or G est acyclique : contradiction). On peut donc appliquer l'hypothèse de récurrence à G' pour obtenir un tri topologique de G', c'est-à-dire une séquence $t_0, t_1, \dots, t_{n-1}$ des sommets $\{s_0, \dots, s_n\} \setminus \{s_i\}$ compatible avec l'orientation des arêtes. Alors la séquence $s_i, t_0, t_1, \dots, t_{n-1}$ obtenue en ajoutant le sommet s_i en tête de la précédente est un tri topologique du graphe G complet.

  En effet, soit une arête $s \to t$ quelconque de G, montrons que s apparaît bien avant t dans la séquence $s_i, t_0, t_1, \dots, t_{n-1}$.

  - Si ni s ni t n'est égal à s_i, alors l'arête $s \to t$ apparaît dans le sous-graphe G', et par hypothèse les sommets s et t apparaissent dans le bon ordre dans le tri topologique $t_0, t_1, \dots, t_{n-1}$ de G'.
  - Si $s = s_i$, alors $s = s_i$ apparait bien avant $t = t_j$, dans la séquence $s_i, t_0, t_1, \dots, t_{n-1}$.
  - Il n'est pas possible que $t = s_i$, car l'existence de l'arête $s \to s_i$ contredirait l'hypothèse selon laquelle le sommet s_i a un degré entrant 0.

6 Trouver la voie

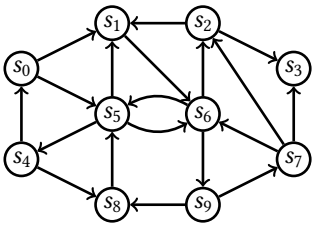
6.1 Problème : recherche de chemin

On se donne un graphe, un sommet de départ, et un sommet cible. Question : le sommet cible est-il accessible à partir du sommet de départ ? C'est-à-dire : existe-t-il un chemin allant du sommet de départ ou sommet cible, en suivant les arêtes du graphe ? Si oui, peut-on indiquer un tel chemin ? Et si plusieurs chemins sont possibles, peut-on privilégier les chemins les plus courts ? Dans ce chapitre, nous cherchons donc à *explorer* un graphe.

6.2 Algorithme : parcours en profondeur

Pour parcourir un graphe, on peut suivre une stratégie très simple : avancer dans une direction quelconque jusqu'à aboutir à un cul-de-sac, puis revenir sur ses pas jusqu'au dernier embranchement et choisir une nouvelle direction (c'est-à-dire une direction qui n'a pas encore été explorée).

On peut voir ceci comme une stratégie récursive : pour explorer un graphe à partir d'un sommet s, on considère tour à tour toutes les arêtes $s \to t_i$, et on explore récursivement à partir de chacune des cibles t_i prises à tour de rôle. Attention cependant : il est possible lors de cette exploration récursive de retourner à un sommet à partir duquel l'exploration a déjà été faite, voire de retourner au point de départ. Dans ce cas, on ne veut pas refaire l'exploration déjà faite, et encore moins tourner en rond. On mémorise donc les sommets déjà vus, afin de ne pas les explorer à nouveau.



Exemple d'exploration. On prend pour point de départ le sommet s_6.

1. Explorer s_6, puis s_2, puis s_1.
2. Le seul successeur possible est s_6, déjà exploré, revenir au sommet précédent s_2 et choisir une autre voie. Explorer s_3.
3. Aucun successeur, revenir au précédent s_2. Aucune voie inexplorée restante, revenir au précédent s_6 et choisir une autre voie.
4. Explorer s_5 puis s_4 (note : le successeur s_1 de s_5 a déjà été exploré), puis s_0.
5. Les seuls successeurs possibles sont s_1 et s_5, déjà explorés, revenir au sommet précédent s_4 et choisir une autre voie. Explorer s_8.
6. Unique successeur s_5 déjà exploré, revenir au sommet précédent s_4. Aucune voie inexplorée restante, revenir encore au précédent s_5. Aucune voie inexplorée restante, revenir encore au précédent s_6 et choisir une autre voie. Explorer s_9, puis s_7.
7. Aucun successeur non exploré, revenir au précédent s_9. Aucune voie inexplorée restante, revenir au précédent puis s_6. Aucune voie inexplorée non plus, et pas non plus de précédent restant. Arrêt.

Code java. Pour marquer les sommets déjà explorés, on peut utiliser un tableau `vu` contenant un booléen par sommet du graphe, et tel que `vu[i]` vaut **`true`** si et seulement si s_i a effectivement été rencontré. Alors, avant tout appel récursif on peut vérifier si le sommet considéré n'a pas déjà été visité. La fonction principale `dfs` initialise ce tableau de booléens, puis appelle la fonction récursive `explore` réalisant notre stratégie d'exploration. À la fin le tableau `vu` indique, pour chaque numéro i, si le sommet s_i est accessible par un chemin à partir de la source `s`.

```
private static boolean[] vu;

private static void explore(Graphe g, int s) {
    vu[s] = true;
    for (int v : g.voisins(s)) {
        if (!vu[v]) explore(g, v);
    }
}

static boolean[] dfs(Graphe g, int s) {
    vu = new boolean[g.taille()];
    explore(g, s);
    return vu;
}
```

Voici le détail de l'exécution de `dfs` sur l'exemple précédent. On note `explore` k pour un appel récursif sur le sommet s_k, et `ignore` k lorsque le sommet s_k est examiné dans une énumération de voisins, mais ignoré car déjà marqué.

	Sommets vus									
Action	0	1	2	3	4	5	6	7	8	9
`explore 6`							✓			
`+-- explore 2`			✓				✓			
`|  +-- explore 1`		✓	✓				✓			
`|  |  +-- ignore 6`		✓	✓				✓			
`|  +-- explore 3`		✓	✓	✓			✓			
`+-- explore 5`		✓	✓	✓		✓	✓			
`|  +-- ignore 1`		✓	✓	✓		✓	✓			
`|  +-- explore 4`		✓	✓	✓	✓	✓	✓			
`|  |  +-- explore 0`	✓	✓	✓	✓	✓	✓	✓			
`|  |  |  +-- ignore 1`	✓	✓	✓	✓	✓	✓	✓			
`|  |  |  +-- ignore 5`	✓	✓	✓	✓	✓	✓	✓			
`|  |  +-- explore 8`	✓	✓	✓	✓	✓	✓	✓		✓	
`|  |  |  +-- ignore 5`	✓	✓	✓	✓	✓	✓	✓		✓	
`|  +-- ignore 6`	✓	✓	✓	✓	✓	✓	✓		✓	
`+-- explore 9`	✓	✓	✓	✓	✓	✓	✓		✓	✓
`  +-- explore 7`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`  |  +-- ignore 2`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`  |  +-- ignore 3`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

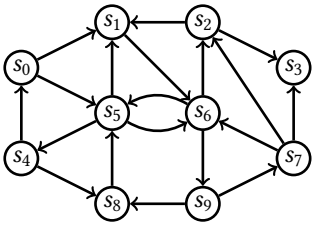
Ce mode d'exploration est appelé ***parcours en profondeur***, et est caractérisé comme suit : on s'avance aussi loin que possible sur un chemin donné, pour ne revenir sur nos pas qu'une fois un cul-de-sac atteint. Ceci est lié à l'emboîtement des appels récursifs : un appel donné à `explore(g, s)` ne se termine qu'une fois que toute l'exploration à partir de `s` est faite.

Question : que se passerait-il si l'instruction `vu[s] = true;` *était placée à la fin de l'exploration plutôt qu'au début ?*

6.3 Algorithme : parcours en largeur

Partant d'un sommet s donné, le parcours en profondeur consiste à choisir *une* voie et à la suivre jusqu'au bout avant de considérer les autres voies qui étaient possibles. Alternativement, on peut vouloir suivre toutes les voies en parallèle, en progressant en cercles concentriques autour de s : d'abord tous les voisins immédiats, puis les sommets accessibles depuis l'ensemble de ces voisins, puis les sommets accessibles depuis l'ensemble de ces suivants, etc. On parle ici de ***parcours en largeur***.

Exemple d'exploration. On part du sommet s_6.



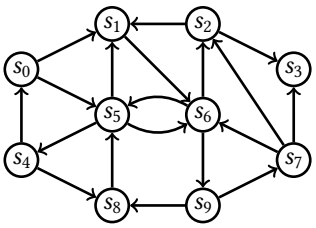
1. On regarde la source s_6. Ses voisins immédiats sont s_2, s_5 et s_9.
2. On regarde à tour de rôle s_2, s_5 et s_9, et on découvre les nouveaux sommets s_1, s_3, s_4, s_7 et s_8.
3. On regarde à tour de rôle les cinq précédents, et on découvre encore un nouveau sommet s_0.
4. Après s_0, on ne trouve plus de sommets non encore découverts : arrêt.

Code java. Il n'est plus question ici d'exploration récursive. L'analyse d'un sommet s du k-ème cercle consiste à observer ses voisins v, et à les enregistrer comme devant être analysés à l'étape $k+1$ (du moins, ceux des voisins qui n'ont pas déjà été vus). En pratique, il n'est pas nécessaire de matérialiser la transition entre les différents « cercles » : on stocke les sommets du k-ème et du $k+1$-ème cercle dans une même *file* de sommets en attente, et on traite un par un les sommets pris dans cette file. La discipline de file, dite *fifo* (« first in, first out »), assure que les premiers sommets analysés sont ceux qui ont été enregistrés en premier. Alors, tous les sommets du k-ème cercle seront analysés avant tous les sommets du $k+1$-ème cercle, eux-mêmes analysés avant tous les sommets du $k+2$-ème cercle, etc. À nouveau, la fonction renvoie le tableau de booléens identifiant les sommets accessibles depuis la source `s`.

Dans le code proposé ici, on remplace le tableau de booléens `vu` par un tableau d'entiers `dist`, tel que `dist[`s`]` vaut `-1` pour un sommet s non visité, et k, pour un sommet visité du k-ème cercle. On obtient donc une information plus précise : pas seulement d'accessibilité, mais de distance par rapport à la source.

```
static int[] bfs(Graphe g, int s) {
    int[] dist = new int[g.taille()];
    for (int t : g.sommets()) dist[t] = -1;
    Queue<Integer> enAttente = new ArrayDeque<>();
    dist[s] = 0;
    enAttente.add(s);
    while (!enAttente.isEmpty()) {
        int t = enAttente.remove();
        for (int v : g.voisins(t)) {
            if (dist[v] < 0) {
                dist[v] = dist[t] + 1;
                enAttente.add(v);
            }
        }
    }
    return dist;
}
```

Voici le détail de l'exécution de `bfs` sur l'exemple précédent. La file apparaît comme une ligne dans laquelle les éléments sont ajoutés par la droite et retirés par la gauche. Note : l'ordre dans lequel on considère les successeurs d'un sommet est a priori arbitraire.



										dist									
t	**voisins**	**enAttente**								0	1	2	3	4	5	6	7	8	9
		6								–	–	–	–	–	–	0	–	–	–
6	**2**, **5**, **9**		**2**	**5**	**9**					–	–	1	–	–	1	0	–	–	1
2	**1**, **3**			5	9	**1**	**3**			–	2	1	2	–	1	0	–	–	1
5	1, **4**, 6				9	1	3	**4**		–	2	1	2	2	1	0	–	–	1
9	**7**, **8**					1	3	4	**7**	**8**	–	2	1	2	2	1	0	2	2	1
1	6						3	4	7	8	–	2	1	2	2	1	0	2	2	1
3	∅							4	7	8	–	2	1	2	2	1	0	2	2	1
4	**0**, 8								7	8	**0**	3	2	1	2	2	1	0	2	2	1
7	6, 2, 3									3	0	3	2	1	2	2	1	0	2	2	1
8	5										0	3	2	1	2	2	1	0	2	2	1
0	1, 5											3	2	1	2	2	1	0	2	2	1

6.4 Comparaison des deux parcours

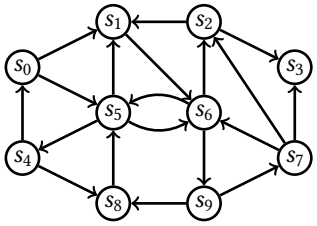
On peut ramener les parcours en profondeur et en largeur a une structure commune, mais l'un et l'autre n'explorent pas les sommets dans le même ordre.

Parcours en profondeur, sans récurrence. On peut obtenir une autre manière d'écrire un parcours en profondeur en prenant le code du parcours en largeur, et en remplaçant la file d'attente par une *pile*. La discipline de pile, dite *lifo* (« last in, first out »), fait que le premier sommet analysé est celui qui a été enregistré en dernier. Autrement dit, on poursuit d'abord dans la direction ouverte par le sommet courant, avant de revenir aux autres sommets du même cercle.

```
static boolean[] dfs(Graphe g, int s) {
    boolean[] vu = new boolean[g.taille()];
    Deque<Integer> enAttente = new ArrayDeque<>();
    vu[s] = true;
    enAttente.push(s);
    while (!enAttente.isEmpty()) {
        int t = enAttente.pop();
        for (int v : g.voisins(t)) {
            if (!vu[v]) {
                vu[v] = true;
                enAttente.push(v);
            }
        }
    }
    return vu;
}
```

Exemple d'exécution sur l'exemple précédent. La pile apparaît comme une ligne dans laquelle les éléments sont ajoutés et retirés par la droite. Note : l'ordre dans lequel on considère les successeurs d'un sommet est a priori arbitraire.

Question : que se passerait-il si on remplaçait les deux instructions `vu[s] = true;` *et* `vu[v] = true;` *par une unique instruction* `vu[t] = true;` *placée juste après* `s = enAttente.pop()` *?*

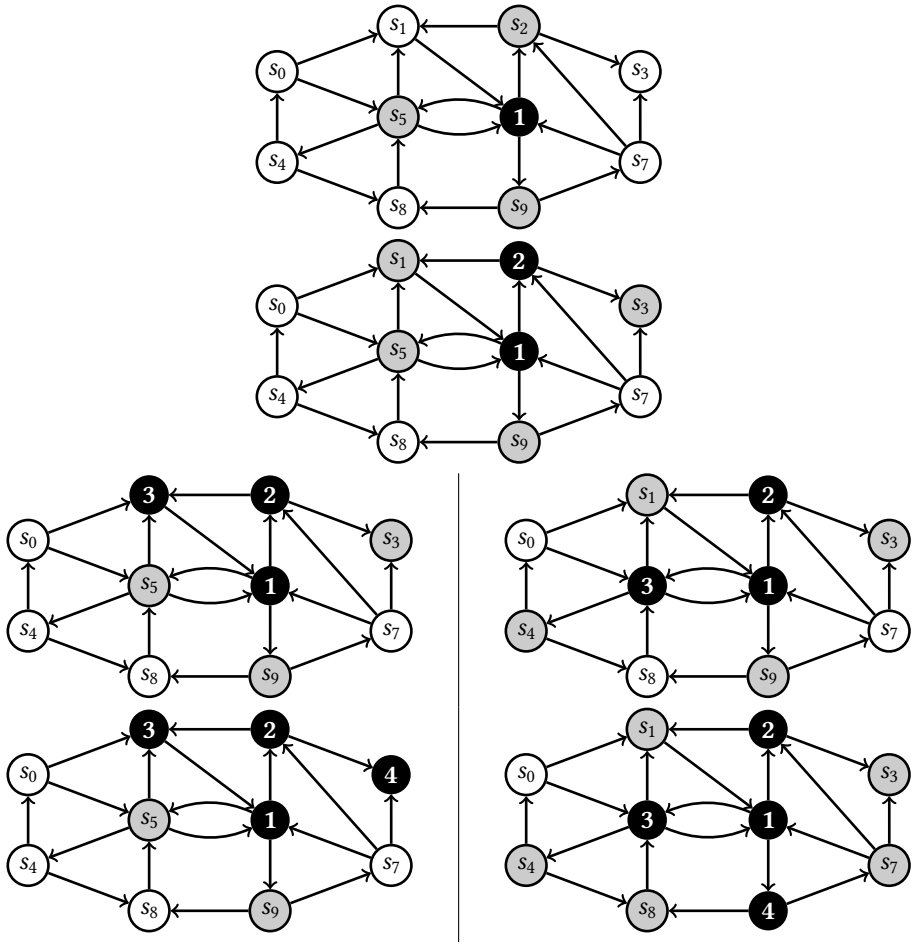


t	voisins	enAttente			
		6			
6	**9**, **5**, **2**	**9**	**5**	**2**	
2	**3**, **1**	9	5	**3**	**1**
1	6	9	5	3	
3	∅	9	5		
5	6, **4**, 1	9	4		
4	**8**, **0**	9	**8**	**0**	
0	5, 1	9	8		
8	5	9			
9	8, **7**	**7**			
7	6, 3, 2				

Note : dans la version récursive, une « pile » était bien présente. Mais où ?

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 pas 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.

On peut observer que le parcours en profondeur (à gauche) visite en troisième et en quatrième deux sommets qui étaient accessibles depuis le deuxième sommet visité (mais pas depuis la source). À l'inverse, le parcours en largeur (à droite) explore en premier les trois voisins immédiats de la source.



6.5 Reconstruction de chemins.

Une petite modification de nos algorithmes de parcours permet de reconstruire un chemin de la source vers n'importe quel sommet accessible. On modifie pour cela le tableau de booléens `vu` en un tableau de sommets `pred` tel que :

- pour tout sommet s_i non vu, `pred[`i`]` vaut `-1`,
- pour tout sommet s_i vu, `pred[`i`]` contient le numéro du sommet s_j depuis lequel on a vu s_i,
- cas particulier pour la source s_{i_0} elle-même : on initialise `pred[`i_0`]` $= i_0$.

Ainsi, lorsque `pred[`i`]` est un entier positif $j \neq i$, on sait qu'on a une arête $s_j \to s_i$, *et que le sommet s_j est lui-même accessible depuis la source.* En outre, seul le sommet source vérifie `pred[`i`]` $= i$. À l'inverse, `pred[`i`]` vaut `-1` exactement dans les cas où `vu[`i`]` aurait valu **`false`**. On ajoute pour finir une fonction qui, à partir de ce tableau des prédécesseurs, reconstruit le chemin complet. Cette adaptation s'applique aussi bien au parcours en profondeur qu'au parcours en largeur. Voici la version basée sur `bfs`.

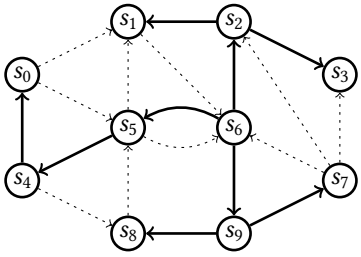
```
public static int[] bfs(Graphe g, int s) {
    int[] pred = new int[g.taille()];
    for (int t : g.sommets()) pred[t] = -1;
    Queue<Integer> enAttente = new ArrayDeque<>();
    pred[s] = s;
    enAttente.add(s);
    while (!enAttente.isEmpty()) {
        int t = enAttente.remove();
        for (int v : g.voisins(t)) {
            if (pred[v] < 0) {
                pred[v] = t;
                enAttente.add(v);
            }
        }
    }
    return pred;
}

public static List<Integer> chemin(Graphe g, int s, int t) {
    int[] pred = bfs(g, s);
    LinkedList<Integer> chemin = new LinkedList<>();
    chemin.add(t);
    while (t != s) {
        t = pred[t];
        chemin.addFirst(t);
    }
    return chemin;
}
```

Dans le tableau de prédécesseurs fourni par un tel parcours, la source n'a pas de prédécesseur, les sommets du premier cercle ont pour prédécesseur la source, les sommets du deuxième cercle ont pour prédécesseur un sommet du premier cercle, etc. Pour notre graphe exemple, et une exploration en largeur des sommets dans l'ordre 6, 2, 5, 9, 1, 3, 4, 7, 8, 0, le tableau des prédécesseurs serait celui-ci.

0	1	2	3	4	5	6	7	8	9
4	2	6	2	5	6	6	9	9	6

On peut également visualiser ceci par une relation « prédécesseur », relation binaire sur les sommets du graphe qui ne retient que les arêtes qui ont réellement été utilisées lors du parcours.



6.6 Approfondissement : analyse du parcours en profondeur

On considère ici l'algorithme `dfs` donné page 40, c'est-à-dire l'exploration récursive.

Complexité. Considérons un graphe avec n sommets et k arêtes.

Première remarque : la fonction `explore` est appelée au plus une fois par sommet du graphe. En effet, elle n'est appelée que sur des sommets s vérifiant `vu`$[s]$ = `false`, et modifie cette valeur en `true` avant toute autre chose. Une fois la première instruction de l'appel réalisée elle ne peut donc plus être appelée une deuxième fois sur le même sommet.

Dans chaque appel à `explore` on a les opérations suivantes :

- modification de `vu` pour le sommet courant,
- énumération des voisins,
- consultation de `vu` pour chaque voisins,
- éventuels appels récursifs.

Le coût propre d'un appel, ne tenant compte que des opérations de l'appel lui-même et pas de ses sous-appels récursifs, est donc proportionnel au nombre de voisins du sommet (son degré).

L'ordre de grandeur maximum du coût total est donc donné par le nombre n de sommets (pour les appels récursifs) et le nombre k d'arêtes (pour le cumul des voisins énumérés dans chaque appel). D'où une complexité $O(n + k)$.

Le coût réel peut être inférieur dans le cas où une grande part des sommets *n'est pas* accessible depuis la source.

Spécification et correction. Spécification : l'algorithme `dfs` prend en entrée un graphe g, et un sommet s valide de g, et renvoie un tableau A tel que pour tout sommet t de G, on a $A[t]$ = `true` si et seulement si t est accessible depuis s par un chemin de g.

Montrons, par récurrence forte sur le nombre total d'appels récursifs déclenchés, qu'un appel `explore(`g`, `s`)` ne marque que des sommets accessibles depuis s.

- Cas de base : aucun appel récursif. Alors le seul sommet marqué est s lui-même, qui est bien accessible depuis s par le chemin vide.
- Cas héréditaire : les sommets marqués lors de l'appel `explore(`g`, `s`)` sont s lui-même, et les sommets marqués par les appels récursifs immédiats `explore(`g`, `v_i`)` effectués pour un certain nombre de voisins $\{v_1, \dots, v_k\}$ de s. Ces appels récursifs immédiats sont inclus dans l'appel principal : chacun déclenche un nombre total d'appels récursifs inférieur d'au moins 1 au total de l'appel principal, et on peut leur appliquer l'hypothèse de récurrence. Ainsi, chaque appel `explore(`g`, `v_i`)` ne marque que des sommets accessibles depuis v_i. Or, un sommet accessible depuis v_i est également accessible depuis s, puisqu'on a une arête $s \to v_i$.

Montrons que, pour tout sommet t tel qu'il existe un chemin $s \to \dots \to t$, le sommet t est marqué après l'appel `explore(`g`, `s`)`. On le montre par récurrence sur la longueur du chemin.

- Cas de base, pour la longueur 0 : cela signifie que $t = s$, et ce sommet est bien marqué immédiatement.
- Pour un chemin de longueur $n+1$, on décompose en $s \to \dots \to u \to t$ où le chemin de s à u est de longueur n. Par hypothèse de récurrence, u est bien marqué, ce qui signifie qu'un appel `explore(`g`, `u`)` a été déclenché. Lors de cet appel le sommet t, voisin de u, a été examiné. Alors soit t était déjà marqué, soit on a eu un appel `explore(`g`, `t`)` qui a bien marqué le sommet.

Note : pour être parfaitement rigoureux dans les hypothèses de cette preuve, on suppose que les sommets ne peuvent être marqués que *par la fonction* `explore`.

Avec ces deux points, on a bien montré que l'appel `explore(g, s)` réalisé dans la fonction `dfs` marque bien exactement les sommets accessibles depuis `s`.

6.7 Approfondissement : analyse du parcours en largeur

On considère ici l'algorithme `bfs` donné page 42, c'est-à-dire l'exploration avec une file *fifo*.

Complexité. Considérons un graphe avec n sommets et k arêtes.

Remarquons d'abord que chaque sommet ne peut être ajouté qu'une seule fois à la file `enAttente`. En effet, cet ajout est soumis à un test préalable de distance non définie, et la distance est justement définie au moment même où le sommet est ajouté, empêchant tout nouvel ajout du même sommet. Remarquons en passant que, de même, une distance définie dans le tableau `dist` n'est jamais modifiée par la suite.

Chaque tour de la boucle **`while`** traite un nouveau sommet de la file `enAttente`. On a donc au maximum n tours, pour chacun des n sommets. Le coût d'un tour de boucle donné est proportionnel au nombre de voisins du sommet considéré. Le coût total est donc $O(n + k)$, comme pour le parcours en profondeur.

Spécification et correction. Spécification : l'algorithme `bfs` prend en entrée un graphe g et un sommet s valide de g, et renvoie un tableau d'entiers D tel que pour tout sommet t de g, on a $D[t] \geq 0$ si et seulement si t est accessible depuis s et à distance $D[t]$ (c'est-à-dire que le plus petit nombre d'arêtes d'un chemin de s à t est $D[t]$), et $D[t] = -1$ sinon.

Notons d_t la distance de la source au sommet t, c'est-à-dire le plus petit nombre d'arêtes d'un chemin de s vers t, en posant $d_t = \infty$ lorsque t n'est pas atteignable. Pour un état donné de la file `enAttente`, notons d la distance `dist`$[t]$ renseignée pour le premier sommet de la file, si la file est non vide. La correction est obtenue à l'aide des invariants suivants pour la boucle **`while`** de l'algorithme.

1. La file `enAttente` est constituée :
   - d'abord d'une séquence de sommets à distance d,
   - puis d'une séquence de sommets à distance $d + 1$, qui sont exactement les sommets à distance $d + 1$ voisins des sommets à distance d absents de la première partie.
2. Tout sommet t à distance $d_t \leq d$ ou qui est présent dans la file `enAttente` est tel que `dist`$[t] = d_t$.
3. Tour sommet t à distance $d_t > d$ et qui n'est pas présent dans la file `enAttente` est tel que `dist`$[t] = -1$.

Ces propriétés sont bien valides avant le premier tour de boucle : la file `enAttente` contient alors exclusivement le sommet `s`, qui est l'unique sommet à distance zéro de lui-même.

Supposons les propriétés vraies au début d'un tour de boucle. L'algorithme considère alors le sommet t en tête de la file `enAttente`, qui par définition est à distance d de la source. On énumère ensuite chaque voisin v de t.

- Si v est tel que `dist`$[v] \geq 0$, alors l'algorithme ne fait rien.
- Sinon, par hypothèse on a $d_v > d$, et l'algorithme définit `dist`$[v] = d + 1$ et ajoute v à la file. Cette action est correcte, car on a bien un chemin $s \to \ldots \to t \to v$ de longueur $d + 1$ de la source vers v.

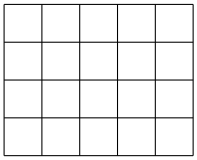
À la fin du tour, t a été retiré de la file, et tous les voisins v de t tels que $d_v = d + 1$ qui n'étaient pas déjà dans la file y ont été ajoutés. Remarque : si t était le dernier sommet de la file à distance d, alors la file est maintenant constituée exactement des sommets à distance $d + 1$.

L'algorithme s'arrête lorsque la file est vide. En notant d la distance du dernier sommet traité, cela signifie que le graphe ne contient aucun sommet à distance $d + 1$, et donc aucun sommet à une distance $> d$. Ainsi, tout sommet d à distance d_t finie est bien tel que `dist`$[t] = d_t$.

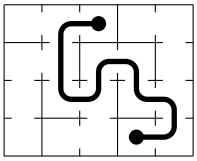
7 Se perdre

7.1 Problème : création d'un labyrinthe

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

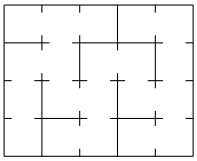


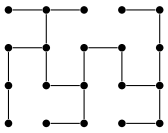
On souhaite ouvrir des portes dans certaines des cloisons, de sorte à former un *labyrinthe parfait* : on veut que quel que soit le choix d'une salle de départ et d'une salle d'arrivée il existe un unique itinéraire permettant d'aller de l'une à l'autre.



Note : quand on mentionne un « unique itinéraire » ici, on écarte implicitement les itinéraires qui contiendraient des rebroussements.

Un tel labyrinthe peut être vu comme un graphe, dont les sommets sont les salles, et les arêtes sont les portes entre deux salles. Ce graphe est non orienté.





Un « itinéraire » entre deux salles du labyrinthe est un chemin entre les sommets correspondants. L'existence hypothétique de plusieurs chemins entre deux sommets signalerait la présence d'un cycle dans le graphe. On peut donc reformuler notre objectif : partant d'un ensemble de sommets, créer un graphe *connexe* (tous les sommets peuvent être reliés deux à deux par des chemins) et *sans cycle*, en sélectionnant des arêtes parmi un ensemble d'arêtes autorisées (ici, une arête doit correspondre à une porte entre deux salles géographiquement adjacentes).

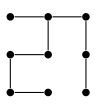
On propose de suivre la stratégie suivante : énumérer toutes les cloisons dans un ordre aléatoire, et pour chacune, y ouvrir une porte si cela ne fait pas apparaître de cycle. On pourrait imaginer une réalisation naïve de cette stratégie, en parcourant le graphe à la recherche d'un cycle à chaque étude d'une potentielle nouvelle arête : on aurait un parcours de coût linéaire en le nombre de salles, répété pour chaque arête potentielle. *Nous allons voir à la place une structure, qui permet de détecter en temps quasiment constant si l'ajout d'une arête crée un cycle.*

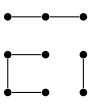
7.2 Composantes connexes d'un graphe

Connexité. Un graphe G non orienté est ***connexe*** si tous les sommets de G peuvent être reliés deux à deux par un chemin.

$$\forall s, s' \in G, \exists p \in \mathsf{chemins}(G), p \ : \ s \to s'$$

Visuellement, un graphe connexe est un graphe « en un seul morceau » (à gauche), tandis qu'un graphe non connexe est un graphe comportant des parties isolées les unes des autres (à droite).





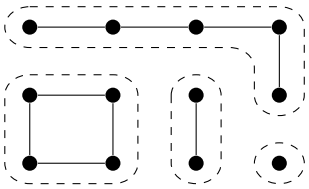
Ces « parties » sont formalisées par la notion de *composante connexe* que nous allons maintenant détailler.

Composante connexe. Dans un graphe G non orienté, une ***composante connexe*** est un sous-ensemble C *non vide* des sommets de G tel que :

1. tous les sommets de C sont connectés deux à deux par des chemins,
2. aucun sommet de C n'est connecté à un sommet hors de C.

Remarque : le critère 2 implique que les chemins du critère 1 utilisent exclusivement des sommets de C.

Visuellement, les composantes connexes sont les différents « morceaux » d'un graphe qui ne serait pas lui-même connexe.



Formellement, on définit une composante connexe de G comme un *sous-graphe connexe maximal* de G. Précisons le vocabulaire de cette définition.

— Un ***sous-graphe*** d'un graphe $G = (S, A)$ est un graphe $G' = (S', A')$ où S' est un sous-ensemble des sommets de G, et A' est l'ensemble des arêtes de G dont les extrémités sont dans S'.

$$\begin{cases} S' \subseteq S \\ A' = \{a \in A \mid \exists s, s' \in S', a : s \to s'\} \end{cases}$$

Remarque : pour former un sous-graphe, on peut choisir un sous-ensemble $S' \subseteq S$ arbitraire. En revanche, une fois S' choisi, l'ensemble A' d'arêtes est fixé (*toutes* les arêtes de G liées aux sommets choisis).

— Dans cette définition, ***maximal*** est utilisé relativement à l'ordre d'inclusion sur les sommets : la clause de maximalité indique donc qu'un sous-graphe connexe de G contenant tous les sommets d'une composante C ne peut être que C elle-même. Autrement dit : si C est une composante connexe de G, alors un sous-graphe $C' \subseteq G$ dans lequel C est inclus au sens strict ne peut pas être connexe.

$$\begin{cases} \forall s, s' \in C, \exists p \in \mathsf{chemins}(C), p : s \to s' & \text{(connexité)} \\ \forall C', C \subsetneq C' \Rightarrow \exists s, s' \in C', \neg\exists p \in \mathsf{chemins}(C'), p : s \to s' & \text{(maximalité)} \end{cases}$$

Les composantes connexes d'un graphe ont un certain nombre de propriétés utiles. En particulier, les composantes connexes d'un graphe G forment une ***partition*** de G :

— tout sommet de G appartient à *une et une seule* des composantes connexes de G.

Nous pourrons démontrer cette propriété avec un petit peu d'arsenal mathématique.

7.3 Équivalences

Une relation d'équivalence regroupe les objets d'un ensemble en paquets en fonction de caractéristiques communes. Une telle classification est associée à une relation binaire homogène, qui associe deux à deux les objets appartenant à un même paquet. On isole trois propriétés d'une telle relation reflétant l'appartenance de plusieurs objets à une même classe :

— tout élément appartient à son propre paquet (réflexivité),

— l'énoncé « deux éléments appartiennent au même paquet » ne dépend pas de l'ordre dans lequel on considère les éléments (symétrie),

— l'appartenance à un même paquet se propage de proche en proche (transitivité).

Ces trois point réunis caractérisent la notion d'équivalence.

Relation d'équivalence. Rappel : une relation binaire homogène $\mathcal{R} \subseteq E \times E$ est un ensemble de paires d'éléments « en relation » l'un avec l'autre. On a vu les cas particuliers suivants :

— la relation $\mathcal{R}$est ***réflexive*** lorsque tout élément est en relation avec lui-même

$$\forall e \in E, e\mathcal{R}e$$

— la relation $\mathcal{R}$ est ***symétrique*** lorsqu'elle ne distingue pas l'ordre de ses deux arguments

$$\forall e_1, e_2 \in E,\ e_1 \mathcal{R} e_2 \Rightarrow e_2 \mathcal{R} e_1$$

— la relation $\mathcal{R}$ est ***transitive*** lorsqu'elle se propage de proche en proche

$$\forall e_1, e_2, e_3 \in E,\ (e_1 \mathcal{R} e_2 \wedge e_2 \mathcal{R} e_3) \Rightarrow e_1 \mathcal{R} e_3$$

Une ***relation d'équivalence*** est une relation binaire homogène qui est à la fois réflexive, symétrique et transitive. Souvent, on utilisera le symbole $\approx$ plutôt que $\mathcal{R}$ pour une telle relation.
Exemples de relations d'équivalence :

— la relation « être égal à »,

— la relation « avoir la même taille que » sur des listes,

— la relation « être lié par un chemin à » dans un graphe non orienté.

Partant d'une relation d'équivalence, on peut reconstruire les « paquets » sous-jacents, sous la forme de *classes d'équivalence.*

Classes d'équivalence. On considère un ensemble E et une relation d'équivalence $\approx$ sur E. La ***classe d'équivalence*** d'un élément $e \in E$ pour $\approx$, notée $[e]$, est l'ensemble des éléments de E qui sont en relation avec e.

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

Les classes d'équivalence de $\approx$ sont toutes les classes $[e]$ des élément $e \in E$. Tout élément e d'une classe d'équivalence C est appelé un ***représentant*** de cette classe. Nous allons montrer une propriété essentielle des classes : chaque élément $e \in E$ appartient à une et une seule classe d'équivalence de $\approx$.
Propriétés.

1. Pour tout $e \in E$ on a $e \in [e]$.
   Preuve : par réflexivité on a $e \approx e$, et donc par définition $e \in [e]$.
2. Deux éléments équivalents définissent la même classe.

   $$\forall e_1, e_2 \in E,\ e_1 \approx e_2 \Rightarrow [e_1] = [e_2]$$

   Preuve : soient e_1 et e_2 deux éléments de E tels que $e_1 \approx e_2$. Montrons que $[e_1] \subseteq [e_2]$.
   Soit $x \in [e_1]$. Par définition $e_1 \approx x$, d'où par symétrie $x \approx e_1$ et par transitivité $x \approx e_2$.
   Ainsi $e_2 \approx x$ par symétrie à nouveau, d'où par définition $x \in [e_2]$.
   On montrerait de même que $[e_2] \subseteq [e_1]$, donc $[e_1] = [e_2]$.
3. Deux classes d'équivalence sont soit disjointes, soit égales.

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

   Preuve : 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 $e_1 \approx e$ et $e_2 \approx e$. Par symétrie on a donc $e \approx e_2$ et par transitivité $e_1 \approx e_2$. Alors la propriété 2 permet de conclure $[e_1] = [e_2]$.*

De ces propriétés, on déduit que l'ensemble $\mathcal{C} = \{C_1, C_2, ...\}$ des classes d'équivalence d'une relation d'équivalence $\approx$ sur E couvre tout E sans chevauchements.

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

Ainsi, les classes d'équivalences de $\approx$ forment bien une ***partition*** de E : chaque élément $e \in E$ appartient à une et une seule classe d'équivalence de $\approx$.

Application aux composantes connexes d'un graphe. Les composantes connexes d'un graphe peuvent être définies comme les classes d'équivalence de la relation d'*accessibilité*. Étant donnés un graphe G et deux sommets s et s' de G, on dit que s' est accessible à partir de s, et on note $s \to^* s'$, s'il existe un chemin dans G entre s et s'.

Une ressemblance avec une notation vue en PIL ne serait pas tout à fait fortuite.

On va d'abord vérifier que cette relation est une équivalence, puis analyser ses classes d'équivalence.

Dans un graphe non orienté, la relation $\to^*$ est une relation d'équivalence.

Preuve :

- *Réflexivité.* Pour tout s on a bien un chemin de s à s (le chemin vide), et donc $s \to^* s$.
- *Symétrie.* Soient s et s' tels qu'il existe un chemin de s à s' dans notre graphe G : $s = s_0 \to s_1 \to \ldots \to s_{n-1} \to s_n = s'$. Le graphe n'étant pas orienté, chaque arête peut être prise dans l'autre sens. On forme ainsi un chemin $s_n \to s_{n-1} \to \ldots \to s_1 \to s_0$ de s' vers s. Bilan : si $s \to^* s'$ alors $s' \to^* s$.
- *Transitivité.* Soient s_1, s_2 et s_3 tels qu'il existe un chemin de s_1 vers s_2 et un chemin de s_2 vers s_3. La concaténation de ces deux chemins forme un chemin allant de s_1 à s_3. Donc : si $s_1 \to^* s_2$ et $s_2 \to^* s_3$ alors $s_1 \to^* s_3$.

La relation $\to^*$ d'accessibilité étant une équivalence, elle définit des classes d'équivalence $[s]$ sur les sommets d'un graphe non orienté, que l'on peut maintenant analyser.

Dans un graphe non orienté G, les classes d'équivalence de la relation d'accessibilité sont précisément les composantes connexes de G.

Preuve : *toute classe $[s]$ est une composante connexe de G.*

- *Tous les sommets d'une classe $[s]$ sont connectés deux à deux par des chemins.*
  Soit $[s]$ une classe de $\to^*$, et $s_1, s_2 \in [s]$ deux sommets de cette classe. Par définition de $[s]$ on a $s \to^* s_1$ et $s \to^* s_2$. Par symétrie on a $s_1 \to^* s$, et par transitivité on déduit $s_1 \to^* s_2$: on a un chemin de s_1 vers s_2.
- *Aucun sommet d'une classe $[s]$ n'est connecté à un élément n'appartenant pas à $[s]$.*
  Soit $[s]$ une classe de $\to^*$ et $s_1 \in [s]$ un sommet de cette classe. Soit s_2 un sommet quelconque de G tel que $s_1 \to^* s_2$. Alors par définition de $[s]$ on a $s \to^* s_1$, et par transitivité on déduit $s \to^* s_2$. Donc $s_2 \in [s]$. Bilan : un sommet s_2 accessible de puis s_1 est nécessairement dans $[s]$, et ainsi aucun sommet hors de $[s]$ ne peut être connecté à un sommet de $[s]$.

Preuve : *toute composante connexe C de G est la classe $[s]$ d'un certain sommet s.*
Soit C une composante connexe de G. Par définition, C n'est pas l'ensemble vide : il existe au moins un sommet $s \in C$.

- *La composante C est incluse dans la classe $[s]$.*
  Soit $s' \in C$ un sommet de la composante connexe C. Comme s et s' sont tous deux dans C, il existe un chemin $s \to^* s'$, et donc $s' \in [s]$ par définition de $[s]$.
- *La classe $[s]$ est incluse dans la composante C.*
  Soit $s' \in [s]$. Par définition on a $s \to^* s'$. Comme s est dans C, le sommet s' ne peut pas être en dehors de C.

On a donc bien $C = [s]$.

Application au problème du labyrinthe. Considérons un graphe G non orienté *sans cycles*, et deux sommets s et s' de G. Alors :

Ajouter l'arête $s \to s'$ à G crée un cycle
si et seulement si
les sommets s et s' sont dans la même composante connexe.

Traduisons cela pour notre processus de génération de labyrinthe. On énumère toutes les cloisons dans un ordre aléatoire, et pour chacune :

- si elle sépare deux salles qui sont dans la même composante connexe (dans la même classe d'accessibilité), ne rien faire,
- si elle sépare deux salles qui sont dans des composantes connexes disjointes (dans des classes d'accessibilité différentes), créer une porte.

Pour compléter la construction, il ne nous manque plus qu'une structure ou un algorithme permettant de manipuler efficacement des classes d'équivalence de sommets.

7.4 Structure de données : Union-Find

La structure *Union-Find* (également appelée *disjoint sets*) permet de manipuler des parties disjointes d'un ensemble E, et en particulier des classes d'équivalence d'éléments de E. La structure fournit deux opérations principales :

— find(e) identifie la classe $[e]$ d'un élément $e \in E$,

— union(e_1, e_2) modifie la structure pour y fusionner les classes de e_1 et e_2.

L'opération find permet de déterminer si deux éléments e_1 et e_2 appartiennent à la même classe : il suffit de tester si find(e_1) = find(e_2). Dans le problème du labyrinthe, c'est ce test qui déterminera si deux salles sont déjà dans la même composante connexe du labyrinthe. L'opération union peut être utilisée pour bâtir une telle structure à partir d'un ensemble de paires d'éléments équivalents :

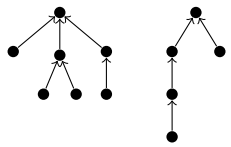
1. on part d'une structure initiale dans laquelle on considère que chaque élément e a une classe réduite à lui-même,
2. pour chaque paire (e_1, e_2) d'éléments équivalents, on fusionne les classes $[e_1]$ et $[e_2]$ à l'aide de l'opération union.

Modélisation avec des graphes. Notre structure d'*union-find* sera un graphe orienté ayant pour sommets les éléments de l'ensemble E, avec deux particularités de forme :

— chaque sommet a au plus une arête sortante,

— le graphe est acyclique.

Un tel graphe est composé de plusieurs blocs ayant des formes comme les suivantes, où tous les chemins convergent vers un élément racine.



Chaque bloc correspond à une classe, et l'élément « racine » d'un bloc peut servir à identifier le bloc (et donc une classe). On donne alors le comportement suivant à nos deux opérations :

— L'opération find(e) renvoie l'élément racine du bloc contenant e. Pour cela, il suffit de suivre les arêtes sortantes à partir de e jusqu'à arriver au bout du chemin.

— L'opération union(e_1, e_2) regroupe les deux blocs contenant e_1 et e_2. Pour cela, il suffit d'ajouter une arête entre les racines de ces deux blocs (à supposer que e_1 et e_2 ne soient pas déjà dans le même bloc).

Réalisation par un tableau. On n'a besoin que de deux informations par sommet :

1. le sommet est-il une racine ?
2. si le sommet n'est pas une racine, quel est le numéro de son unique successeur ?

On peut résumer ces deux informations dans un unique tableau `t` d'entiers, dans lequel

$$\begin{cases} t[i] = i & \text{si } s_i \text{ est une racine} \\ t[i] = j & \text{avec } i \neq j \text{ si } s_j \text{ est l'unique successeur de } s_i \end{cases}$$

Initialisation en java : on crée un tableau `t` dans lequel, pour tout i, $t[i] = i$.

```
class Uf {
  private int[] t;
  Uf(int n) {
    this.t = new int[n];
    for (int i=0; i<n; i++) { t[i] = i; }
  }
```

La méthode `find` doit trouver la racine du bloc de `e`. Pour cela elle s'appelle récursivement sur le successeur de `e`, jusqu'à arriver à un sommet qui est son propre successeur.

```
  int find(int e) {
    int s = t[e];
    if (s == e) { return e; }
    else        { return find(s); }
  }
```

La méthode `union` connecte les blocs de `e1` et de `e2` en désignant l'une des deux racines comme nouveau successeur de l'autre.

```
  void union(int e1, int e2) {
    int r1 = find(e1);
    int r2 = find(e2);
    if (r1 != r2) { t[r1] = r2; }
  }
}
```

Approfondissement : améliorations. Le coût d'utilisation de la structure *union-find* est essentiellement le coût de l'opération find, qui doit parcourir les successeurs d'un sommet jusqu'à trouver la racine du bloc. Le temps est donc proportionnel à la longueur du chemin à parcourir, qui peut lui-même être linéaire en le nombre de sommets. Pour maintenir des longueurs de chemins très courtes, on peut ajouter deux choses à ces algorithmes.

— *Union par rang :* au moment de relier les deux racines `r1` et `r2`, on essaie de mettre l'arête dans le sens qui générera les chemins les moins longs.

Pour réaliser cela, on associe à chaque sommet une information supplémentaire appelée « rang », qui majore la longueur des chemins de son bloc. 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. Note : l'information de rang n'est utile que pour la racine de chaque bloc, on ne cherchera donc pas à la mettre à jour pour les autres.

```
class Uf {
  private int[] t;
  private int[] rang;
  Uf(int n) {
    this.t = new int[n]; for (int i=0; i<n; i++) { t[i] = i; }
    this.rang = new int[n]; for (int i=0; i<n; i++) { rang[i] = 0; }
  }
  void union(int e1, int e2) {
    int r1 = find(e1);
    int r2 = find(e2);
    if (r1 == r2) return;
    if (rang[r1] < rang[r2]) {
      t[r1] = r2;
    } else {
      t[r2] = r1;
      if (rang[r1] == rang[r2]) rang[r1]++;
    }
  }
}
```

— *Compression de chemins :* à chaque utilisation de `find`, on mémorise la racine trouvée pour ne plus jamais avoir besoin de parcourir à nouveau le même chemin.

On réalise cela en indiquant la racine trouvée comme nouveau successeur direct du sommet, et on le fait même pour chacun des sommets rencontrés sur la route. 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.

```
  int find(int e) {
    int s = t[e];
    if (s == e) {
      return e;
    } else {
      int r = find(s);
      t[e] = r;
      return r;
    }
  }
```

Avec ces deux améliorations, la complexité diminue radicalement : chaque opération `find` a maintenant un temps quasiment constant.

7.5 Code final : génération du labyrinthe

Une classe simple pour un graphe non orienté. On conserve le principe précédent selon lequel les sommets sont numérotés.

```
class Graph {
  private final int size;
  private ArrayList<ArrayList<Integer>> adj;
  public Graph(int size) {
    this.size = size;
    this.adj = new ArrayList<>(size);
    for (int s=0; s<size; s++) { adj.add(new ArrayList<>()); }
  }
  public void addEdge(int s, int t) { adj.get(s).add(t); adj.get(t).add(s); }
  public int size() { return size; }
  public Iterable<Integer> succ(int s) { return adj.get(s); }
  public boolean hasEdge(int s, int t) {
      for (int v: succ(s)) { if (v == t) return true; }
      return false;
  }
}
```

Structure pour représenter les cloisons où l'on est susceptible de créer une porte. On donne : le numéro d'une case, et un booléen pour identifier la direction (si `true` : cloison verticale à l'est de la case, si `false` : cloison horizontale au sud).

```
static class Wall {
  int s;
  boolean v;
  Wall(int s, boolean v) { this.s = s; this.v = v; }
}
```

Si notre graphe correspond à une grille carrée de côté n, il aura n^2 sommets, et le sommet à la ligne i et colonne j aura le numéro $i * n + j$. Un voisin à l'est du sommet numéro k a donc le numéro $s + 1$, et un voisin au sud le numéro $s + n$.

Construction du labyrinthe : on initialise un graphe, dans lequel on ajoute des arêtes à mesure que l'on ouvre des portes dans les cloisons. En parallèle, on maintient une structure *union-find* pour savoir quelles salles sont déjà connectées par un chemin. Avant cette étape de construction, on génère l'ensemble des cloisons dans un tableau, et on mélange ce tableau. L'algorithme de mélange utilisé, bien que très simple, est connu pour générer des *mélanges parfaits* (toutes les permutations du tableau sont équiprobables).

```
static Graph mkLaby(int n) {
  Graph g = new Graph(n*n);
  Uf uf = new Uf(n*n);
  ArrayList<Wall> walls = new ArrayList<>(n*n);
  // Generate walls
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      if (i<n-1) { walls.add(new Wall(i*n+j, true)); }
      if (j<n-1) { walls.add(new Wall(i*n+j, false)); }
    }
  }
  // Randomize walls (Fisher-Yates algorithme, a.k.a. Knuth shuffle)
  Random rnd = new Random();
  int k = walls.size();
  for (int i=1; i<k; i++) { Collections.swap(walls, i, rnd.nextInt(i+1)); }
  // Select doors
  for (Wall w : walls) {
    int s = w.s;
    int t = w.s+(w.v?n:1);
    if (uf.find(s) != uf.find(t)) {
      uf.union(s, t);
      g.addEdge(s, t);
    }
  }
  return g;
}
```

Pour finir, une petite fonction pour affichier un labyrinthe en ASCII.

```
static void printLaby(int n, Graph g) {
  for (int j=0; j<n; j++) { System.out.print("####"); }
  System.out.println("##");
  for (int i=0; i<n; i++) {
    System.out.print("##");
    for (int j=0; j<n; j++) {
      System.out.print("␣␣");
      if (j<n-1 && g.hasEdge(i*n+j, i*n+j+1)) { System.out.print("␣␣"); }
      else { System.out.print("##"); }
    }
    System.out.println();
    System.out.print("##");
    for (int j=0; j<n; j++) {
      if (i<n-1 && g.hasEdge(i*n+j, (i+1)*n+j)) { System.out.print("␣␣"); }
      else { System.out.print("##"); }
      System.out.print("##");
    }
    System.out.println();
  }
}
```

Et le résultat !

```
##############################################################################
##  ##      ##  ##          ##      ##      ##  ##  ##  ##          ##  ##
##  ######  ##  ##  ##############  ######  ##  ##  ##  ##  ##########  ##
##  ##          ##              ##  ##          ##  ##          ##      ##
##  ##  ##  ##############  ######  ##  ######  ##  ##  ##############  ##
##      ##  ##  ##              ##  ##  ##  ##      ##                  ##
######  ######  ##########  ######  ##  ##  ##  ##  ######  ##########  ##
##  ##              ##  ##  ##      ##  ##      ##      ##  ##  ##      ##
##  ######  ##  ######  ##  ##  ##  ######  ##  ##########  ##  ######  ##
##          ##          ##  ##  ##          ##          ##  ##          ##
##########  ##  ##  ##  ##  ##  ##  ##########  ######################  ##
##          ##  ##  ##  ##  ##  ##      ##                  ##          ##
##################  ##  ##  ######  ##############  ##############  ######
##  ##              ##              ##      ##  ##                      ##
##  ##  ##############  ##################  ##  ######  ######  ##########
##  ##  ##          ##  ##      ##      ##  ##          ##              ##
##  ##  ##  ######  ##  ##  ######  ######  ######  ##########  ##########
##  ##  ##      ##  ##          ##          ##      ##                  ##
##  ######  ##  ######  ##  ##  ##  ##  ##############  ##################
##          ##  ##  ##  ##  ##  ##  ##              ##              ##  ##
##  ######  ######  ##########  ##  ##  ##  ##  ##  ######  ##########  ##
##  ##  ##  ##                      ##  ##  ##  ##  ##  ##  ##  ##      ##
##  ##  ######  ##############  ##########  ##  ######  ######  ##  ##  ##
##      ##      ##  ##  ##          ##  ##  ##          ##          ##  ##
######  ##  ##  ##  ##  ##############  ##  ######  ##############  ######
##      ##  ##  ##  ##  ##  ##              ##          ##      ##  ##  ##
######  ######  ##  ##  ##  ##########  ######  ##  ######  ##  ##  ##  ##
##      ##  ##          ##      ##      ##  ##  ##  ##          ##      ##
######  ##  ######  ##  ##  ##  ##  ######  ######  ##  ##############  ##
##                  ##      ##  ##          ##  ##          ##          ##
######  ##  ##########  ######  ##  ##  ######  ##  ##################  ##
##      ##  ##              ##  ##  ##  ##                  ##  ##  ##  ##
##  ######  ##  ##  ######################  ##  ######  ######  ##  ##  ##
##      ##  ##  ##  ##          ##  ##      ##      ##      ##  ##      ##
######  ##############  ##########  ##  ##  ##  ##############  ##  ######
##                  ##                  ##  ##  ##                      ##
##############################################################################
```