Exercice 1

On considère le schéma de base de données suivante, qui décrit un ensemble de fabricants de matériel informatique, les matériels qu'ils vendent, leurs clients et ce qu'achètent leurs clients. Les attributs des clés primaires des six premières relations sont soulignés.

```
Production (NomFabricant, Modele)
Ordinateur (Modele, Frequence, Ram, Dd, Prix)
Portable (Modele, Frequence, Ram, Dd, Ecran, Prix)
Imprimante (Modele, Couleur, Type, Prix)
Fabricant (Nom, Adresse, NomPatron)
Client (Num, Nom, Prenom)
Achat (NumClient, NomFabricant, Modele, Quantite)
```

- Chaque client possède un numéro unique connu de tous les fabricants.
- La relation `Production` donne, pour chaque fabricant, l'ensemble des modèles fabriqués par ce fabricant.
- Deux fabricants différents peuvent proposer le même matériel.
- La relation `Ordinateur` donne, pour chaque modèle d'ordinateur, la vitesse du processeur (en Hz), les tailles de la RAM et du disque dur (en Go) et le prix de l'ordinateur (en €).
- La relation `Portable`, en plus des attributs précédents, comporte la taille de l'écran (en pouces).
- La relation `Imprimante` indique, pour chaque modèle d'imprimante, si elle imprime en couleur (Vrai/-Faux), le type d'impression (laser ou jet d'encre) et le prix (en €).
- La relation `Fabricant` stocke les nom et adresse de chaque fabricant, ainsi que le nom de son patron.
- La relation `Client` stocke les noms et prénoms de tous les clients de tous les fabricants.
- Enfin, la relation `Achat` regroupe les quadruplets (client c, fabricant f, modèle m, quantité q) tels que le client de numéro c a acheté q fois le modèle m au fabricant de nom f.

1. Proposer une clé primaire pour la relation `Achat` et indiquer ses conséquences en termes de modélisation.
2. Identifier l'ensemble des clés étrangères éventuelles de chaque table.
3. Donner en SQL des requêtes répondant aux questions suivantes :
   (a) Quels sont les numéros de modèles des matériels fabriqués par l'entreprise Durand ?
   (b) Quels sont les noms et adresses des fabricants produisant des portables dont le disque dur a une capacité d'au moins 500 Go ?
   (c) Quels sont les noms des fabricants qui produisent au moins 10 modèles différents d'imprimantes ?
   (d) Quels sont les numéros des clients n'ayant acheté aucune imprimante ?

Corrigé

1. Sans modification de la table, le triplet `(NumClient, NomFabricant, Modele)` peut faire office de clé primaire. Cela implique lors d'un nouvel achat de modifier l'enregistrement correspondant pour augmenter la valeur du champ `Quantite`. On aurait pu imaginer une autre modélisation de la table en deux tables distinctes :
   - une table `Achat (Num, NumClient, NomFabricant, Date)` indiquant quel client a fait un achat chez quel fabricant à quelle date ;
   - une table `DetailsAchats (NumAchat, Modele, Quantite)` indiquant quel modèle a été acheté en quelle quantité en fonction du numéro d'achat (on peut faire plusieurs achats d'un coup). Dans cette deuxième table, le couple `(NumAchat, Modele)` aurait fait office de clé primaire.
2. Dans `Production`, chacun des champs est une clé étrangère : le premier vers la table `Fabricant`, le deuxième vers les trois tables `Ordinateur`, `Portable` et `Imprimante`.
3. Dans `Achat`, les trois premiers champs sont des clés étrangères : le premier vers `Client`, le deuxième vers `Fabricant` et le troisième vers les trois tables `Ordinateur`, `Portable` et `Imprimante`.
4. On propose les requêtes suivantes :

(a)
```sql
SELECT Modele FROM Production
WHERE NomFabricant = 'Durand';
```

(b)
```sql
SELECT DISTINCT Nom, Adresse FROM Fabricant
JOIN Production ON Nom = NomFabricant
JOIN Portable ON Portable.Modele = Production.Modele
WHERE Dd >= 500
```

(c)
```sql
SELECT NomFabricant FROM Production
JOIN Imprimante ON Imprimante.Modele = Production.Modele
GROUP BY NomFabricant
HAVING COUNT(*) >= 10
```

(d)
```sql
SELECT Num FROM Client
EXCEPT
SELECT DISTINCT NumClient FROM Achat
JOIN Imprimante ON Imprimante.Modele = Achat.Modele)
```

Exercice 2: L'exercice suivant est à traiter dans le langage C.

Dans cet exercice, on considère des arbres binaires non étiquetés. On dit qu'un arbre est **binaire strict** si tout nœud interne a exactement deux enfants, ou autrement dit si tout nœud (interne ou non) a zéro ou deux enfants. On dit qu'un arbre est **parfait** s'il est binaire strict et que toutes ses feuilles sont à même profondeur. On dit qu'il est **presque-parfait** si tous les niveaux de profondeur sont remplis par des nœuds, sauf éventuellement le dernier, rempli par la gauche.

1. Combien existe-t-il d'arbres presque-parfaits de taille 6 ? Les représenter graphiquement.
2. Citer une structure de données qui peut s'implémenter avec des arbres presque-parfaits.
3. Proposer une définition par induction d'un arbre parfait de hauteur h, pour $h \in \mathbb{N}$. Justifier rigoureusement que cette définition coïncide avec celle de l'énoncé.

On implémente un arbre binaire en C par le type `arbre` défini de la manière suivante :

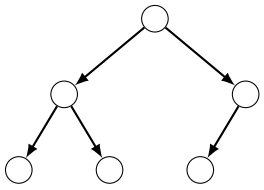
```c
struct Noeud {
    struct Noeud* gauche;
    struct Noeud* droite;
};

typedef struct Noeud arbre;
```

Ainsi, un arbre contient un pointeur vers son fils gauche et son fils droit.

4. Écrire une fonction `int hauteur(arbre* a)` qui prend en argument un pointeur vers un arbre et calcule et renvoie sa hauteur.
5. En déduire une fonction `bool est_parfait(arbre* a)` qui prend en argument un pointeur vers un arbre et renvoie un booléen qui vaut `true` si et seulement si l'arbre pointé par `a` est parfait. Quelle est sa complexité temporelle pour un arbre `a` qui est effectivement parfait ?
6. Écrire une fonction `arbre* plus_grand_presque_parfait(arbre* a)` qui prend en argument un pointeur vers un arbre a et renvoie un pointeur vers le plus grand sous-arbre de a qui est presque-parfait. La complexité de cette fonction doit être linéaire en la taille de l'arbre a.
   Indication : que peut-on dire des enfants gauche et droit d'un arbre presque-parfait de hauteur h ?

Corrigé

1. Il n'existe qu'un seul arbre parfait de taille 6. On peut le représenter par :



2. La structure de tas binaire peut s'implémenter avec des arbres presque-parfaits. Elle permet de réaliser une file de priorité efficace.
3. On propose la définition suivante :
   - l'arbre constitué d'un seul nœud est un arbre parfait de hauteur 0 ;
   - pour $h > 0$, l'arbre constitué d'un nœud et ayant pour enfants gauche et droit l'arbre parfait de hauteur $h-1$ est un arbre parfait de hauteur h.

   Montrons qu'elle coïncide avec celle de l'énoncé. Appelons temporairement « parfait (bis) » un arbre défini de cette manière.
   - Montrons par récurrence sur h qu'un arbre parfait A de hauteur h est un arbre parfait (bis) :
     * si $h = 0$, alors A est constitué d'un seul nœud dont est parfait (bis) ;
     * supposons la propriété établie jusqu'à $h \in \mathbb{N}$ fixé. Soit A un arbre parfait de hauteur $h+1$. Comme A est binaire strict, il possède deux enfants. De plus, chacun de ses enfants est de hauteur h (car toutes les feuilles sont à même profondeur), est binaire strict et possède chacun toutes ses feuilles à même profondeur. Les enfants sont donc des arbres parfaits de hauteur h, donc des arbres parfaits (bis) de hauteur h par hypothèse de récurrence. On en conclut que A est bien un arbre parfait (bis) de hauteur $h+1$.

     On conclut par récurrence.
   - Montrons par induction qu'un arbre parfait (bis) est un arbre parfait :
     * comme précédemment, le cas de base est assuré ;
     * supposons que A est un arbre parfait de hauteur h. Alors l'arbre $B = N(A, A)$ reste binaire strict et possède toutes ses feuilles à même profondeur. Il est donc parfait.
4. On propose la fonction suivante (on rappelle qu'il n'existe pas de fonction `max` en C) :

```c
int hauteur(arbre* a){
    if (a == NULL){
        return -1;
    } else {
        int hg = hauteur(a->gauche);
        int hd = hauteur(a->droite);
        return 1 + ((hg<hd)?hd:hg);
    }
}
```

5. On utilise la définition inductive. Les cas de base sont l'arbre vide et la feuille.

```c
bool est_parfait(arbre* a){
    if (a == NULL || (a->gauche == NULL && a->droite == NULL)) {
        return true;
    } else {
        int hg = hauteur(a->gauche);
        int hd = hauteur(a->droite);
        return hg == hd && est_parfait(a->gauche) && est_parfait(a->droite);
    }
}
```

   On remarque que la fonction `hauteur` a une complexité qui est linéaire en la taille de l'arbre. On obtient donc pour `est_parfait` une complexité vérifiant :

$$C(a) = \begin{cases} \mathcal{O}(1) & \text{si } |a| \leqslant 1 \\ C(g) + C(d) + \mathcal{O}(|a|) & \text{si } a = N(g,d) \text{ et } |a| > 1 \end{cases}$$

Pour un arbre parfait a de taille n, on a donc comme formule de récurrence : $C(n) = 2C\left(\frac{n-1}{2}\right) + \mathcal{O}(n)$. On reconnait une formule qui se résout en $C(n) = \mathcal{O}(n \log n)$.

6. Pour réussir à écrire une fonction efficace, il faut remarquer qu'un arbre presque-parfait de hauteur $h > 0$ a pour enfants gauche et droit :
   - soit un arbre parfait de hauteur $h-1$ et un arbre presque-parfait de hauteur $h-1$;
   - soit un arbre presque-parfait de hauteur $h-1$ et un arbre parfait de hauteur $h-2$.

   en remarquant qu'un arbre parfait est un arbre presque-parfait particulier. Ainsi, on peut écrire la fonction en utilisant une fonction intermédiaire qui calcule pour chaque nœud un quadruplet contenant :
   - la taille et la hauteur du sous-arbre ;
   - un booléen qui détermine si le sous-arbre est parfait ;
   - un booléen qui détermine si le sous-arbre est presque-parfait.

   Si on connait ces informations pour les deux enfants d'un nœud, alors on peut les calculer pour ce nœud en temps constant. Cela donnera bien une complexité linéaire en la taille de l'arbre.
   On implémente dans un premier temps une structure de quadruplet :

```
struct Quad{
    int t;
    int h;
    bool parf;
    bool pparf;
};

typedef struct Quad quad;
```

On commence alors par écrire une fonction auxiliaire qui fait ces calculs, tout en mettant à jour un pointeur de pointeur d'arbre et un pointeur d'entier, pointant vers le plus grand sous-arbre presque-parfait et sa taille respectivement.

```
quad pgpp_aux(arbre* a, arbre** best, int* tbest){
    if (a == NULL){
        quad q = {.t = 0, .h = -1, .parf = true, .pparf = true};
        return q;
    } else {
        quad qg = pgpp_aux(a->gauche, best, tbest);
        quad qd = pgpp_aux(a->droite, best, tbest);
        quad q = {.t = 1 + qg.t + qd.t,
                  .h = 1 + ((qg.h<qd.h)?qd.h:qg.h),
                  .parf = qg.h == qd.h && qg.parf && qd.parf,
                  .pparf = (qg.h == qd.h && qg.parf && qd.pparf) ||
                           (qg.h == qd.h + 1 && qg.pparf && qd.parf)};
        if (q.pparf && q.t > *tbest){
            *best = a;
            *tbest = q.t;
        }
        return q;
    }
}
```

enfin, on peut écrire la fonction finale :

```
arbre* plus_grand_presque_parfait(arbre* a){
    arbre* best = NULL;
    int tbest = 0;
    pgpp_aux(a, &best, &tbest);
    return best;
}
```

Exercice 3

On considère le programme suivant, ici en OCaml, dans lequel n fils d'exécution incrémentent tous un même compteur partagé.

```
(* Nombre de fils d'exécution *)
let n = 100

(* Un même compteur partagé *)
let compteur = ref 0

(* Chaque fil d'exécution de numéro i va incrémenter le compteur *)
let f i = compteur := !compteur + 1

(* Création de n fils exécutant f associant à chaque fil son numéro *)
let threads = Array.init n (fun i -> Thread.create f i)

(* Attente de la fin de n fils d'exécution *)
let () = Array.iter (fun t -> Thread.join t) threads
```

On rappelle que l'on dispose en OCaml des trois fonctions `Mutex.create : unit -> Mutex.t` pour la création d'un verrou, `Mutex.lock : Mutex.t -> unit` pour le verrouillage et `Mutex.unlock : Mutex.t -> unit` pour le déverrouillage, du module `Mutex` pour manipuler des verrous.

1. Quelles sont les valeurs possibles que peut prendre le compteur à la fin du programme.
2. Identifier la section critique et indiquer comment et à quel endroit ajouter des verrous pour garantir que la valeur du compteur à la fin du programme soit n de manière certaine.

Dans la suite de l'exercice, on suppose que l'on ne dispose pas d'une implémentation des verrous. On se limite au cas de deux fils d'exécution, numérotés 0 et 1. Nous cherchons à garantir deux propriétés :

- *Exclusion mutuelle* : un seul fil d'exécution à la fois peut se trouver dans la section critique ;
- *Absence de famine* : tout fil d'excéution qui cherche à entrer dans la section critique pourra le faire à un moment.

On utilise pour cela un tableau `veut_entrer` qui indique pour chaque fil d'exécution s'il souhaite entrer en section critique ainsi qu'une variable `tour` qui indique quel fil d'exécution peut effectivement entrer dans la section critique. On propose ci-dessous deux versions modifiées `f_a` et `f_b` de la fonction `f`, l'objectif étant de pouvoir exécuter `f_a 0` et `f_a 1` de manière concurrente, et de même pour `f_b`.

```
let veut_entrer = [|false; false|]
let tour = ref 0

let f_a i =
        let autre  = 1 - i in
        veut_enter.(i) <- true;
        while veut_entrer.(autre) do () done;
        (* section critique *)
        veut_entrer.(i) <- false

let f_b i =
        let autre = 1 - i in
        veut_entrer.(i) <- true;
        tour := i;
        while veut_entrer.(autre) && !tour = autre do () done;
        (* section critique *)
        veut_entrer.(i) <- false
```

3. Expliquer pourquoi aucune de ces deux versions ne convient, en indiquant la propriété qui est violée.
4. Proposer une version `f_c` qui permet de garantir les deux propriétés.

5. Connaissez-vous un algorithme permettant de généraliser à n fils d'exécution ? Rappeler très succinctement son principe.

Corrigé

1. A priori, toute valeur entre 1 et 100 (inclus de deux côtés) est possible.
2. Pour obtenir 100 de manière certaine, on protège la section critique comme suit :

```
let m = Mutex.create ()

let f i =
   Mutex.lock m;
   compteur := !compteur + 1
   Mutex.unlock m;
```

3. Avec `f_a` on peut avoir famine avec la trace d'exécution suivante :
   - Le fil 0 calcul `autre` puis indique que `veut_entrer.(0)` vaut `true`.
   - Le fil 1 calcul `autre` puis indique que `veut_entrer.(1)` vaut `true`.
   - Les deux fils sont alors bloqués dans la boucle `while`.

   Avec `f_b` il n'y a pas exclusion mutuelle ce qui s'observe via la trace suivante :
   - Le fil 0 calcule `autre`, indique qu'il veut entrer, fixe `tour` à 0 puis entre en section critique (ce qui est possible puisque ce n'est pas le tour du fil 1).
   - Le fil 1 fait exactement la même chose.
   - Les deux fils sont alors en même temps en section critique.
4. Il s'agit de reconstruire l'algorithme de Peterson :

```
let f_c i =
   let autre = 1-i in
   veut_entrer.(i) <- true;
   tour := autre;
   while veut_entrer.(autre) && !tour = autre do () done;
   (* section critique *)
   veut_entrer.(i) <- false;
```

5. L'algorithme de la boulangerie de Lamport permet de généraliser à n fils. On y utilise un tableau `veut_entrer` à n cases et un tableau d'entiers `tickets` à n cases. Lorsqu'un fil i veut entrer en section critique, il modifie sa case `veut_entrer.(i)` puis calcule et stocke son ticket comme étant (le maximum du tableau `tickets`) +1. Il attend ensuite d'être le fil souhaitant aller en section critique disposant du couple (`tickets.(i)`, i) le plus petit pour l'ordre lexicographique.

Exercice 4: Graphes, rayons et applications

Définition

Un graphe non orienté est un couple $G = (S, A)$ où S est un ensemble sommets, potentiellement infini, et A est un ensemble d'arêtes, $A \subseteq \mathcal{P}_2(S)$. La **taille** de G est le cardinal $|S|$. On dit que G est fini si S est fini. On dit que G est dénombrable si S est infini dénombrable.

Pour $n \geqslant 0$, un **chemin** est une suite finie $(s_0, \ldots, s_n)$ de sommets telle que pour $0 \leqslant i < n$, $\{s_i, s_{i+1}\} \in A$. Le chemin est dit **simple** si chaque sommet y apparaît au plus une fois.

Un **rayon** de G est une suite infinie de sommets $(s_i)_{i\in\mathbb{N}}$ telle que pour tout entier $n \in \mathbb{N}$, $(s_0, \ldots, s_n)$ est un chemin simple.

G est dit **connexe** si pour tous sommets $s, t \in S^2$, il existe un chemin de s à t.

1. Montrer que si G admet un rayon, alors il est infini.

Définition

Pour $G = (S, A)$ un graphe et $s \in S$, le **degré** de s est son nombre de voisins, c'est-à-dire le cardinal de l'ensemble $\{t \in S \mid \{s, t\} \in A\}$.

G est dit **localement fini** si tout sommet de G est de degré fini. Il est dit **localement borné** si l'ensemble des degrés est borné.

2. Donner un exemple de graphe infini, connexe et localement borné. Donner un exemple de graphe infini, connexe, localement fini mais pas localement borné.

Dans la suite, on dit qu'un graphe admet la propriété (K) s'il est connexe, infini et localement fini. On cherche à montrer le résultat suivant :

Tout graphe dénombrable qui vérifie (K) admet un rayon.

3. Est-ce que chacune des trois propriétés de (K) est nécessaire pour que ce théorème soit vrai ?
4. Montrer qu'il existe un graphe G infini et localement fini, tel que pour tout $k \in \mathbb{N}$, G possède un chemin simple de taille k mais tel que G n'admette pas de rayon.
5. Montrer que si $G = (S, A)$ satisfait (K) et $s \in S$, alors $G[S \setminus \{s\}]$ est une union disjointe finie de graphes connexes, dont au moins l'un d'entre eux satisfait (K).
6. Montrer que si G satisfait (K), alors il admet un rayon.

Application à la coloration de graphes.

Définition

Soit $G = (S, A)$ un graphe. Une k-coloration de G est une fonction $f : S \to \{0, \dots, k-1\}$ telle que pour $\{s, t\} \in A$, $f(s) \neq f(t)$. On dit que G est k-coloriable s'il existe une k-coloration de G.

7. Montrer qu'un graphe dénombrable G est k-coloriable si et seulement si tous ses sous-graphes finis sont k-coloriables.

Application aux mots infinis

Définition

Soit Σ un alphabet. On note Σ^∞ l'ensemble des mots infinis sur Σ. Pour $u \in \Sigma^\infty$, un **découpage** de u est une suite infinie de mots finis non vides $(v_i)_{i\in\mathbb{N}} \in (\Sigma^+)^{\mathbb{N}}$ telle que $u = v_0 v_1 v_2 \dots$

8. Soit $L \subseteq \Sigma^*$ un langage quelconque et $u \in \Sigma^\infty$. Montrer qu'il existe un découpage $(v_i)_{i\in\mathbb{N}}$ tel qu'à partir d'un certain rang, soit tous les v_i appartiennent à L, soit aucun v_i n'appartient à L.

Corrigé

Le théorème à montrer en question 6 est appelé lemme de König.

1. Soit (s_i) un rayon. Alors pour tout $n \in \mathbb{N}$, $(s_0, \dots, s_n)$ est un chemin simple, donc $|S| \geqslant n + 1$. On en déduit que $|S|$ est infini.
2. Le graphe $(\mathbb{N}, A)$ où $A = \{\{i, i+1\}, i \in \mathbb{N}\}$ est infini, connexe et localement borné. Si à ce graphe, on rajoute i sommets adjacents à chaque sommet i, le graphe résultant est infini, connexe, localement fini (le degré de i est majoré par $i+2$), mais pas localement borné (pour $i > 0$, le degré de i est égal à $i+2$).
3. Chaque propriété est nécessaire pour que le théorème soit vrai, au sens où si on en supprime une, le résultat peut être faux. En effet :
   - le graphe $(\mathbb{N}, \emptyset)$ est infini et localement fini, mais non connexe et ne possède pas de rayon (par contre, il existe des graphes non connexes, infinis et localement finis qui possèdent un rayon) ;
   - la question 1 montre que tout graphe possédant un rayon est infini ;

– le graphe $(\mathbb{N}, A)$ où $A = \{(0, i) \mid i \in \mathbb{N}^*\}$ est connexe, infini, mais pas localement fini et ne possède pas de rayon (par contre, il existe des graphes connexe, infini et pas localement finis qui possèdent un rayon).

4. Il suffit de considérer un graphe qui est l'union disjointe de chemins de longueurs k, pour tout $k \in \mathbb{N}$. Il ne possède pas de chemin infini.
5. G étant localement fini, $G[S \setminus \{s\}]$ possède un nombre fini de composantes connexes (au plus le degré de s). Chacune de ces composantes connexes est localement finie (car le degré de chaque sommet n'a pu que diminuer). L'une d'entre elle est infinie (sinon G serait fini). Cette composante vérifie donc (K).
6. On construit le chemin par récurrence : on part d'un sommet s_0 quelconque. Par la question précédente, $G[S\setminus\{s_0\}]$ possède une composante connexe vérifiant (K). On choisit s_1 un sommet de cette composante, et on continue de la même manière pour construire la suite infinie qui est bien un chemin simple infini.
7. On remarque que le sens direct est immédiat, car on restreint la fonction.

   Pour le sens réciproque, sans perte de généralité, supposons que $G = (\mathbb{N}, A)$ (G a $\mathbb{N}$ pour ensemble de sommets).

   Pour $i \in \mathbb{N}$, on pose $G_i = G[[\![0, i]\!]]$ (le sous-graphe fini de G induit par $[\![0, i]\!]$). On pose $F_i = \{f : [\![0, i]\!] \to [\![0, k-1]\!] \mid f \text{ est une } k\text{-coloration de } G_i\}$ et $F = \bigcup_{i\in\mathbb{N}} F_i$. On définit alors le graphe $G' = (F, A')$ tel que pour $f, g \in F^2$, $\{f, g\} \in A'$ si et seulement s'il existe $i \in \mathbb{N}$ tel que $f \in F_i$, $g \in F_{i+1}$ et $g_{|[\![0,i]\!]} = f$. Alors :
   – G' est dénombrable, car F l'est (chaque ensemble F_i est fini) ;
   – G' est localement borné donc localement fini, car chaque sommet est de degré au plus $k+1$ (une seule restriction possible, et k extensions possibles) ;
   – G' possède au plus k composantes connexes (celles des fonctions qui à 0 associe une valeur de $[\![0, k-1]\!]$).

   L'une de ces composantes connexes vérifie donc la propriété (K), donc possède un rayon. Ce rayon converge vers une k-coloration de G (toutes les k-colorations de ce rayons coïncident pour chaque sommet de G sur lesquels elles sont définies).
8. On pose $u = a_1a_2a_3\ldots$, où les $a_i \in \Sigma$. Pour $1 \leqslant i < j$, on pose $u_{ij} = a_ia_{i+1}\ldots a_{j-1}$. On définit alors le graphe $G = (\mathbb{N}, A)$, où A est défini de la manière suivante : pour $j \in \mathbb{N}$, $\{i, j\} \in A$ si $u_{ij} \in L$ et i est maximal pour cette propriété. S'il n'existe pas de tel i, alors $\{0, j\} \in A$.

   Dès lors, G est connexe et infini. Par le lemme de König, soit il possède un rayon, qui forme un découpage de u tel que tous les mots sont dans L à partir du rang 1, soit il possède un sommet i de degré infini. Soient $j_1, j_2, \ldots$, les sommets adjacents à i qui sont supérieurs à i. Par définition de A, chaque mot $u_{j_kj_{k+1}} \notin L$ (car i est maximal pour que $u_{ij_{k+1}} \in L$). Dès lors, $u = u_{1j_1}u_{j_1j_2}u_{j_2j_3}\ldots$ est un découpage de u tel qu'à partir du rang 1, tous les mots sont dans $\overline{L}$.

Exercice 5: Décidabilité de l'arithmétique de Presburger

On considère dans cet exercice la logique de Presburger, c'est-à-dire la logique du premier ordre sur le langage $\{0, 1, +, =\}$. Formellement, un **terme** est défini par induction comme une constante (0 ou 1), une variable (x, y, x_1, x_2, ...) ou une addition ($+$) entre deux termes. Une **formule** est définie par induction comme une égalité entre deux termes ($=$), une négation ($\neg$), conjonction ($\wedge$), disjonction ($\vee$), implication ($\to$) de formules, ou une quantification existentielle ($\exists$) ou universelle ($\forall$) d'une formule.

Par exemple, φ_0 définie par $\forall x\, \exists y\, (x = 0 \vee x = y + 1)$ et φ_1 définie par $\forall x\, \forall y\, \forall z\, (x + y) + z = x + (y + z)$ sont des formules en logique de Presburger.

Pour réaliser des démonstrations en arithmétique de Presburger, on considère les règles données en annexes, correspondant aux règles de la logique classique auxquelles on ajoute des axiomes et le raisonnement par récurrence.

Question 1 Montrer que les règles suivantes (où t, u et v désignent des termes) sont dérivables à partir des règles $(=_i)$ et $(=_e)$:

$$\frac{\Gamma \vdash u = v}{\Gamma \vdash t[x := u] = t[x := v]} =_c \qquad \frac{\Gamma \vdash t = u}{\Gamma \vdash u = t} =_s \qquad \frac{\Gamma \vdash t = u \quad \Gamma \vdash u = v}{\Gamma \vdash t = v} =_t$$

Question 2 Montrer que la règle (R_2) est dérivable à partir des règles de la logique classique et des autres règles de l'arithmétique de Presburger.

Question 3 Montrer que la formule φ_1 est démontrable. On ne demande pas l'arbre de preuve complet, mais un résumé des différentes étapes de la preuve.

Pour la suite, on cherche à démontrer que l'arithmétique de Presburger est décidable, c'est-à-dire qu'il existe un algorithme qui, étant donnée une formule en logique de Presburger, détermine si elle est démontrable ou non. On introduit pour cela une sémantique : on dit qu'une formule φ est **vraie dans** $\mathbb{N}$, et on note $\mathbb{N} \vDash \varphi$, si la formule est vraie en considérant les variables quantifiées comme appartenant à $\mathbb{N}$. Pour une formule non close $\varphi(x_1, \ldots, x_n)$ et des entiers $a_1, \ldots, a_n \in \mathbb{N}^n$, on dit que φ est **vraie dans** $\mathbb{N}$ **aux valeurs** $a_1, \ldots, a_n$, et on note $\mathbb{N} \vDash \varphi(a_1, \ldots, a_n)$, si elle est vraie en substituant chaque x_i par l'entier a_i. On admet que la logique de Presburger est **complète** pour cette sémantique, c'est-à-dire que si $\mathbb{N} \vDash \varphi$, alors φ est démontrable.

À un n-uplet d'entiers $(a_1, \ldots, a_n) \in \mathbb{N}^n$, on associe un mot de Σ^*, $\Sigma = \{0,1\}^n$, noté également $(a_1, \ldots, a_n)$ par abus de notation, défini comme le mot $(a_1^{(0)}, \ldots, a_n^{(0)})(a_1^{(1)}, \ldots, a_n^{(1)}) \ldots (a_1^{(m)}, \ldots, a_n^{(m)})$, où $(a_i^{(0)} a_i^{(1)} \ldots a_i^{(m)})_2$ correspond à la décomposition binaire de a_i, en commençant par les bits de poids faible. On suppose qu'on rajoute autant de 0 que nécessaire pour que toutes les écritures binaires soient de même taille. Par exemple, au triplet $(2, 7, 1)$, on associe le mot $(0,1,1)(1,1,0)(0,1,0)$.

Pour $x_1, \ldots, x_n$ des variables, on appelle **formule élémentaire** une formule de la forme $x_i = 0$, $x_i = 1$, $x_i = x_j$ ou $x_i + x_j = x_k$. On dit qu'une formule $\varphi(x_1, \ldots, x_n)$ est **rationnelle** si le langage sur Σ $L(\varphi) = \{(a_1, \ldots, a_n) \in \Sigma^* \mid \mathbb{N} \vDash \varphi(a_1, \ldots, a_n)\}$ est rationnel.

Question 4 Montrer que les formules élémentaires sont rationnelles.

Question 5 Montrer qu'une formule obtenue par conjonctions, disjonctions, négations et implications de formules élémentaires est rationnelle. En déduire que toute formule sans quantificateur est rationnelle.

Question 6 On suppose $\varphi(x_1, \ldots, x_n)$ rationnelle. Montrer que $\exists x_1\, \varphi(x_1, \ldots, x_n)$ est rationnelle.

Question 7 En déduire que toute formule de la logique de Presburger est rationnelle, puis que la logique de Presburger est décidable.

Annexe : règles d'inférence de la logique de Presburger

Introduction et élimination

Opérateur	Introduction	Élimination
Implication $\to$	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi} \to_i$	$\dfrac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \to_e$
Conjonction $\wedge$	$\dfrac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \wedge_i$	$\dfrac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \wedge_e^g$ et $\dfrac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi} \wedge_e^d$
Disjonction $\vee$	$\dfrac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi} \vee_i^g$ et $\dfrac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi} \vee_i^d$	$\dfrac{\Gamma \vdash \varphi \vee \psi \quad \Gamma, \varphi \vdash \sigma \quad \Gamma, \psi \vdash \sigma}{\Gamma \vdash \sigma} \vee_e$
Négation $\neg$	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi} \neg_i$	$\dfrac{\Gamma \vdash \varphi \quad \Gamma \vdash \neg\varphi}{\Gamma \vdash \bot} \neg_e$
Atomiques $\top$ et $\bot$	$\dfrac{}{\Gamma \vdash \top} \top_i$	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash \varphi} \bot_e$
Égalité $=$	$\dfrac{}{\Gamma \vdash t = t} =_i$	$\dfrac{\Gamma \vdash \varphi[x := t] \quad \Gamma \vdash t = u}{\Gamma \vdash \varphi[x := u]} =_e$
Existentiel $\exists$	$\dfrac{\Gamma \vdash \varphi[x := t]}{\Gamma \vdash \exists x \varphi} \exists_i$	$\dfrac{\Gamma \vdash \exists x \varphi \quad \Gamma, \varphi \vdash \psi \quad x \notin V_F(\Gamma) \cup V_F(\psi)}{\Gamma \vdash \psi} \exists_e$
Universel $\forall$	$\dfrac{\Gamma \vdash \varphi \quad x \notin V_F(\Gamma)}{\Gamma \vdash \forall x \varphi} \forall_i$	$\dfrac{\Gamma \vdash \forall x \varphi}{\Gamma \vdash \varphi[x := t]} \forall_e$

où x désigne une variable, t et u des termes et $V_F(\Gamma)$ et $V_F(\psi)$ désignent les ensembles des variables libres (*Free*) des formules de Γ et de ψ respectivement.

Autres règles

Nom de la règle	Description
Axiome	$\dfrac{}{\Gamma, \varphi \vdash \varphi}$ ax
Affaiblissement	$\dfrac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi}$ aff
Tiers exclu	$\dfrac{}{\Gamma \vdash \varphi \vee \neg\varphi}$ te

On ajoute les axiomes suivants :

- 0 n'est pas un successeur : $\dfrac{}{\Gamma \vdash \forall x \neg(0 = x + 1)} R_1$
- tout entier non nul est un successeur : $\dfrac{}{\Gamma \vdash \forall x\, (x = 0 \vee \exists y\, x = y + 1)} R_2$
- l'incrémentation est inversible : $\dfrac{}{\Gamma \vdash \forall x \forall y\, (x + 1 = y + 1) \to (x = y)} R_3$

– 0 est l'élément neutre pour l'addition : $\dfrac{}{\Gamma \vdash \forall x\,(x+0=x)}\ R_4$

– l'incrémentation est associative : $\dfrac{}{\Gamma \vdash \forall x\,\forall y\,(x+y)+1=x+(y+1)}\ R_5$

Enfin, on ajoute une infinité dénombrable de règles exprimant le principe de récurrence : pour toute formule $\varphi(x, y_1, \ldots, y_n)$ ayant pour variables libres x, y_1, …, y_n :

$$\frac{\Gamma \vdash \varphi(0, y_1, \ldots, y_n) \qquad \Gamma \vdash \forall x\,(\varphi(x, y_1, \ldots, y_n) \to \varphi(x+1, y_1, \ldots, y_n))}{\Gamma \vdash \forall x\,\varphi(x, y_1, \ldots, y_n)}\ \text{rec}$$

Corrigé

Question 1 On pose φ la formule $t[x := u] = t$. On obtient alors l'arbre de preuve suivant :

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash t[x:=u] = t[x:=u]}\ =_i}{\dfrac{\Gamma \vdash \varphi[x:=u] \qquad \Gamma \vdash u = v}{\Gamma \vdash \varphi[x:=v]}\ =_e}}{\Gamma \vdash t[x:=u] = t[x:=v]}$$

Pour la deuxième règle, on pose φ la formule $x = t$. On a :

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash t = t}\ =_i}{\dfrac{\Gamma \vdash \varphi[x:=t] \qquad \Gamma \vdash t = u}{\Gamma \vdash \varphi[x:=u]}\ =_e}}{\Gamma \vdash u = t}$$

Enfin, pour la troisième règle, c'est une application directe de $=_e$: on pose φ la formule $t = x$. On a alors :

$$\dfrac{\dfrac{\Gamma \vdash t = u}{\dfrac{\Gamma \vdash \varphi[x:=u] \qquad \Gamma \vdash u = v}{\Gamma \vdash \varphi[x:=v]}\ =_e}}{\Gamma \vdash t = v}$$

Question 2 On a l'arbre de preuve :

$$\dfrac{\dfrac{}{\vdash 0 = 0}\ =_i}{\vdash 0 = 0 \vee \exists y\,(0 = y+1)}\ \vee_i \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash x+1 = x+1}\ =_i}{\vdash \exists y\,(x+1 = y+1)}\ \exists_i}{\vdash x+1 = 0 \vee \exists y\,(x+1 = y+1)}\ \vee_i}{x = 0 \vee \exists y\,(x = y+1) \vdash x+1 = 0 \vee \exists y\,(x+1 = y+1)}\ \text{aff}}{\vdash (x = 0 \vee \exists y\,(x = y+1)) \to (x+1 = 0 \vee \exists y\,(x+1 = y+1))}\ \to_i$$
$$\vdash \forall x\,(x = 0 \vee \exists y\,(x = y+1))\ \text{rec}$$

Ce qui montre bien que la règle (R_2) est dérivable à partir des autres.

Question 3 On va faire une preuve par récurrence. On commence par l'initialisation :

$$\dfrac{\dfrac{\dfrac{}{\vdash \forall z\,(z+0=z)}\,R_4}{\vdash (x+y)+0=x+y}\,\forall_e \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \forall z\,(z+0=z)}\,R_4}{\vdash y+0=y}\,\forall_e}{\vdash (x+z)[z:=y+0]=(x+z)[z:=y]}\,=_c}{\vdash x+(y+0)=x+y}\,=_s}{\vdash (x+y)+0=x+(y+0)}\,=_t$$

Puis l'hérédité, en notant φ la formule $(x+y)+z=x+(y+z)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \forall y\,\forall z\,(y+z)+1=y+(z+1)}\,R_5}{\vdash (y+z)+1=y+(z+1)}\,\forall_e\times 2}{\vdash x+((y+z)+1)=x+(y+(z+1))}\,=_c \qquad \dfrac{\dfrac{\dfrac{}{\vdash \forall x\,\forall y\,(x+y)+1=x+(y+1)}\,R_5}{\vdash \forall y\,(x+y)+1=x+(y+1)}\,\forall_e}{\vdash (x+(y+z))+1=x+((y+z)+1)}\,\forall_e}{\vdash (x+(y+z))+1=x+(y+(z+1))}\,=_t$$
$$\dfrac{\vdash (x+(y+z))+1=x+(y+(z+1))}{\varphi\vdash (x+(y+z))+1=x+(y+(z+1))}\,\text{aff}$$

puis :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \forall x\,\forall z\,(x+z)+1=x+(z+1)}\,R_5}{\vdash \forall z\,((x+y)+z)+1=(x+y)+(z+1)}\,\forall_e}{\vdash ((x+y)+z)+1=(x+y)+(z+1)}\,\forall_e}{\varphi\vdash ((x+y)+z)+1=(x+y)+(z+1)}\,\text{aff}}{\varphi\vdash (x+y)+(z+1)=((x+y)+z)+1}\,=_s \qquad \dfrac{\dfrac{}{\varphi\vdash\varphi}\,\text{ax}}{\varphi\vdash ((x+y)+z)+1=(x+(y+z))+1}\,=_c}{\varphi\vdash (x+y)+(z+1)=(x+(y+z))+1}\,=_t$$

et en regroupant les deux preuves précédentes :

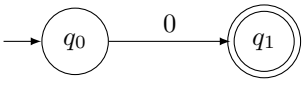
$$\dfrac{\dfrac{\varphi\vdash (x+(y+z))+1=x+(y+(z+1)) \qquad \varphi\vdash (x+y)+(z+1)=(x+(y+z))+1}{\varphi\vdash (x+y)+(z+1)=x+(y+(z+1))}\,=_t}{\vdash \varphi\to (x+y)+(z+1)=x+(y+(z+1))}\,\to_i$$
$$\dfrac{}{\vdash \forall z\,((x+y)+z=x+(y+z)\to (x+y)+(z+1)=x+(y+(z+1)))}\,\forall_i$$

On conclut alors :

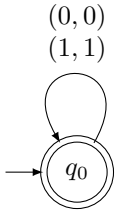
$$\dfrac{\dfrac{\vdash (x+y)+0=x+(y+0) \qquad \vdash \forall z\,(x+y)+z=x+(y+z)\to (x+y)+(z+1)=x+(y+(z+1))}{\vdash \forall z\,(x+y)+z=x+(y+z)}\,\text{rec}}{\vdash \varphi_1}\,\forall_i\times 2$$

Question 4 On commence par des cas particuliers :

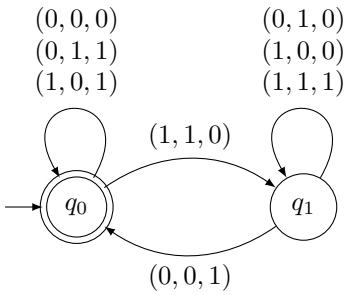
- $x_1=0$, pour $n=1$, est rationnelle, car $L(x_1=0)$ est reconnu par :



  De même, $x_1=1$ est rationnelle.
- $x_1=x_2$, pour $n=2$, est rationnelle, car $L(x_1=x_2)$ est reconnu par :



– $x_1 + x_2 = x_3$, pour $n = 3$, est rationnelle, car $L(x_1 + x_2 = x_3)$ est reconnu par :



L'état q_0 correspond à l'état sans retenue, et l'état q_1 est l'état qui permet de garder en mémoire la propagation de retenue.

On étend naturellement ces deux solutions à toutes les formules élémentaires en gardant ces transitions pour les variables apparaissant dans la formule et en rajoutant dans chaque uplet toutes les valeurs possibles pour les autres composantes.

Question 5 On peut montrer que $L(\varphi \wedge \psi) = L(\varphi) \cap L(\psi)$, $L(\varphi \vee \psi) = L(\varphi) \cup L(\psi)$, $L(\neg\varphi) = \overline{L(\varphi)}$ et $L(\varphi \to \psi) = L(\neg\varphi \vee \psi)$. Les langages rationnels étant clos par union, intersection et complémentaire, on obtient le résultat voulu.

Par ailleurs, une formule de la forme $t_1 + t_2 + \ldots + t_p = t_{p+1} + \ldots + t_q$, où les t_i sont des variables ou des constantes, peut s'écrire comme une conjonction de formules élémentaires :

– pour chaque t_i qui est une constante, on la remplace par une variable y_i et on rajoute une clause $y_i = t_i$;
– en rajoutant des variables intermédiaires, on peut réduire le nombre d'additions de chaque côté de l'égalité. Par exemple, $y_1 + y_2 + y_3 = y_4 + y_5$ est sémantiquement équivalente à $y_1 + z_2 = z_4 \wedge y_2 + y_3 = z_2 \wedge y_4 + y_5 = z_4$.

En combinant ce résultat avec le précédente, on en déduit que toute formule sans quantificateur est rationnelle.

Question 6 Supposons $\varphi(x_1, \ldots, x_n)$ rationnelle et soit $A = (Q, \Sigma = \{0,1\}^n, \delta, q_0, F)$ un AFD reconnaissant $L(\varphi)$. On pose B l'automate non déterministe défini par $B = (Q, \Sigma' = \{0,1\}^{n-1}, \Delta, \{q_0\}, F')$ tel que :

– pour un état $q \in Q$ et une lettre $(b_1, \ldots, b_{n-1}) \in \Sigma'$, on pose $\Delta(q, (b_1, \ldots, b_{n-1})) = \{\delta(q, (b_1, \ldots, b_{n-1}, 0)), \delta(q, (b_1, \ldots, b_{n-1}, 1)\}$, autrement dit on supprime la dernière composante des lettres dans les transitions de A ;
– $F' = \{q \in Q \mid \exists p \in F \mid \exists u \in (\{0\}^{n-1} \times \{0,1\})^* \mid \delta^*(q, u) = p\}$, autrement dit les états finaux dans B sont ceux qui permettent d'atteindre un état final de A en lisant un mot dont les lettres ne contiennent que des 0 sur les $n-1$ premières composantes (pour gérer le cas où x_n correspond à l'entier a_n avec l'écriture binaire la plus grande).

Montrons que B reconnait $L(\exists x_n\, \varphi)$:

– supposons $(a_1, \ldots, a_{n-1}) \in L(\exists x_n\, \varphi)$. Alors il existe $a_n \in \mathbb{N}$ tel que $\mathbb{N} \vDash \varphi(a_1, \ldots, a_n)$. On en déduit que $(a_1, \ldots, a_n) \in L(\varphi)$. Distinguons deux cas :
  * si $a_n \neq \max\{a_1, \ldots, a_n\}$, alors $\delta^*(q_0, (a_1, \ldots, a_n)) \in F \cap \Delta^*(q_0, (a_1, \ldots, a_{n-1})) \subseteq F' \cap \Delta^*(q_0, (a_1, \ldots, a_{n-1}))$ donc $(a_1, \ldots, a_{n-1}) \in L(B)$;
  * sinon, $\delta^*(q_0, (a_1, \ldots, a'_n)) \in F' \cap \Delta^*(q_0, (a_1, \ldots, a_{n-1}))$, où a'_n correspond à l'entier dont l'écriture binaire est celle de a_n, tronquée au nombre de bits significatifs le plus grand parmi $a_1, \ldots, a_{n-1}$. À nouveau, on a $(a_1, \ldots, a_{n-1}) \in L(B)$.
– la réciproque se fait sur la même idée : si $(a_1, \ldots, a_{n-1}) \in L(B)$, alors il existe un chemin acceptant dans B. On peut lui associer un chemin acceptant dans A dont les $n-1$ premières composantes des lettres lues correspondent à $(a_1, \ldots, a_{n-1})$. On en déduit qu'il existe $a_n \in \mathbb{N}$ tel que $\mathbb{N} \vDash \varphi(a_1, \ldots, a_n)$, et donc que $\mathbb{N} \vDash \exists x_n\, \varphi(a_1, \ldots, a_{n-1}, x_n)$.

Question 7 On traite le cas $\forall x_n\, \varphi(x_1, \ldots, x_n)$ en remarquant qu'une telle formule est sémantiquement équivalente à $\neg(\exists x_n\, \neg\varphi(x_1, \ldots, x_n))$. De plus, la conjonction, disjonction, négation et implication de formules non élémentaires se traite de la même manière que les formules élémentaires. On conclut par induction sur

les formules que toute formule de la logique de Presburger est rationnelle.

On obtient alors un automate sur l'alphabet $\Sigma = \{0,1\}^0$ (c'est-à-dire dont les lettres sont des 0-uplets). Il suffit alors de remarquer que $\mathbb{N} \vDash \varphi \Leftrightarrow L(\varphi) \neq \emptyset$. Or, étant donné un automate fini (déterministe ou non), on peut tester si le langage reconnu est vide par un parcours de graphe : il faut vérifier s'il existe un chemin depuis un état initial vers un état final. Tout cela donne bien un algorithme permettant de tester si une formule est vraie dans $\mathbb{N}$. Comme on a admis la complétude de la logique de Presburger pour cette sémantique, on en déduit que la logique de Presburger est décidable.

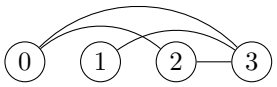
Exercice 6: Graphes de saut

Définition

Un **graphe de saut** de longueur $n \geqslant 1$ est un graphe non-orienté $G = (S, A)$ tel que $S = [\![0, n-1]\!]$ et tel que pour tous $a \leqslant b < c \leqslant d < n$, $\{b, c\} \in A \Rightarrow \{a, d\} \in A$.

Exemple

Voici un graphe de saut de longueur 4.



1. Dessiner tous les graphes de saut de longueur 1, 2 et 3.
2. Proposer un algorithme en pseudo-code qui prend en argument la matrice d'adjacence d'un graphe et détermine si ce graphe est un graphe de saut. Déterminer sa complexité en temps et en espace.

Définition

Un graphe de saut de longueur n est dit **comprimé** si pour tout $i < n-1$, les sommets i et $i+1$ ne sont pas adjacents.

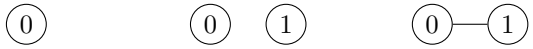
3. Montrer que les graphes de saut comprimés de longueur $n+1$ sont en bijection avec les graphes de saut de longueur n.
4. Soit P_n le nombre de graphes de saut de longueur n. On pose par convention $P_0 = 1$. Déterminer une formule de récurrence pour calculer P_{n+1} en fonction de $P_0, \ldots, P_n$.

Le but des questions suivantes est de considérer s'il existe une permutation des sommets transformant un graphe quelconque en un graphe de saut.

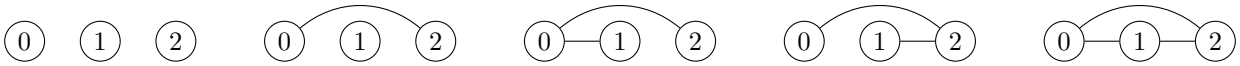
5. Décrire en français un algorithme qui prend en argument un tableau T de longueur n représentant une permutation de 0 à $n-1$ inclus et modifie et renvoie T pour qu'il représente la prochaine permutation suivant un certain ordre total $\prec$ sur les permutations. Si T est la dernière permutation de cet ordre, alors le programme renvoie un tableau vide.
6. Décrire en français un algorithme naïf prenant en argument un entier $n \geqslant 0$ et une matrice d'adjacence M de taille $n \times n$ et décide s'il existe une permutation de $\{0, \ldots, n-1\}$ telle que le graphe résultant est un graphe de saut. Déterminer sa complexité en temps et en espace.
7. Montrer que pour tout graphe $G = (\{0,1,2\}, A)$, il existe une permutation de $\{0,1,2\}$ telle que le graphe résultant est un graphe de saut.
8. Déterminer un graphe $G = (\{0,1,2,3\}, A)$ dont aucune permutation ne résulte en un graphe de saut.
9. Que dire de l'existence de graphes non connexes tels qu'il existe une permutation des sommets donnant un graphe de saut ?

Corrigé

1. Les graphes de saut de longueur 1 et 2 sont :



Les graphes de saut de longueur 3 sont :



2. On commence par remarquer que la définition est équivalente à : $\forall b < c, \{b, c\} \in E \Rightarrow (b = 0$ ou $\{b-1, c\} \in E)$ et $(c = n-1$ ou $\{b, c+1\} \in E)$. Il est clair que la définition initiale implique cette définition. La réciproque se fait par récurrence descendante sur b et ascendante sur c.
Cela permet d'imaginer un algorithme naïf de complexité temporelle quadratique et de complexité spatiale constante :

```
Algorithme : Graphe_de_saut
Entrée : matrice d'adjacence M de taille n fois n
Pour b = 0 à n - 1 Faire
   Pour c = b à n - 1 Faire
      Si M[b][c] = 1
         Si (b > 0 et M[b-1][c] = 0) ou (c < n - 1 et M[b][c+1] = 0)
            Renvoyer Faux
Renvoyer Vrai
```

3. On considère f la fonction qui prend en argument un graphe de saut comprimé $G = (\llbracket 0, n \rrbracket, E)$ de longueur $n+1$ en un graphe de saut $G' = (\llbracket 0, n-1 \rrbracket, E')$ de longueur n de la manière suivante :

$$E' = \{\{a, b-1\}, a < b \text{ et } \{a, b\} \in E\}$$

G étant comprimé, $\{a, b-1\}$ n'est jamais un singleton, donc f est bien définie. Montrons que G' est bien un graphe de saut. Soit $\{b, c\} \in E'$, $b < c$ et (a, d) tels que $a \leqslant b$ et $c \leqslant d$. Alors $\{b, c+1\} \in E$ par définition de E'. G étant un graphe de saut, on en déduit $\{a, d+1\} \in E$, ce qui implique $\{a, d\} \in E'$ et G' est bien un graphe de saut.
En posant g la fonction qui à un graphe de saut $G' = (\llbracket 0, n-1 \rrbracket, E')$ associe $G = (\llbracket 0, n \rrbracket, E)$ en posant :

$$E = \{\{a, b\}, a < b \text{ et } \{a, b-1\} \in E'\}$$

il est clair que $g(G')$ est un graphe de saut comprimé (car $\{a, b-1\} \in E' \Rightarrow a < b-1$) et que g est la réciproque de f.

4. Soit $n \in \mathbb{N}$ et soit $G = (\llbracket 0, n \rrbracket, E)$ un graphe de saut de taille $n+1$. Soit $i \leqslant n$ le plus petit sommet tel que $\{i, i+1\} \in E$, ou $i = n$ si un tel sommet n'existe pas. Dès lors, le sous-graphe de G induit par $\llbracket 0, i \rrbracket$ est un graphe de saut de longueur $i+1$, comprimé par définition de i. Le sous-graphe de G induit par $\llbracket i+1, n \rrbracket$ est un graphe de saut de longueur $n-i$. De plus, toutes les arêtes entre un sommet de $\llbracket 0, i \rrbracket$ et un sommet de $\llbracket i+1, n \rrbracket$ existent car $\{i, i+1\} \in E$ ou le deuxième ensemble est vide. Enfin, par la question précédente, le sous-graphe induit par $\llbracket 0, i \rrbracket$ est en bijection avec un graphe de saut de longueur i (car il est comprimé). On en déduit que le nombre de graphes de saut de longueur $n+1$ est $P_i P_{n-i}$ pour i fixé.

Finalement, la formule de récurrence est $P_{n+1} = \sum_{i=0}^{n} P_i P_{n-i}$.

5. On considère $\prec$ comme l'ordre lexicographique sur les images de $0, \ldots, n-1$. Il est clair que c'est un ordre total. En écrivant les premières permutations pour $n = 4$, essayons de déterminer un algorithme pour passer d'une permutation à la suivante :

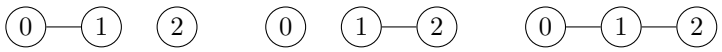
$$0123 \quad 0132 \quad 0213 \quad 0231 \quad 0312 \quad 0321 \quad 1023$$

On remarque que 0 reste en première position pour les 6 premières permutations. Le moment où 0 est remplacé a lieu lorsque les chiffres qui suivent sont par ordre décroissant (sinon il existerait une autre permutation commençant par 0 plus grande selon l'ordre lexicographique). Dès lors, on remplace 0 par le plus petit des chiffres qui suit parmi ceux qui sont plus grands que 0, et on réordonne les chiffres par ordre croissant. On propose alors l'algorithme suivant :

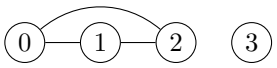
- Poser j le plus grand indice tel que $T[j] < T[j+1]$.
- Si j n'est pas défini, renvoyer le tableau vide.
- Sinon poser $k > j$ le plus grand indice tel que $T[j] < T[k]$ (k est bien défini, car il vaut au pire $j+1$).
- Permuter $T[j]$ et $T[k]$.
- Retourner le sous-tableau $T[j+1 : n]$ (car il était par ordre décroissant) et renvoyer T.

6. On propose alors l'algorithme suivant :
   - Poser T le tableau $[0, 1, \ldots, n-1]$.
   - Tant que T n'est pas le tableau vide, faire les deux instructions suivantes :
   - Si le graphe G auquel on applique la permutation T est un graphe de saut, renvoyer Vrai.
   - Transformer T par l'algorithme précédent.
   - Renvoyer Faux.

   La complexité en temps est $\mathcal{O}(n!n^2)$ et celle en espace est $\mathcal{O}(n)$.
7. Il suffit d'étudier les graphes sur $\{0, 1, 2\}$ qui ne sont pas des graphes de saut, à savoir :



   Dans le premier et le dernier cas, il suffit d'intervertir 1 et 2. Dans le deuxième cas, il suffit d'intervertir 0 et 1.
8. Le seul graphe qui convient est :



   En effet, 3 n'étant lié ni à 0, ni à 1 qui sont liés, le sommet 3 doit apparaître entre les sommets 0 et 1. On raisonne de même pour 1 et 2 et 0 et 2, ce qui est impossible.
9. Si un tel graphe existe, alors ses composantes connexes sauf au plus une doivent être des singleton (sinon chaque sommet d'une C.C. de taille > 1 doit se trouver entre deux sommets reliés d'une autre C.C. de taille > 1 ce qui n'est pas possible). De plus, la C.C. non triviale doit être bipartie : comme il existe une C.C. triviale composée d'un sommet v, pour chaque arête $\{x, y\}$ de la C.C. non triviale, v doit se trouver entre x et y. Cela permet de décomposer la C.C. en deux (les sommets avant et ceux après), et aucune arête ne doit exister entre deux sommets d'un même côté.