Table des matières

Troisième devoir surveillé d'informatique

Durée : quatre heures. Il est recommandé de traiter leux quatre exercices dans l'ordre.

Première partie

Logique

Dans cette partie, $\wedge$, $\vee$, $\neg$, désignent respectivement les connecteurs de conjonction, de disjonction et de négation. Les symboles V, F désignent respectivement les valeurs de vérité VRAI et FAUX du calcul propositionnel.

Dans la suite, on s'intéresse au problème Horn-Sat, qui peut être décrit de la façon suivante : étant donnée une formule de Horn P, existe-t-il une interprétation I telle que $[P]_I = V$?

L'objectif est d'expliciter un algorithme permettant de résoudre ce problème.

1 Définitions

Définition 1.1. *(Formule propositionnelle)*

Soit X un ensemble de symboles appelés variables propositionnelles. Les formules propositionnelles sur X sont alors définies de façon inductive comme suit :

- *V, F sont des formules propositionnelles ;*
- *tout élément x de X est une formule propositionnelle ;*
- *si P et Q sont des formules propositionnelles, alors $\neg P$, $(P \wedge Q)$, $(P \vee Q)$ sont des formules propositionnelles.*

Dans la suite, les variables propositionnelles et formules propositionnelles considérées sont construites à l'aide d'un ensemble X qui ne sera pas explicité.

Définition 1.2. *(Littéral)*

Soit x une variable propositionnelle. Un littéral est la formule x ou la formule $\neg x$. Le littéral x (respectivement $\neg x$) est un littéral positif (respectivement négatif).

Définition 1.3. *(Clause disjonctive)*

Soient $n \in \mathbb{N}$, x_1 , x_2 , … , x_n des littéraux. La formule $x_1 \vee x_2 \vee ... \vee x_n$ est appelée clause disjonctive à n littéraux (ou de taille n). Par convention, $()$ désigne la clause disjonctive vide, c'est-à-dire la clause sans littéral.

Une **clause de Horn** est une clause disjonctive contenant au plus un littéral positif.
Une **clause unitaire** est une clause composée d'un unique littéral.

Définition 1.4. *(Forme normale conjonctive)*
Soit P une formule propositionnelle. On dit que P est une forme normale conjonctive s'il existe un entier $n \in \mathbb{N}$, $C_1, C_2, ..., C_n$ des clauses disjonctives vérifiant $P = C_1 \wedge C_2 \wedge ... \wedge C_n$. Par convention, une forme normale conjonctive sans clause est représentée par $\emptyset$.

Définition 1.5. *(formule de Horn)*
Soit P une formule propositionnelle. On dit que P est une formule de Horn s'il existe $n \in \mathbb{N}$, $C_1, C_2, ..., C_n$ des clauses de Horn vérifiant $P = C_1 \wedge C_2 \wedge ... \wedge C_n$.

Définition 1.6. *(Interprétation)*
Soient $n \in_N$ et $x_1, x_2, ...x_n$ des variables propositionnelles. Une interprétation est une application $I : \{x_1, x_2, ..., x_n\} \to \{V, F\}$.

Définition 1.7. *(Évaluation)*
Soient P une formule propositionnelle, $x_1, x_2, ..., x_n$ les variables propositionnelles apparaissant dans P et I une interprétation sur $\{x_1, x_2, ..., x_n\}$. L'évaluation de P suivant l'interprétation I que l'on note $[P]_I$ est définie par récurrence de la façon suivante :

- *si $P \in V, F$, alors $[P]_I = P$;*
- *si $P \in x_1, x_2, ..., x_n$, alors $[P]_I = I(P)$;*
- *si $P = \neg Q$ où Q est une formule propositionnelle, alors $[P]_I = \neg[Q]_I$;*
- *si $P = A \circ B$ où A, B sont des formules propositionnelles et $\circ \in \wedge, \vee$, alors $[P]_I = [A]_I \circ [B]_I$.*

Par convention, $[()]_I = Fet[\emptyset]_I = V$.

Définition 1.8. *(Satisfiable)*
Soit P une formule propositionnelle. On dit que P est satisfiable s'il existe une interprétation I telle que $[P]_I = V$.

1. Pour chacune des formules suivantes, montrer qu'elle est satisfiable ou qu'elle est non satisfiable.
   (a) $P_1 = (x \vee y) \wedge (\neg x \vee \neg y) \wedge (x \vee \neg y)$,
   (b) $P_2 = (x) \wedge (\neg x \vee \neg y) \wedge (\neg y \vee z) \wedge (z)$,
   (c) $P_3 = ()$,
   (d) $P_4 = (x \vee \neg y \vee \neg t) \wedge (z \vee \neg t \vee \neg x \vee \neg y)$.

   solution : J'utiliserai les notations de l'algèbre de Boole dans les calculs ci-dessous.
   (a)
   $$\begin{aligned} P_1 \equiv & (x+y)(\bar{x}+\bar{y})(x+\bar{y}) \\ \equiv & (x\bar{y} + x + y\bar{x}) \\ \equiv & x\bar{y} \end{aligned}$$
   Ainsi, P_1 est satisfiable : il suffit d'assigner à x la valeur V et à y la valeur F.
   (b) On trouve $P_2 \equiv x\bar{y}z$ donc P_2 est satisfiable.
   (c) La formule () est non satisfiable vu la définition 1.7.
   (d) On trouve $P_4 \equiv xz + \bar{y} + \bar{t}$ donc P_4 est satisfiable. N'importe quelle interprétation où t est fausse convient.
2. Parmi les formules de la question 1, lesquelles sont des formules de Horn ? On ne justifiera pas la réponse.
   solution : Les formules P_2, P_3 et P_4 sont de Horn.

2 Satisfiabilité d'une formule de Horn

Définition 2.1. *(Propagation unitaire)*
Soit P une forme normale conjonctive contenant une clause unitaire (x). On construit une formule P' à partir de P de la façon suivante :
solution : Toute interprétation I qui satisfait P devra vérifier $I(x) = V$.

1. *supprimer de P toutes les clauses où x apparaît ;*
   solution : toute clause contenant x sera automatiquement vérifiée par toute interprétation I telle que $I(x) = V$.
2. *enlever toutes les occurrences du littéral $\neg x$.*
   solution : Toute clause contenant $\bar{x}$ devra être satisfaite par un autre moyen qu'en utilisant $\bar{x}$.

Cette procédure de simplification est appelée propagation unitaire.

Par exemple, si $P = (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z) \wedge (x)$, on a alors P' $= (y \vee z)$.$SiP = (x) \wedge (\neg x)$, on a alors P' $= ()$.

Dans la suite, on note $\Pi(P)$ une formule sans clause unitaire obtenue en itérant la propagation unitaire sur P. On admet que si $\Pi(P)$ ne contient pas de clause vide () alors $\Pi(P)$ est unique et que si $\Pi(P)$ contient au moins une clause vide alors toute formule sans clause unitaire obtenue par itération de la propagation unitaire sur P contient au moins une clause vide.

3. On pose :

   $P = (x_1) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_2 \vee \neg x_1 \vee x_3) \wedge (x_3 \vee \neg x_4 \vee x_5) \wedge (\neg x_1 \vee x_2 \vee x_5)$

   Calculer $\Pi(P)$. On pourra donner le résultat directement sans détailler les calculs.
   solution : On trouve

   $$\begin{aligned} P\text{'} &= (x_3 \vee x_4) \wedge \neg x_2 \wedge (\neg x_2 \vee x_3) \wedge (x_3 \vee \neg x_4 \vee x_5) \wedge (x_2 \vee x_5) \\ P\text{''} &= (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4 \vee x_5) \wedge x_5 \\ P\text{'''} &= (x_3 \vee x_4) \end{aligned}$$

   Il n'y a plus de clause unitaire, donc $\Pi(P) = (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$.

4. Soit P une formule de Horn. Montrer que $\Pi(P)$ est une formule de Horn.
   solution :
   - Quand on supprime un littéral dans une clause de Horn, on obtient une nouvelle clause de Horn.
   - Quand on supprime des clauses de Horn dans une formule de Horn, ou qu'on remplace une clause de Horn par une autre clause de Horn, on obtient une nouvelle formule de Horn.

   À partir de là, pour toute formule Q de Horn contenant une clause unitaire, Q' est une formule de Horn.

   Et alors par récurrence immédiate, $\Pi(P)$ est une formule de Horn.

5. Soit P une forme normale conjonctive. Montrer que P est satisfiable si et seulement si $\Pi(P)$ est satisfiable.
   solution : Dans la suite j'abrégerai « forme normale conjonctive » en « FNC ».
   Il suffit de prouver que pour tout FNC Q contenant un littéral unitaire, Q est satisfiable si et seulement si Q' l'est. En effet le résultat souhaité sera alors obtenu par récurrence immédiate sur le nombre de propagations unitaires nécessaires pour passer de P et $\Pi(P)$.

   Soit donc Q une FNC contenant un littéral unitaire l. Soit x la variable utilisée dans l, donc $l = x$ ou $l = \neg x$. Soit $n \in \mathbb{N}$ et $c_1, ..., c_n$ les clauses de Q. Donc $Q = \bigwedge_{i=1}^{n} c_i$. Soit $m \in \mathbb{N}$ et $d_1, ..., d_m$ les clauses de Q'.

   - Supposons Q satisfiable, et soit I une interprétation telle que $[Q]_I = V$. Donc pour tout $i \in [\![1, n]\!]$, $[c_i]_I = V$. Comme l'un des c_i vaut l, on déduit que $[l]_I = V$.
     Soit $j \in [\![1, m]\!]$. Vérifions que $[d_j]_I = V$. Par définition du la propagation unitaire, il existe $i \in [\![1, n]\!]$ tel que d_j est obtenu à partir de c_i en supprimant un (voir plusieurs) éventuel $\bar{l}$, où $\bar{l}$ désigne $\neg x$ si $l = x$, ou x si $l = \neg x$. Notons $i \in \mathbb{N}$ le nombre de $\bar{l}$ dans c_i, et $k_1, ..., k_j$ les autres littéraux. Donc $c_i \equiv (\bar{l})^i \vee k_1 ... \vee k_j$.
     Dès lors :

     $$\begin{aligned} [c_i]_I &= [\bar{l}]_I^i \vee \left[k_1 \vee ... \vee k_j\right]_I \quad \Big) [l]_= V \textit{ donc } [\bar{l}]_I = F \\ &= F \vee \left[k_1 \vee ... \vee k_j\right]_I \\ &= \left[k_1 \vee ... \vee k_j\right]_I \\ &= [d_j]_I \end{aligned}$$

     Comme I satisfait c_i, nous déduisons qu'elle satisfait d_j.

     Ainsi toutes les clauses de Q' sont satisfaites.
   - Supposons Q' satisfiable. Soit x la variable utilisée dans l. La formule Q' a ses variables dans $X \setminus \{x\}$. Comme elle est satisfiable, il existe une interprétation I_0 sur $X \setminus \{x\}$ qui satisfait Q'. Nous étendons cette interprétation I_0 en une interprétation I sur X en posant $I(x) = V$ si $l = x$ et $I(x) = F$ si $l = \neg x$. De cette manière, $[l]_I = V$.
     Soit $i \in [\![1, n]\!]$. Montrons que $[c_i]_I = V$.
     - ◇ Si c_i contient l, alors $[c_i]_I = V$ car c'est une disjonction dont l'un des éléments est vérifié par I (en l'occurrence l).
     - ◇ Sinon, il existe $j \in [\![1, m]\!]$ tel que d_j est obtenu en supprimant un (ou plusieurs) éventuels $\bar{l}$ dans c_i. Or $[d_j]_I = [d_j]_{I_0} = V$. Donc quel que soit $k \in \mathbb{N}$, $[c_i]_I = [(\bar{l})^k \vee d_j] = [(\bar{l})^k]_I \vee V = V$.

6. Soit P une forme normale conjonctive. Montrer que si () apparaît dans $\Pi(P)$, alors P n'est pas satisfiable.
   solution : Supposons que () apparaît dans $\Pi(P)$. Cette clause n'est pas satisfiable puisque pour toute interprétation I, $[()]_I = F$. Alors $\Pi(P)$ qui est une conjonction contenant () n'est pas satisfiable non plus. Et par la question 5 P n'est pas satisfiable.
7. Soit P une formule de Horn ne contenant ni de clause vide ni de clause unitaire positive. Montrer que P est satisfiable.
   solution : Il suffit de prendre une interprétation I dans laquelle toutes les variables sont fausses. En effet vu les hypothèses, chaque clause contient au moins un littéral négatif.
8. Soit P une formule de Horn. Montrer que P n'est pas satisfiable si et seulement si () apparaît dans $\Pi(P)$.
   solution : Le « si » a été prouvé pour une forme normale disjonctive générale. Voyons le « seulement si ».
   On suppose que () n'apparaît pas dans $\Pi(P)$. La formule $\Pi(P)$ ne contient pas de plus de littéral unitaire, sans quoi on aurait pu appliquer la propagation unitaire une fois de plus. Donc par la question précédente, $\Pi(P)$ est satisfiable, et par 5 P aussi.

3 Programmation

Un littéral sera représenté par le type Caml suivant :

```
type litteral = P of string | N of string;;
```

Pour toute chaîne de caractère x représentant une variable, `P x` représente le littéral positif x, et `N x` le littéral négatif $\neg x$.

Une clause est alors une liste de littéraux, et une formule sous forme normale conjonctive une liste de clauses :

```
type clause = litteral list;;
type fnc = clause list;;
```

On abrégera par la suite « formule sous forme normale disjonctive » en « FNC ».

1. Écrire une fonction pour tester si une FNC est une formule de Horn.
2. Écrire une fonction `propagation_unitaire` prenant une FNC p et un littéral unitaire d'icelle et renvoyant la FNC p' obtenue par propagation unitaire à partir de p.
3. Écrire une fonction `reduite_unitaire` prenant une FNC p et renvoyant $\Pi(P)$.
4. Écrire une fonction satisfiable prenant en entrée une FNC supposée représenter une formule de Horn et indiquant si celle-ci est satisfiable.

Deuxième partie
Tronc commun

L'objectif de cette partie est de proposer une implémentation en langage Python d'une solution au problème de Freudenthal. Hans Freudenthal (1905-1990), mathématicien allemand naturalisé néerlandais, spécialiste de topologie algébrique, est connu pour ses contributions à l'enseignement des mathématiques. En 1969, il soumet à une revue mathématique le problème suivant :

Un professeur dit à ses deux étudiants Sophie et Pierre : « J'ai choisi deux entiers x et y entre 0 et 100, tels que $1 < x < y$ et $x + y \leq n$. J'ai confié à Pierre la valeur Π du produit de x et y. J'ai confié à Sophie la valeur Σ de la somme de x et y. Pierre, Sophie, je vous demande de trouver x et y. »

Pierre et Sophie engagent alors le dialogue suivant :

- *Pierre : "Je ne connais pas les nombres x et y."*
- *Sophie : "Avant même que tu me le dises, je savais déjà que tu ne connaissais pas x et y."*
- *Pierre : "Ah ! eh bien maintenant je connais x et y."*
- *Sophie : "Très bien, mais moi aussi alors maintenant je connais x et y."*

En vous aidant de ce dialogue, c'est maintenant à vous lecteur de trouver x et y.

Dans la suite, on note $\mathcal{N}_n = \left\{ (x, y) \in \mathbb{N}^2 \;\middle|\; 1 < x < y \text{ et } x + y \leq n \right\}$. Si la discussion entre Sophie et Pierre semble stérile, une quantité importante d'informations est cependant échangée qui amène au bout du dialogue à la solution.

1. À quelle condition sur x et y Pierre aurait-il pu dire dès le début : "Je connais x et y" ?
solution : Pierre pouvait connaître x et y ssi il existait un unique couple (a, b) tel que $a < b$ et $\Pi = ab$.

2. Écrire une fonction `CoupleProd(n)` qui renvoie la liste des entiers P pour lesquels il existe au moins deux couples $(x, y) \in \mathcal{N}_n$ tels que $xy = P$. Par exemple, `CoupleProd(9)` renvoie `[12]` puisque $12 = 3 \times 4 = 2 \times 6$ et qu'aucune autre valeur ne satisfait la propriété.
solution : Remarquons que pour tout $(x, y) \in \mathcal{N}_n$, $x \leqslant n - y$ donc $xy \leqslant (n - y) \times y \leqslant \frac{n^2}{4}$ (maximum d'un polynôme de degré 2 de racines 0 et n). Et comme xy est entier, on peut remplacer la division par une division euclidienne. Ceci nous donne une borne sur l'ensemble des P possibles.

```
def coupleProd(n):
    """ Renvoie la liste des P pour lesquels il existe au moins deux couples (x,y)∈N_n
        ↪ tels que x*y==P.
    Après la première ligne du dialogue, nous savons que Π n'est pas parmi ces nombres.
    """

    nbdécomp = [0 for i in range(n*n//2+1) ]

    for x in range(2, n-1):
        for y in range(x+1, n-x+1):
            nbdécomp[x*y] += 1

    return [ P for P in range(n*n//2+1) if nbdécomp[P]>1 ]
```

3. Soit un entier $S \leq n$. Écrire une fonction `Prod(S)` qui renvoie, pour l'ensemble des $(x, y) \in \mathcal{N}_n$ tels que $x + y = S$, la liste des entiers $P = xy$. Par exemple, `Prod(8)` renvoie `[12,15]` puisque $8 = 6 + 2 = 3 + 5$.
solution :

```
∈Π
def Prod(S,n):
    """ Renvoie les x*y pour (x,y)∈N_n tel que x+y = S.
    Autrement dit ce sont les produits possibles connaissant la somme."""
    res=[]
    for x in range(2,n-1):
        y=S-x
        if y > x and x+y <= n : # Comme l'énoncé demande pas d'optimiser, aller au plus
            ↪ simple pour gagner du temps et limiter le risque d'erreurs.
            res.append(x*y)
    return res
# J'aurais quand même envie de dédoublonner pour accélerer la suite... Faire un tri
    ↪ fusion strict par exemple.
```

4. En déduire une fonction `Candidat_S(n)` qui renvoie la liste des entiers S tels que la liste `Prod(S,n)` est incluse dans la liste `CoupleProd(n)`.
solution : `Candidat_S(n)` est donc la liste des sommes compatibles avec l'information que Sophie savait que Pierre ne savait pas.
En particulier, après la première réponse de Sopie, Pierre sait que $\Sigma \in$`Candidat_S(n)`.

```
∈Π∈
# Ça sera plus clair en écrivant une fonction inclus.
# Note : optimisable si tableaux triés.
def inclus(X, Y):
    """ Indique si X ⊂ Y."""
    for x in X:
        if not x in Y:
            return False
    return True

def CandidatS(n):
    """ Renvoie les S possibles, sachant que Sophie savait que Pierre ne connaissait pas
        ↪ (x,y). Càd les S tels que tous les produits P∈Prod(S,n) sont dans
        ↪ coupleProd(n).
    Nous savons à partir de la deuxième ligne du dialogue que S figure parmi ces nombres.
    """
    res=[]
```

```
    P_possibles = coupleProd(n)
    for S in range(5,n):
        if inclus(Prod(S,n), P_possibles):
            res.append(S)
    return res
```

solution : Pierre peut déduire le couple (x, y) : cela implique que dans `Candidat_S(n)`, il n'y a qu'un seul S tel que $\Pi \in$`Prod(S,n)`.

Pour déterminer cet unique S, on recherche tout d'abord les produits P pour lesquels il existe deux sommes S_1 et S_2 dans la liste `Candidat_S(n)` telles que

- $S_1 < S_2$;
- et P apparaît dans les listes `Prod(S_1 ,n)` et `Prod(S_2 ,n)`.

5. Écrire une fonction `Double_P(n)` qui renvoie la liste des produits P satisfaisant cette condition.
solution :

```
∈Π∈⊂∈
# Je commence par une fonction d'intersection
def intersection(X,Y):
    return [ x for x in X if x in Y]
# Encore optimisable si tableaux triés...

# Notons que ma fonction CandidatS renvoie un tableau strictement croissant.
# Version pas du tout optimisée :
def Double_P(n):
    """ Renvoie les produits P pour lesquels il existe au moins deux sommes possibles
        ↪ dans Candidat_S(n).
    À la troisième ligne du dialogue, nous apprenons que ces nombres ne sont pas égaux à
        ↪ Π sans quoi Pierre n'aurait pas pu conclure.
    """
    res=[]
    S_possibles = CandidatS(n)
    nS = len(S_possibles)
    for i in range(nS):
        for j in range(i+1, nS):
            S1, S2 = S_possibles[i], S_possibles[j]
            for P in intersection(Prod(S1,n), Prod(S2,n)):
                res.append(P)
    return res

# Pour optimiser, calculer au préalable des Prod(S,n) nécessaires.
# Et renvoyer un dictionnaire.
```

6. À quelle condition Sophie a-t-elle pu conclure à la dernière ligne du dialogue ? Écrire une fonction `Reste_S(n)` qui renvoie la liste des valeurs possibles pour Σ compte-tenu de cette dernière information.
solution :

Ces produits P de `double_P(n)` ne peuvent être égaux à Π : Pierre n'aurait pas pu conclure. Sophie sait donc que $\Pi \in$ `Prod(`Σ`, n)` $\setminus$ `Double_P(n)`.

Et si Sophie peut conclure c'est que cet ensemble est un singleton.

Ainsi, Σ est tel que `Prod(`Σ`,n)` $\setminus$ `Double_P(n)` est un singleton.

Par conséquent je calcule l'ensemble des S de `Candidats_S(n)` tels que `Prod(S,n)` $\setminus$ `Double_P(n)` est un singleton.

```
∈Π∈⊂∈Π
def Reste_P(S,n):
    """ Renvoie les éléments de Prod(S,n) qui ne sont pas dans Double_P(n).
    Sophie a pu conclure en calculant Reste_P(Σ,n). Donc Reste_P est un singleton.
    """
    P_pas_possibles = Double_P(n)
    return [P for P in Prod(S,n) if not P in P_pas_possibles ]
```

```
ϵΠϵ⊂ϵΠΣ
def Reste_S(n):
    """ Renvoie les S pour lesquels Reste_P(S,n) est un singleton."""
    res=[]
    for S in CandidatS(n):
        if len(Reste_P(S,n)) == 1:
            res.append(S)
    return res
```

7. Écrire une fonction `Solution(n)` qui renvoie les couple (x, y) possibles compte tenu des informations disponibles.
solution :
Nous récupérons les S pour lesquels `Reste_P(S,n)` est un singleton, nous prenons les P correspondant (le contenu du singleton), et nous déduisons x et y.

```
ϵΠϵ⊂ϵΠΣ
def Solution(n):
    res = []
    for S in Reste_S(n):
        for P in Reste_P(S,n):
            # Si la solution est unique, ces boucles s'exécutent une et une seul fois
            # x et y sont racines du polynôme X^2-SX+P
            d=S*S-4*P
            res.append(( (S-d**.5)/2, (S+d**.5)/2 ) ) #Normalement ce sont des entiers
    return res
```

8. Freudenthal avait choisi $n = 100$, et le problème avait alors une unique solution. Écrire une fonction `n_possibles(N)` prenant un entier $N \in \mathbb{N}^*$ et renvoyant les valeurs de $n \in [\![1, N[\![$ pour lesquelles le problème admet une unique solution.

Troisième partie

Bases de données

Nous nous intéressons à une base de données des zoos qui contient deux tables. La première table, `zoos`, a quatre colonnes dont id, un identifiant unique pour chaque zoo. Quelques lignes sont données ci-après.

id	nom	pays	continent
FR42	Zoo de La Flèche	France	Europe
RU12	Parc zoologique de Novossibirsk	Russie	Asie
RU5	Parc zoologique de saint-Pétersbourg	Russie	Asie
	⋮	⋮	⋮

La seconde table, `animaux`, a 6 colonnes, notamment un identifiant unique pour chaque animal (id) et l'identifiant du zoo qui héberge l'animal (zoo).

id	nom	espece	sexe	naissance	zoo
ke860	Kaiko	Chameau	F	2013	FR42
ic431	Jeffrey	Pytho royal	M	2016	RU12
gz599	Antaus	Annaconda vert	M	2016	RU12
⋮	⋮	⋮	⋮	⋮	⋮

1. Écrire une requête SQL renvoyant la table (on veut les colonnes **`id`**, `nom`, `naissance` et `zoo`) des chamelles (chameaux femelles).
solution :

```
SELECT id, nom, naissance, zoo
  FROM animaux
  WHERE espece = "Chameau" AND sexe = "F"
```

2. Écrire une requête SQL renvoyant la table des bonobos mâles vivant en Asie.
solution :

```sql
SELECT *
  FROM zoo JOIN animaux ON zoo.id = animaux.zoo
  WHERE epece="Bonob" ANd sexe="M" AND continent="ASIE"
```

3. Écrire une requête renvoyant la liste des pays ayant des zoos sur plusieurs continents.
solution : Déjà, toutes les informations nécessaires sont dans la table `zoos`.
Je vais faire une auto-jointure, pour obtenir des lignes qui concernent deux zoos d'un même pays mais de deux continents différents.

```sql
SELECT pays
  FROM zoos as z1 JOIN zoos AS z2 ON z1.pays=z2.pays AND z1.continent <> z2.continent
```

Quatrième partie

Graphes et arbres

Dans cet exercice, on considère des graphes non orientés connexes. Les sommets d'un graphe à n sommets ($n \in \mathbb{N}^*$) sont numérotés de 0 à $n-1$. On suppose qu'aucune arête ne boucle sur un même sommet.

Un chemin de longueur $p \in \mathcal{N}$ d'un sommet a vers un sommet b dans un graphe est la donnée de $p+1$ sommets $s0, s1, ..., sp$ tels que $s0 = a$, $sp = b$ et, pour tout$k \in [\![1, p]\!]$, les sommets $sk-1$ et sk sont reliés par une arête.

Un plus court chemin d'un sommet a vers un sommet b dans un graphe G est un chemin de longueur minimale parmi tous les chemins de a vers b. Sa longueur est notée $d_G(a, b)$.

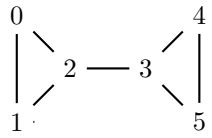
Le diamètre d'un graphe G, noté diam(G), vaut le maximum des longueurs des plus courts chemins entredeux sommets du graphe G. Autrement dit,

$$\text{diam}(G) = \max_{a,b \text{ sommets de } G} d_G(a, b)$$

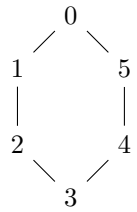
Un chemin maximal d'un graphe G est un plus court chemin de G de longueur diam(G).

1 Exemples de graphes

1. Donner sans justification le diamètre et les chemins maximaux pour chacun des deux graphes G_1 et G_2 de la figure 1.



Graphe G_1



graphe G_2

FIGURE 1 – Exemples de graphes

solution : Ces deux graphes sont de diamètre 3. Voici leur chemins maximaux :

- $(0, 2, 3, 4)$, $(0, 2, 3, 5)$, $(1, 2, 3, 4)$, $(1, 2, 3, 5)$
- $(0, 1, 2, 3)$, $(1, 2, 3, 4)$, $(2, 3, 4, 5)$, $(3, 4, 5, 0)$, $(4, 5, 0, 1)$, $(5, 0, 1, 2, 3)$

et les chemins obtenus en renversant ceux-ci puisque les graphes sont non orientés.

En OCaml, les graphes sont représentés par liste d'adjacence et implémentés par le type

```
type graphe = int list array;;
```

2. Graphes de diamètre maximal.
   (a) Dessiner sans justification un graphe à 5 sommets ayant un diamètre le plus grand possible.

(b) Écrire en OCaml une fonction `diam_max` de type `int -> graphe` qui prend en argument un entier naturel n non nul et qui renvoie un graphe à n sommets de diamètre maximal.
solution :

```
let diam_max n =
  let res = Array.make n [] in
  res.(0)<-[1];
  res.(n-1)<-[n-2];
  for i = 1 to n-2 do
    res.(i)<- [i-1;i+1]
  done;
  res
;;
```

3. Graphes de diamètre minimal.

(a) Dessiner sans justification un graphe à 5 sommets ayant un diamètre le plus petit possible.
solution : Il suffit que tous les sommets soient reliés entre eux : le diamètre est alors 5. Un tel graphe s'appelle une « clique ».

(b) Écrire en OCaml une fonction `diam_min` de type `int -> graphe` qui prend en argument un entier naturel n non nul et qui renvoie un graphe à n sommets de diamètre minimal.
solution :

```
let rec int_sans_i i n=
  (* Renvoie l'intervalle [|0;n|] privé de i *)
  if n= -1 then []
  else if n=i then int_sans_i i (n-1)
  else n :: int_sans_i i (n-1)
;;

(* La méthode de base utilise une boucle for. Si on veut gagner du temps, il existe
    ↪ une fonction Array.init qui prend un entier n et une fonction f, et qui
    ↪ renvoie le tableau de n cases tel que la case i contient f(i). *)
let diam_min n=
  Array.init n (fun i->int_sans_i i (n-1))
;;
```

2 Algorithmes de calcul du diamètre

Dans cette partie, on suppose encore que les graphes sont représentés par listes d'adjacence.

1. Donner l'entrée et la sortie de l'algorithme de Dijkstra. Comment cet algorithme permet-il de calculer le diamètre d'un graphe ?
solution :
   - Entrées : un graphe G, et deux sommets d et a d'icelui.
   - Sortie : un plus court chemin de d à a, ou la distance entre ces deux sommets.

   Il suffit d'utiliser cet algorithme sur tous les couples de sommets de G et de prendre le maximum des distances obtenues pour obtenir le diamètre.
2. Quel parcours de graphe peut être utilisé pour le calcul du diamètre ?
solution : Un parcours en largeur permet de calculer la distance entre deux sommets dans un graphe non pondéré. Il peut être utilisé de la même manière que l'algorithme de Dijkstra.
3. Laquelle des deux méthodes précédentes est la mieux adaptée pour calculer le diamètre d'un graphe ?
solution : Un parcours en largeur est moins complexe. En effet, la structure du programme est semblable, mais il utilise une file d'attente au lieu d'une file de priorité (tas). Or les opérations de base sur une file d'attente ont une complexité (amortie) en $O(1)$, alors que celles sur les tas sont en $O(\log n)$ où n est le nombre d'éléments dans le tas, nombre qui pourra aller jusqu'à $|A|$, A étant l'ensemble des arêtes de G.

   En outre un parcours en largeur est plus simple à programmer.

3 Diamètre d'un arbre binaire

Dans cette partie, on s'intéresse aux arbres binaires, qui sont des cas particuliers de graphes. On travaille avec une représentation spécifique de ces graphes particuliers, implémentée en OCaml par le type suivant :

```
type arbre = Feuille | Noeud of int * arbre * arbre ;;
```

Le *graphe sous-jacent* G_A à un arbre binaire A est défini comme le graphe orienté dont

- les sommets correspondent aux nœuds de l'arbre (et pas aux feuilles) ;
- les arêtes correspondent aux branches de l'arbre reliant deux nœuds (et non pas celles reliant un nœud à une feuille).

Le diamètre d'un arbre binaire est alors défini comme le diamètre de son graphe sous-jacent.
La figure 2 présente un exemple d'arbre binaire A (à gauche) et de son graphe sous-jacent G_A.

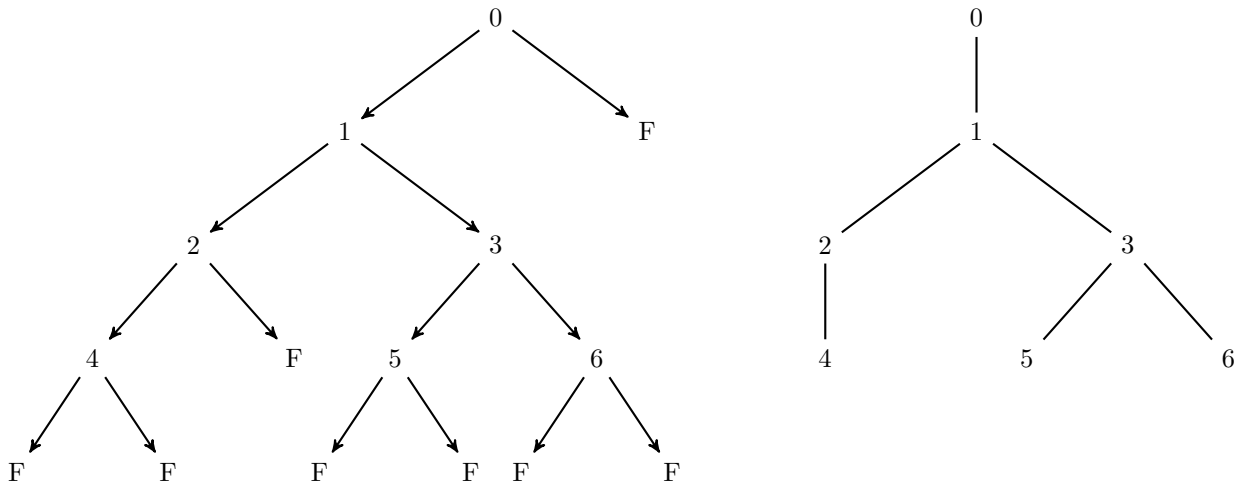


FIGURE 2 – Exemple d'arbre

1. Donner l'expression OCaml représentant l'arbre A de l'exemple. Donner le diamètre de A et les chemins maximaux du graphe sous-jacent G_A.
   solution :

```
let exemple =
  Noeud(0,
        Noeud(1,
              Noeud(2,
                    Noeud(4, Feuille, Feuille),
                    Feuille
                ),
              Noeud(3,
                    Noeud(5, Feuille, Feuille),
                    Noeud(6,Feuille, Feuille)
                )
          )
        ,
          Feuille
    );;
```

   Le diamètre de A est 4, et ses chemins maximaux sont $(4, 2, 1, 3, 5)$, $(4, 2, 1, 3, 6)$ et leur renversés.

2. Quel est le nombre r d'arêtes du graphe sous-jacent à un arbre binaire possédant n nœuds ?
   solution : Un arbre binaire de n nœuds a $n - 1$ arêtes.

Une première approche pour calculer le diamètre d'un arbre consiste à le transformer en un graphe et à employer un algorithme général sur les graphes de la partie 2.

3. Écrire en OCaml une fonction `nb_noeuds` de type `arbre -> int` qui renvoie le nombre de nœuds d'un arbre binaire donné en argument.
solution :

```
let rec nb_noeuds = function
  |Feuille -> 0
  |Noeud(_, fg, fd) -> 1 + nb_noeuds fg + nb_noeuds fd
;;
```

4. Écrire en OCaml une fonction `numerotation` de type `arbre -> (arbre*int)` qui prend en argument un arbre binaire A à n nœuds et qui renvoie le couple formé d'un arbre binaire A' de même graphe sous-jacent que A et dont les nœuds sont étiquetés de 0 à $n-1$, et du nombre n de nœuds de a.
solution :

```
(* J'utilise une fonction aux qui prend en entrée le prochain numéro à utiliser, et qui
    ↪ renvoie le prochain numéro disponible.*)
let numeroted a =
  let rec aux i = function
      (* i : prochain numéro à utiliser
         Renvoie le couple (arbre numéroté, prochain numéro dispo)
       *)
    |Feuille -> Feuille, i
    |Noeud(_, fg, fd) ->
      let nfg, j = aux (i+1) fg in
      let nfd, k = aux j fd in
      Noeud(i, nfg, nfd), k
  in
  aux 0 a
;;
```

5. Écrire en OCaml une fonction `arbre_vers_graphe` de type `arbre -> graphe` qui prend en argument un arbre binaire A à n nœuds étiquetés de 0 à $n-1$ et qui renvoie le graphe G_A sous-jacent à A (le type graphe est défini dans la partie 1).
solution :

```
let arbre_vers_graphe a=
  let an, n = numeroted a in
  let res = Array.make n [] in

  let ajoute_arete i j =
    (* Ajoute une arête non orientée entre i et j. *)
    res.(i) <- j::res.(i);
    res.(j) <- i::res.(j)
  in

  let rec parcours_arbre pere = function
      (*
        Entrées : un arbre a
                  pere : étiquette du père de l'arbre passé en arg.
        Effet : rajoute dans res les arêtes de a
       *)
    |Feuille -> ()
    |Noeud(e, fg, fd) ->
      ajoute_arete pere e;
      parcours_arbre e fg;
      parcours_arbre e fd
  in
  begin
  match a with
  |Feuille ->()
  |Noeud(e, fg, fd) -> parcours_arbre e fg; parcours_arbre e fd
  end;

  res
```

```
;;
```

6. Décrire un algorithme qui calcule le diamètre d'un arbre de type arbre en se ramenant à un graphe. Quelle est sa complexité ?
   solution : L'algorithme suggéré par les question précédentes consiste à transformer l'arbre en graphe, puis calculer son diamètre par des parcours en largeur. Détaillons ses étapes et sa complexité. Soit a un arbre, et n son nombre de nœuds.
   (a) On numérote les étiquettes au moyen de `numeroted` : $O(n)$.
   (b) Création du graphe par `arbse_vers_graphe` : $O(n)$.
   (c) Pour un graphe (S, A), un parcours en largeur est un $O\left(|S| + |A|\right)$. Ici, le nombre d'arêtes et de sommets sont en $O(n)$, donc un parcours en largeur est en $O(n)$.
   Or nous devons lancer un parcours en largeur pour tout couple de sommets. La complexité est alors en $O(n^3)$.
   Total : $O(n^3)$.

Une seconde approche pour calculer le diamètre d'un arbre consiste à employer une technique diviser-pour-régner. Pour tout arbre A non réduit à une feuille, de la forme `Noeud (x, arbre_g, arbre_d)`, on note

- A_g le fils gauche de A, représenté par `arbre_g` ;
- A_d le fils droit de A, représenté par `arbre_d`.

La hauteur de l'arbre A, notée $h(A)$, est la longueur du plus long chemin descendant de la racine vers une feuille. Dans l'arbre exemple de la partie 3, l'arbre A est de hauteur 4.

7. Soit A un arbre binaire. Proposer une formule donnant la longueur d'un chemin maximal passant par la racine. Justifier.
   solution : Un chemin passant par la racine de A doit passer par chacun des deux fils A_g et A_d(ou terminer ou commencer par la racine, dans le cas où un fils est une feuille). En effet, sinon il y aurait un aller-retour et le chemin ne serait pas un plus court chemin.
   Or le plus long chemin, dans le graphe sous-jacent, partant de la racine de A_g a pour longueur $h(A_g)$, et le plus long chemin partant de la racine de A_d a pour hauteur $h(A_d)$.
   Ainsi, le plus long chemin passant par la racine de A a pour longueur $h(A_g) + h(A_d) + 2$. Le « +2 » vient des deux arêtes utilisées pour atteindre puis quitter la racine.
   Enfin, on constate que cette formule vaut encore si A_g ou A_d est une feuille.

8. Écrire en OCaml une fonction `diam_arbre` de type `arbre -> int` qui calcule le diamètre d'un arbre donné en argument. Cette fonction devra être de complexité linéaire en le nombre de nœuds de l'arbre.
   solution : Soit A un arbre non réduit à une feuille, et c un chemin maximal dans A. Il y a deux cas :
   - c passe par la racine de A, il est alors de longueur $h(A_g) + h(A_d) + 2$;
   - ou il est inclus dans A_g ou dans A_d, et sa longueur est obtenue par un appel récursif.

   Enfin, pour obtenir une complexité optimale, il convient d'écrire une fonction qui renvoie le diamètre le l'arbre mais aussi sa hauteur.

```
let max3 a b c = max (max a b) c;;
let diam_arbre a =

  let rec aux = function
      (*
        Entrée : un arbre a
        Sortie : (diamètre de a, hauteur de a)
       *)
    |Feuille -> 0, -1
    |Noeud(_, fg, fd) ->
      let dfg, hfg = aux fg
      and dfd, hfd = aux fd in
      max3
        (hfd + hfg +2)
        dfg
        dfd,
      (1+max hfg hfd)
  in
  let res, _ = aux a in
  res
;;
```