Première épreuve d'admissibilité - Problème n° 1
Éléments de correction

Ce document donne des éléments de correction pour la première épreuve d'admissibilité. Les algorithmes et programmes suggérés dans ce document constituent une proposition mais d'autres propositions peuvent bien-entendu être données par les candidats.

Ce problème s'intéresse au processus de séquençage de l'ADN. Dans ce contexte, l'une des étapes importantes du séquençage permet de déterminer quelles sont les parties de la séquence d'ADN recherchée mais dans un ordre indéterminé. Le problème consiste alors à **reconstruire** la séquence complète à partir de ces parties. La suite de ce sujet propose une résolution algorithmique à ce problème.

Pour cela, nous représentons **une séquence** S_n d'ADN de longueur n par une suite de n lettres, chaque lettre appartenant à l'ensemble $\{A, T, G, C\}$.
Exemple : $S_9 = AGGTCAGGT$ est une séquence d'ADN de 9 lettres.

Recherche de mots dans une séquence d'ADN

Dans un premier temps nous allons chercher à déterminer quelles sont les parties de longueur fixe d'une séquence, appelées **mots de la séquence** par la suite.
Formellement, soient S_n une séquence d'ADN et l un entier tel que $0 < l \leq n$. On appelle *mot de S_n* de longueur l toute suite de l lettres contiguës contenue dans la séquence S_n.
Exemples : GTC est un mot de longueur 3 de la séquence $S_9 = AGGTCAGGT$, tandis que ATT n'en est pas un.

1. **Combien de mots de longueur l existe-t-il dans une séquence de longueur n (en comptant les répétitions possibles) ?**

   Il y a $n - l + 1$ mots de longueur l dans une séquence de longueur n.
   Par exemple dans `ABCDEFGH` une séquence de longueur 8 il y a 6 mots de longueur 3 : {`ABC`, `BCD`, `CDE`, `DEF`, `EFG`, `FGH`}

2. **Proposer un algorithme qui, étant donnés un entier n, une séquence d'ADN S_n de longueur n et un entier strictement positif l ($l \leq n$), calcule la liste de tous les mots de longueur l de S_n.**

   Une telle liste peut contenir plusieurs fois le même mot. Vous supposerez que la séquence d'ADN est contenue dans un tableau de caractères. Il n'est pas demandé de décrire les manipulations de listes que vous aurez éventuellement besoin d'effectuer, comme l'ajout d'un élément dans une liste par exemple.
   Exemple : l'algorithme appelé sur la séquence $S_9 = = AGGTCAGGT$ et $l = 3$ doit calculer la liste [AGG,GGT,GTC,TCA,CAG,AGG,GGT].

Voici un algorithme pour donner la liste de tous les mots de longueur l d'une séquence d'ADN de longueur n avec $n \geq l$: la séquence d'ADN est stockée dans un tableau nommé `S_n` indicé de 0 à $n-1$; `liste_mot` est la liste de tous les mots de longueur l que nous voulons construire.

```
liste_mot <- liste vide
mot <- tableau de l caractères vide
Pour i_seq de 0 à n-l faire
     Pour i_mot de 0 à l-1 faire
         mot[i_mot] = S_n[i_seq + i_mot]
     Fin Pour
     liste_mot = liste_mot U mot.
Fin Pour
renvoyer liste_mot
```

3. **Quelle est la complexité de votre algorithme en nombre d'opérations, en fonction de n et de l ?**

   Le nombre d'itérations de la boucle interne est de l. Le nombre d'itérations de la boucle externe est de $n-l+1$. La complexité en nombre d'opérations de l'algorithme est de $(n-l+1) \times l$.
   La réponse $O((n-l) \times l))$ est aussi acceptée.

   On s'intéresse à la question de déterminer tous les mots distincts de longueur l dans S_n. Nous cherchons ici à proposer une fonction Python. Une séquence S_n sera alors décrite comme une chaîne de caractères de longueur n.
   Exemple : la liste de tous les mots distincts de longueur 3 *dans la séquence* S_9 = *"AGGTCAGGT" où* $n = 9$ *et* $l = 3$ *est ["AGG","GGT","GTC","TCA","CAG"].*

4. **Soient n et l deux entiers ($n \geq l > 0$). Combien de mots distincts de longueur l existe-t-il au plus dans une séquence de longueur n ?**

   Le nombre de mots de longueur l possibles sur un alphabet à 4 lettres est 4^l. Le nombre maximum de mots de longueur l dans une séquence de longueur n est $n-l+1$ comme vu dans la première question. Donc le nombre maximum de mots distincts de longueur l dans une séquence de longueur n est $\min\{n-l+1, 4^l\}$.

5. **Proposer une structure de données en Python adaptée pour stocker les mots de la séquence sans répétition.**

   Il s'agit de stocker un ensemble non ordonné de mots sans duplication. La structure adaptée en Python est la structure `set`. La structure `liste` ne gère pas les duplications, il faut donc que ce soit le programme qui gère les duplications si la structure `liste` est utilisée.

6. **Proposer une fonction `liste_mots` en Python qui prend en arguments une séquence S_n, un entier n et un entier l ($n \geq l > 0$), et qui renvoie la liste de tous les mots distincts de longueur l de S_n.**

   Fonction `liste_mots` gérant manuellement les duplications :

```
def liste_mots(Sn,n,l):
    mots = []
    for i in range(n-l+1):
```

```
            if Sn[i:i+l] not in mots:
                mots.append(Sn[i:i+l])
        return mots
```

Fonction `liste_mots` utilisant un set :

```
def liste_mots(Sn,n,l):
    ensemble = set()
    for i in range(0,n-l+1):
        ensemble.add(tuple(Sn[i:i+l])) // Les items d'un set doivent être immutables.
    return ensemble
```

Reconstruction d'une séquence d'ADN

L'une des difficultés lors du séquençage de l'ADN est qu'il faut reconstruire la séquence complète à partir de l'ensemble des mots de longueur l qui la composent, sans qu'aucune information sur leur position dans la séquence ne soit donnée.
Le reste de cet exercice va se concentrer précisément sur la résolution algorithmique de ce problème.

Soient S_n une séquence d'ADN de longueur n et l un entier tel que $0 < l \leq n$. On appelle spectre($\mathrm{S_n}, \mathrm{l}$) l'ensemble des mots de longueur l distincts qui sont dans la séquence S_n.
Exemple : étant donnée la séquence $\mathcal{S}_9 = AGGTCAGGT$, *spectre(*$S_9$*,3)* $= \{AGG, CAG, GGT, GTC, TCA\}$.
Le problème de la **reconstruction** de la séquence d'ADN consiste, à partir de la connaissance d'un spectre $\mathcal{SP}$ contenant des mots de longueur l, à déterminer une séquence d'ADN S compatible avec $\mathcal{SP}$, c'est-à-dire telle que spectre$(S, l) = \mathcal{SP}$. Plus formellement, on peut formuler le problème **Reconstruction** de la manière suivante :
Données : un entier l ; un ensemble $\mathcal{SP}$ de mots de longueur l.
Sortie : une séquence d'ADN S telle que spectre$(S, l) = \mathcal{SP}$.

Pour résoudre ce problème, nous allons nous intéresser aux recouvrements existant entre les mots d'un spectre.

Soient M un mot de longueur $n > 0$ et k un entier tel que $0 < k \leq n$, on appelle *préfixe* de M de longueur k le mot constitué des k premières lettres de M. De manière similaire, on appelle *suffixe* de M de longueur k le mot constitué des k dernières lettres de M.

Soient deux mots M_1 et M_2 de longueur n_1 et n_2 respectivement et soit $k \leq \min(n_1, n_2)$ un entier strictement positif. Il existe un *recouvrement de longueur* k entre M_1 et M_2 si et seulement si le suffixe de M_1 de longueur k est identique au préfixe de M_2 de longueur k. Nous appellerons également *longueur du recouvrement maximal* entre M_1 et M_2, noté lrmax($\mathrm{M_1}, \mathrm{M_2}$) dans la suite, le plus grand entier k tel qu'il existe un recouvrement de longueur k entre M_1 et M_2. Si aucun recouvrement n'existe entre M_1 et M_2, lrmax$(M_1, M_2) = 0$
Exemple : lrmax$(ACGG, CTAG) = 0$ *et* lrmax$(ACGG, GGAT) = 2$.

7. **Que valent :**
   - lrmax$(ATGC, GGTA)$?
   - lrmax$(TGGCGT, CGTAAATG)$?
   - lrmax$(GCTAGGCTAA, AGGCTAAGTCGAT)$?
   - lrmax$(TCTAGCCAGCTAGC, TAGCCAGCTAGCACT)$?

- lrmax$(ATGC, GGTA) = 0$
- lrmax$(TGGCGT, CGTAAATG) = 3$ car CGT est un suffixe de $TGGCGT$ et un préfixe de $CGTAAATG$
- lrmax$(GCTAGGCTAA, AGGCTAAGTCGAT) = 7$ car $AGGCTAA$ est un suffixe de $GCTAGGCTAA$ et un préfixe de $AGGCTAAGTCGAT$.
- lrmax$(TCTAGCCAGCTAGC, TAGCCAGCTAGCACT) = 12$ car $TAGCCAGCTAGC$ est un suffixe de $TCTAGCCAGCTAGC$ et un préfixe de $TAGCCAGCTAGCACT$

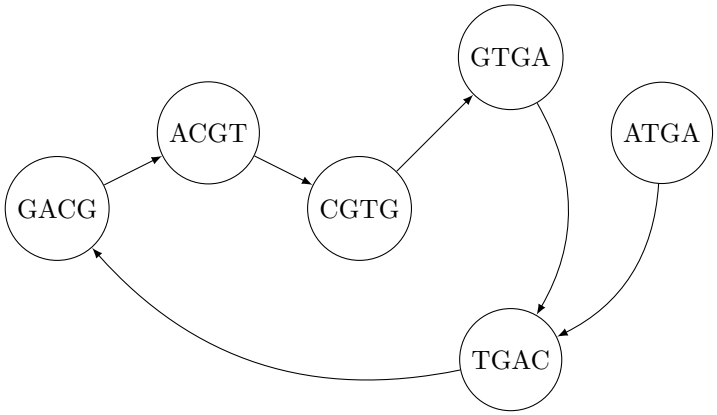
Première modélisation

Soient $l \geq 2$ un entier et $\mathcal{SP}$ un spectre contenant des mots de longueur l. Nous allons modéliser notre problème par un graphe orienté $G = (V, E)$, dans lequel :

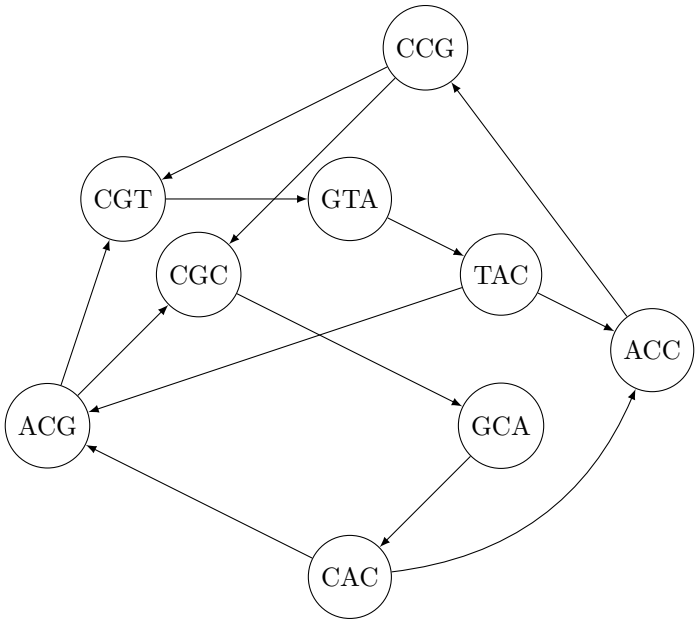
- Chaque sommet de V correspond à un mot du spectre et chaque mot du spectre est représenté par un et un seul sommet.
- L'ensemble E contient un arc entre $v_1 \in V$ et $v_2 \in V$ si et seulement si lrmax$(v_1, v_2) = l - 1$.

8. **Quelle est la modélisation obtenue sous forme de graphe pour les spectres et longueurs suivants :**
   - $\{GTGA, ATGA, GACG, CGTG, ACGT, TGAC\}$ et $l = 4$?
   - $\{TAC, ACC, ACG, CAC, CCG, CGT, CGC, GCA, GTA\}$ et $l = 3$?

   Voici la modélisation obtenue pour $\{GTGA, ATGA, GACG, CGTG, ACGT, TGAC\}$ et $l = 4$:



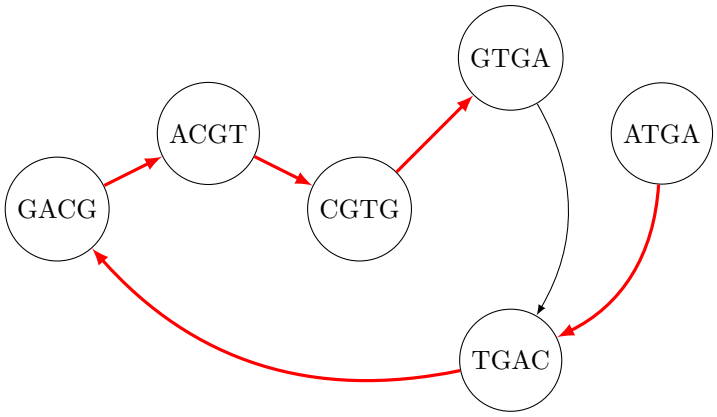
   Voici la modélisation obtenue pour $\{TAC, ACC, ACG, CAC, CCG, CGT, CGC, GCA, GTA\}$ et $l = 3$:



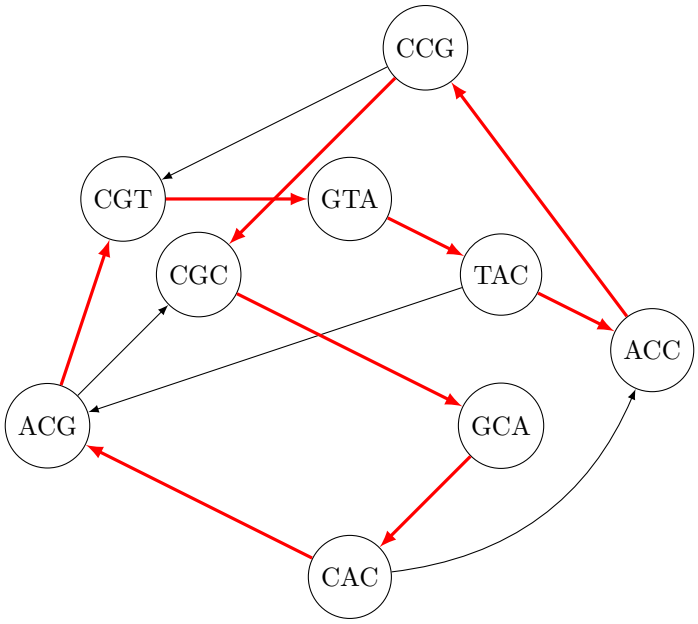
9. **Proposer une formulation du problème de reconstruction comme un parcours de graphe. De quel problème classique de théorie des graphes ce problème se rapproche-t-il ?**

   Tous les mots du spectre doivent apparaître dans la séquence reconstruite. Il faut donc parcourir tous les sommets du graphe au moins une fois. Parcourir une et une seule fois tous les sommets du graphe pour obtenir la séquence reconstruite la plus courte possible correspond au problème de recherche d'un chemin hamiltonien.

10. **Proposer pour chacun des graphes obtenus dans la question 8 une solution au problème Reconstruction.**



   Le chemin hamiltonien considéré (en rouge) dans la figure ci-dessus est composé des nœuds suivants (et dans cet ordre) : $ATGA, TGAC, GACG, ACGT, CGTG, GTGA$. Ce qui donne la séquence suivante : $ATGACGTGA$. Comme le sommet $ATGA$ est un sommet sans arc entrant, il est forcément le début d'un chemin et comme tous les nœuds ensuite n'ont qu'un seul arc sortant, il n'y a qu'un chemin possible.



Le circuit (en rouge) dans la figure ci-dessus est composé des nœuds suivants (et dans cet ordre) : $ACG, CGT, GTA, TAC, ACC, CCG, CGC, GCA, CAC, ACG$.

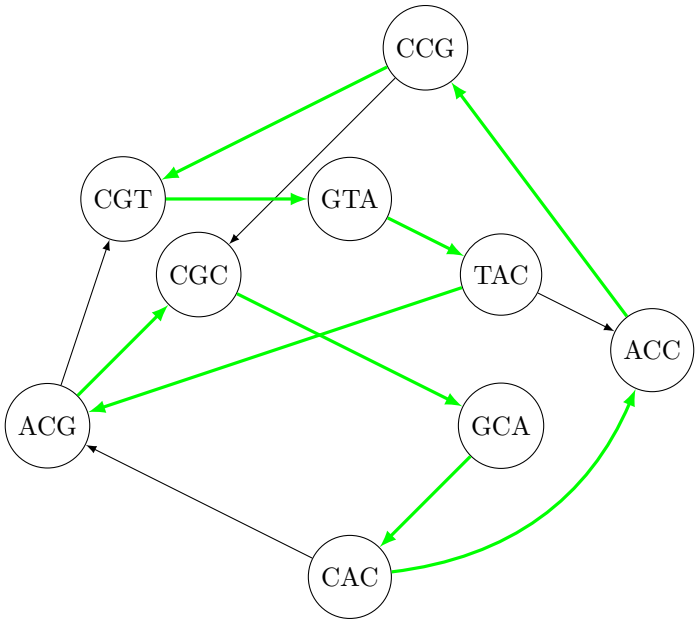
Dans ce circuit, il existe plusieurs chemins, ce qui donne différentes réponses possibles :

— Par exemple le chemin composé des nœuds suivants $ACG, CGT, GTA, TAC, ACC, CCG, CGC, GCA, CAC$ donne la séquence suivante : $ACGTACCGCAC$.

— On pourrait commencer par le nœud CAC et terminer par le nœud GCA, ce qui donnerait le parcours suivant : $CAC, CGT, GTA, TAC, ACC, CCG, CGC, GCA$, et la séquence suivante : $CACGTACCGCA$.

— On pourrait commencer par le nœud GCA et terminer par le nœud CGC, ce qui donnerait le parcours suivant : $GCA, CAC, CGT, GTA, TAC, ACC, CCG, CGC$, et la séquence suivante : $GCACGTACCGC$.

Dans la figure suivante il existe aussi d'autres circuits (qui permettent d'avoir d'autres chemins), comme par exemple le circuit en vert dans la figure ci-dessous :



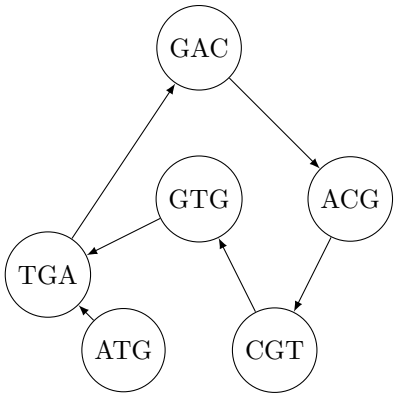
Deuxième modélisation

Il est possible de modéliser autrement ce problème, toujours sous forme de graphe. Nous modélisons maintenant les données ($l \geq 2$ un entier et $\mathcal{SP}$ un spectre contenant des mots de longueur l) par un graphe orienté $G = (V, E)$ où :

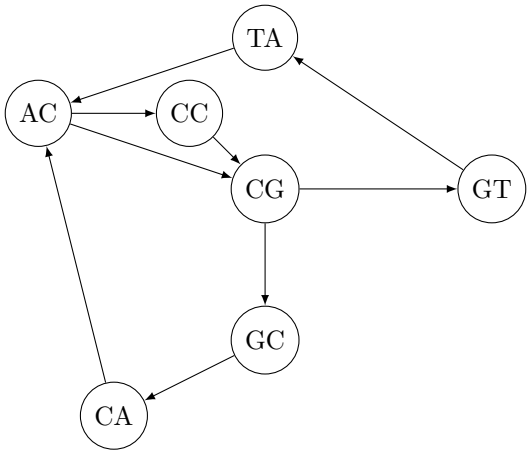
- V est l'ensemble de tous les préfixes de longueur $l-1$ et de tous les suffixes de longueur $l-1$ des mots du spectre.
- L'ensemble E contient un arc entre les sommets $v_1 \in V$ et $v_2 \in V$ si et seulement si le spectre $\mathcal{SP}$ contient un mot M de longueur l tel que v_1 est le préfixe de M de longueur $l-1$ et v_2 est le suffixe de M de longueur $l-1$.

11. **Quelle est la modélisation obtenue sous forme de graphe pour les spectres suivants :**
    - $\{GTGA, ATGA, GACG, CGTG, ACGT, TGAC\}$ et $l = 4$?
    - $\{TAC, ACC, ACG, CAC, CCG, CGT, CGC, GCA, GTA\}$ et $l = 3$?

    Pour le spectre $\{GTGA, ATGA, GACG, CGTG, ACGT, TGAC\}$:
    — La liste des préfixes est : $GTG, ATG, GAC, CGT, ACG, TGA$
    — La liste des suffixes est : TGA, ACG, GTG, CGT, GAC
    — La liste des sommets du graphe est donc : $ATG, ACG, CGT, GTG, GAC, TGA$.
    Voici le graphe obtenu :



Pour le spectre $\{TAC, ACC, ACG, CAC, CCG, CGT, CGC, GCA, GTA\}$:
— La liste des préfixes est : $TA, AC, CA, CC, CG, GC, GT$
— La liste des suffixes est : $AC, CC, CG, GT, GC, CA, TA$
— La liste des sommets du graphe est donc : $AC, CC, CG, GT, GC, CA, TA$.



On rappelle que dans un graphe orienté $G = (V, E)$, un *chemin* d'origine u ($\in V$) et d'extrémité v ($\in V$) est défini par une suite d'arcs consécutifs reliant u à v. Un *circuit* est un chemin dont les deux extrémités sont identiques. Un chemin (resp. circuit) est dit *eulérien* s'il contient une fois et une seule chaque arc de G.

12. **Les graphes construits à l'aide de la modélisation de la question 11 contiennent-ils un circuit eulérien ? un chemin eulérien ?**

Pour le premier graphe, il contient un chemin eulérien : $ATG, TGA, GAC, ACG, CGT, GTG, TGA$, mais il ne contient pas de circuit eulérien puisqu'il n'y a aucun arc entrant dans le sommet ATG. Le sommet ATG est forcément le sommet initial du chemin puisqu'il a un seul arc sortant.

Pour le second graphe, il contient plusieurs circuits eulériens, comme par exemple :
— $AC, CC, CG, GT, TA, AC, CG, GC, CA, AC$.
— $AC, CG, GT, TA, AC, CC, CG, GC, CA, AC$.
— $AC, CG, GC, CA, AC, CC, CG, GT, TA, AC$.
— $AC, CC, CG, GC, CA, AC, CG, GT, TA, AC$.

13. **En quoi un circuit ou un chemin eulérien dans un graphe construit avec cette modélisation constitue-t-il une solution au problème de la reconstruction de séquence d'ADN ?**

Dans cette modélisation les arcs représentent les mots du spectre. Il faut qu'une solution contienne au moins une fois chaque mot du spectre, il faut donc passer au moins une fois par chaque arc du graphe. Si l'on veut la solution la plus courte possible, il faut passer une et une seule fois par chaque arc du graphe.

Dans la suite de l'exercice, nous nous restreignons au circuit eulérien qui est plus simple à déterminer qu'un chemin eulérien.
Soit $(u, v) \in E$. On dit que l'arc (u, v) est un arc entrant du sommet v et un arc sortant du sommet u.
On définit le degré entrant d'un sommet v, noté $d_e(v)$, comme le nombre d'arcs entrants du sommet v.
On définit le degré sortant d'un sommet u, noté $d_s(u)$, comme le nombre d'arcs sortants du sommet u.

14. **Montrer qu'un graphe orienté contient un circuit eulérien si et seulement si $\forall v \in V, d_e(v) = d_s(v)$.**

Le graphe construit par modélisation est un graphe connexe.
Il est facile de vérifier que si un graphe est eulérien alors les sommets ont un degré entrant égal au degré sortant. En effet, à chaque fois qu'on arrive sur un sommet par un arc entrant, on doit ressortir de ce sommet par un arc sortant puisqu'on parcourt tous les arcs, sauf pour le sommet sur lequel on arrive à la fin du parcours. Ce dernier correspond au sommet de départ puisqu'on a un circuit. Donc pour chaque sommet $v \in V$ on a $d_e(v) = d_s(v)$.
Nous allons montrer que si pour chaque sommet $v \in V, d_e(v) = d_s(v)$ alors le graphe est eulérien. Pour construire un circuit eulérien, on part d'un sommet u quelconque et on parcourt un chemin quelconque en passant par des arcs qui n'ont pas encore été utilisés. On arrête le chemin quand on arrive sur un sommet qui n'a pas d'arc de sortie possible, c'est-à-dire un sommet pour lequel tous les arcs sortants ont déjà été empruntés. Dans un graphe où tous les sommets respectent la règle $\forall v \in V, d_e(v) = d_s(v)$, le seul sommet pour lequel cette situation peut arriver est le sommet de départ, u. Le chemin construit commence et termine dans le même sommet. Nous avons donc un circuit. Si le circuit est eulérien, alors c'est terminé. Sinon, le circuit construit contient un sommet w qui a des arc adjacents non encore visités. Dans le graphe initial, pour chaque sommet il y a autant d'arcs sortants que d'arcs entrants. Dans le circuit construit, cette condition est aussi vérifiée. Donc, dans les arcs non encore visités, cette condition est aussi vraie. Il doit donc exister un chemin commençant et terminant en w contenant tous les arcs non encore visités. On peut combiner les 2 circuits ainsi construits, on prend le premier chemin de u à w issu du 1er circuit, puis le second chemin de w à w et enfin le premier chemin de w à u issu du 1er circuit. Puisque le graphe est connexe, on peut répéter cette dernière étape autant de fois qu'il reste des arcs non encore visités jusqu'à obtenir un circuit eulérien. Étant donné qu'à chaque fois que l'on effectue cette étape, on retire des arcs non visités, ce processus termine nécessairement.

Dans la suite de l'exercice, un graphe sera représenté en Python par un dictionnaire dont les clefs sont des chaînes de caractères (`str`) représentant les sommets du graphe et les valeurs sont des listes de chaînes de caractères (`list`) contenant les sommets voisins de la clé.

15. **Écrire une fonction `presence_circuit_eulerien` en Python qui prend en arguments un graphe orienté G et qui détermine si G a un circuit eulérien.**

Voici une fonction `presence_circuit_eulerien` possible (des algorithmes plus efficaces que celui proposé ici sont possibles) :

```
def presence_circuit_eulerien (G) :
        presence = True // booleen stockant la présence d'un circuit eulérien
        for i in G.keys() :            // Pour chaque sommet
// la longueur de la liste d'adjacence donne le degré sortant
                deg_sortant = len(G[i])
// pour le calcul du degré entrant
                deg_entrant = 0
// on parcourt toutes les listes d'adjacence
                for j in G.values() :
// Si le sommet est présent alors on augmente le degré entrant de 1
                      if i in j :
                             deg_entrant = deg_entrant + 1
                if deg_sortant !=  deg_entrant:
                      presence = False

        return presence
```

Nous supposons que les graphes considérés dans les questions 16, 17, 18, 19 et 20 comprennent un circuit eulérien.

16. **Écrire une fonction `construction_circuit` en Python qui prend en arguments un graphe orienté G et un sommet v quelconque de G et qui renvoie un circuit ayant pour origine le sommet v.**

Fonction `construction_circuit` :

```
def construction_circuit(G,v):
// On initialise le marquage de tous les sommets à Faux
       marquage = {}
       for i in G.keys() :
              marquage[i] = [False] * len(G[i])
       continuer = True
// On part du sommet v et on initialise le circuit  avec le sommet v
       sommet = v
       circuit = [v]
// Tant que l'on peut rajouter des sommets
       while continuer :
              continuer = False
// On parcourt tous les voisins du sommet courant
              for (ieme_voisin, marque) in enumerate(marquage[sommet]) :
// Si le i_eme voisin n'est pas marqué. On l'ajoute  dans le circuit,
 on marque le sommet et il devient le nouveau sommet courant.
                     if not marque:
                            prochain_sommet = G[sommet][ieme_voisin]
                            circuit.append(prochain_sommet)
                            marquage[sommet][ieme_voisin] = True
                            sommet = prochain_sommet
                            continuer = True
                            break
       return circuit
```

17. **Écrire une fonction `enleve_circuit` en Python qui prend en arguments un graphe orienté G et un circuit C de G et qui supprime de G les arcs appartenant au circuit C.**

Fonction `enleve_circuit` :

```
def enleve_circuit(G,C):
// On part du premier stocké dans le circuit
       init = C[0]
       sommet = 1
       while sommet < len(C):
// On enlève l'arête entre le sommet init et  le sommet sommet dans le graphe
              G[init].remove(C[sommet])
// On passe aux sommets suivants sur le circuit
              init = C[sommet]
              sommet = sommet + 1
```

18. **Écrire une fonction `sommet_commun` en Python qui prend en arguments un graphe orienté G et un circuit C (qui n'est pas inclus dans G) et qui renvoie un sommet commun de degré non nul appartenant à la fois à G et à C.** Nous supposerons que G et C ont au moins un tel sommet en commun.

Fonction `sommet_commun` :

```
def sommet_commun(G,C):
       i = 0
 // Pour chaque sommet du circuit
      while i < len(C):
              init = C[i]
// Si le sommet a un degré non nul dans le graphe  alors il remplit les conditions
              if len(G[init]) != 0 :
                     return init
              i = i + 1
```

19. **Écrire une fonction `fusion_circuits` en Python qui prend en arguments deux circuits $C1$ et $C2$ et v un sommet commun à $C1$ et $C2$ et qui renvoie un circuit composé des arcs de $C1$ et des arcs de $C2$.**

Fonction `fusion_circuit` :

```
def fusion_circuits(C1,C2,v):
   indice1,indice2 = C1.index(v),C2.index(v)
    return C1[:indice1] + C2[indice2:-1] + C2[:indice2+1] + C1[indice1+1:]
```

20. **À partir des fonctions Python que vous avez proposées aux questions 16, 17, 18 et 19, écrire une fonction `circuit_eulerien` en Python qui prend un entrée un graphe orienté G et qui renvoie un circuit eulérien de G.**

Fonction `circuit_eulerien` :

```
def circuit_eulerien(G):
    m=0                    // nombre d'arcs dans G
    for i in G:
        m = m + len(G[i])
```

```
sommet = list(G)[0]   // choix arbitraire d'un sommet de G
circuit = construction_circuit(G,sommet)
enleve_circuit(G,circuit)

while len(circuit) - 1 < m:   // tant qu'on n'a pas capturé toutes les arcs de G
    sommet = sommet_commun(G,circuit)
    C = construction_circuit(G,sommet)
    enleve_circuit(G,C)
    circuit = fusion_circuits(circuit,C,sommet)

return circuit
```

21. **Quelle est la séquence d'ADN déterminée par la fonction proposée à la question 20 sur le graphe obtenu à la question 11 avec le spectre $\mathcal{SP} = \{TAC, ACC, ACG, CAC, CCG, CGT, CGC, GCA, GTA\}$ et $l = 3$?**

22. **Proposer une autre séquence d'ADN compatible avec le spectre $\mathcal{SP} = \{TAC, ACC, ACG, CAC, CCG, CGT, CGC, GCA, GTA\}$ et différente de la séquence déterminée à la question 21.**

Pour ces deux questions, en fonction des circuits eulériens, il peut y avoir différentes séquences ADN trouvées, à permutation circulaire près :
— $AC, CC, CG, GT, TA, AC, CG, GC, CA, AC$, la séquence peut être $ACCGTACGCAC$
— $AC, CG, GT, TA, AC, CC, CG, GC, CA, AC$, la séquence peut être $ACGTACCGCAC$
— $AC, CG, GC, CA, AC, CC, CG, GT, TA, AC$, la séquence peut être $ACGCACCGTAC$
— $AC, CC, CG, GC, CA, AC, CG, GT, TA, AC$, la séquence peut être $ACCGCACGTAC$

Première épreuve d'admissibilité - Problème n° 2
Éléments de correction

Réponse : Les requêtes SQL suggérées dans ce document constituent une proposition mais d'autres réponses peuvent être apportées.

Préambule : les différentes parties de ce problème peuvent être traitées indépendamment les unes des autres, bien que les définitions et notations données au fur et à mesure du sujet soient communes à toutes les parties concernées.

Nous nous intéressons dans ce problème aux systèmes de gestion de base de données (SGBD), c'est-à-dire aux logiciels qui permettent de stocker et manipuler des informations contenues dans les bases de données. Les SGBDs implémentent différents mécanismes pour assurer les bonnes propriétés des bases de données, notamment la persistance des données en cas de panne, l'atomicité et la cohérence des transactions ainsi que la protection des données en fonction des droits des utilisateurs.

1 Interrogation et manipulation de bases de données

Vous disposez d'un site de critiques gastronomiques dans lequel les utilisateurs peuvent évaluer des restaurants. Ce site s'appuie sur un modèle relationnel composé de trois relations.

Les restaurants sont stockés dans la relation *Restaurant*. Chaque restaurant est identifié par son *rID* et a un *nom*, un *type* (de cuisine) et une *adresse*.

La relation *Evaluateur* décrit les évaluateurs qui sont identifiés par leur *eID* et décrits par leur *pseudonyme* et leur date d'inscription (*dateInscription*).

Enfin, la relation *Evaluation* décrit les appréciations faites par les évaluateurs sur les restaurants. Chaque évaluation est identifiée par l'ensemble d'attributs *rID*, *eID*, *dateEval* où *rID* référence un restaurant dans la relation *Restaurant* et *eID* référence un évaluateur dans la relation *Evaluateur*. Une évaluation contient également une *note*.

Ceci donne le schéma relationnel suivant :

- *Restaurant*(<u>*rID*</u>, *nom*, *type*, *adresse*)
- *Evaluateur*(<u>*eID*</u>, *pseudonyme*, *dateInscription*)
- *Evaluation*(<u>$rID^{\#}$, $eID^{\#}$, *dateEval*</u>, *note*)

On suppose que les tables, associées à ce schéma relationnel, viennent d'être créées avec :

```
/* Création de la table Restaurant */
CREATE TABLE Restaurant(
rID integer,
nom varchar(50),
type varchar(30),
adresse varchar(50));
/* Création de la table Evaluateur */
```

```
CREATE TABLE Evaluateur(
eID integer,
pseudonyme varchar(30),
dateInscription date);
/* Création de la table Evaluation */
CREATE TABLE Evaluation(
rID integer,
eID integer,
dateEval date,
note integer);
```

1. **Implanter en SQL les contraintes et requêtes suivantes.**
   (a) *rID* est la clé primaire de la relation *Restaurant*.
   Réponse :
   ```
   ALTER TABLE Restaurant ADD CONSTRAINT pkResto PRIMARY KEY (rID);
   ```
   (b) *eID* est la clé primaire de la relation *Evaluateur*. Le couple (*pseudonyme*, *dateInscription*) est également clé.
   Réponse :
   ```
   ALTER TABLE Evaluateur
   ADD CONSTRAINT pkEvaluateur PRIMARY KEY (eID);
   ALTER TABLE Evaluateur
   ADD CONSTRAINT ucEvaluateur UNIQUE (pseudonyme,dateInscription);
   ```
   (c) Déterminer tous les restaurants qui ont été évalués par l'évaluateur dont l'*eID* est 135.
   Réponse :
   ```
   SELECT rID FROM Evaluation WHERE eID=135 ;
   ```

   (d) On souhaite obtenir les paires d'évaluateurs qui ont noté le même restaurant. Retourner le pseudonyme de ces deux évaluateurs en éliminant les duplications ((a, b) et (b, a) représentent la même paire d'évaluateurs).
   Réponse :
   ```
   SELECT DISTINCT w1.pseudonynme, w2.pseudonyme
   FROM Evaluation e1, Evaluation e2, Evaluateur w1, Evaluateur w2
   WHERE e1.rID = e2.rID AND w1.eID = e1.eID
   AND w2.eID = e2.eID AND w1.pseudonyme> w2.pseudonyme ;
   ```
   (e) Pour tous les cas où la même personne note deux fois le même restaurant et donne une note plus élevée la seconde fois, retourner le pseudonyme de l'évaluateur et le nom du restaurant.
   Réponse :
   ```
   SELECT DISTINCT w.pseudonyme, p.nom
   FROM Evaluation e, Evaluation e2, Restaurant p, Evaluateur w
   WHERE p.rID = e.rID AND w.eID = e.eID
   AND e.eID = e2.eID AND e.rID = e2.rID
   AND e2.dateEval > e.dateEval AND e2.note > e.note ;
   ```

   (f) Trouver le pseudonyme de tous les évaluateurs qui ont fait au moins 3 évaluations.
   Réponse :
   ```
   SELECT w.pseudonyme
   ```

```
FROM Evaluateur w, Evaluation r
WHERE w.eID = r.eID
GROUP BY r.eID
HAVING COUNT(r.eID) >= 3 ;
```

(g) Les restaurants (respectivement les évaluateurs) de la relation *Evaluation* doivent être contenus dans la relation *Restaurants* (respectivement *Evaluateur*) :
$Evaluation[rID] \subseteq Restaurant[rID]$ (respectivement $Evaluation[eID] \subseteq Evaluateur[eID]$).
Réponse :

```
ALTER TABLE Evaluation
ADD FOREIGN KEY (rID) REFERENCES Restaurant(rID);
ALTER TABLE Evaluation
ADD FOREIGN KEY (eID) REFERENCES Evaluateur(eID);
```

2. **Donner une définition des propriétés d'atomicité, de cohérence et de persistance énoncées en introduction du sujet.**
Réponse :
Atomicité des transactions : la propriété d'atomicité assure que l'ensemble des opérations associées à une transaction est exécuté dans son intégralité ou pas du tout. Si une partie d'une transaction ne peut être faite, alors il faut effacer toute trace de la transaction et remettre les données dans l'état où elles étaient avant la transaction.
Cohérence des transactions : la propriété de cohérence assure que chaque transaction amènera le système d'un état valide à un autre état valide. Tout changement à la base de données doit être valide selon toutes les règles définies (e.g., les contraintes d'intégrité, les déclencheurs). Si la transaction ne peut se terminer sur un état valide, alors toutes les opérations associées à la transaction sont annulées.
La propriété de Persistance assure que lorsqu'une transaction a été confirmée (commit), elle demeure enregistrée même à la suite d'une panne d'électricité, d'une panne de l'ordinateur ou d'un autre problème. En d'autres termes, les données ne sont pas perdues.

2 Inférence de dépendances fonctionnelles

On s'intéresse dans cette partie aux problèmes d'anomalies lors de mises à jour d'une base de données. De tels événements étant difficilement détectables, l'une des solutions possibles consiste à réduire les redondances présentes dans les bases de données.
Pour ce faire, l'une des approches usuelles consiste à étudier les dépendances fonctionnelles *entre les attributs de la base de données :*

Définition 1 (Dépendance fonctionnelle)**.** *Soient R une relation et X et Y deux sous-ensembles de l'ensemble des attributs de R. On dit que, dans une instance de relation r définie sur le schéma R, il existe une dépendance fonctionnelle entre X et Y (ou que X détermine fonctionnellement Y) si et seulement si pour tous n-uplets t_1 et t_2 de r, $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$.*

Une dépendance fonctionnelle entre X et Y est notée $R : X \rightarrow Y$ (ou simplement $X \rightarrow Y$ lorsque le schéma est implicite). La projection d'un n-uplet t sur les attributs X, notée $t[X]$, retourne les valeurs du n-uplet t pour chaque attribut de X.

Intuitivement, une dépendance fonctionnelle entre deux ensembles d'attributs X et Y exprime le fait que si l'on connait la valeur des attributs X alors on peut retrouver sans aucune

ambiguïté les valeurs correspondantes sur les attributs Y. Plus simplement, pour une valeur de X il ne correspond qu'une seule valeur de Y possible.

L'intérêt de l'étude des dépendances fonctionnelles réside dans le fait qu'il est possible, à partir de quelques dépendances fonctionnelles explicites, de déterminer par inférence l'ensembles des dépendances fonctionnelles valides dans un schéma relationnel.

Dans le cas général, une règle d'inférence s'exprime sous la forme d'une relation entre les prémisses *et la* conclusion *de la règle, représentée graphiquement par :*

$$\text{Règle d'inférence : } \frac{\text{Formule}_1 \quad ... \quad \text{Formule}_n}{\text{Conclusion}}$$

Si les prémisses Formule$_1$, .., Formule$_n$ sont vrais, alors Conclusion est vraie.

Dans le contexte des dépendances fonctionnelles, on dispose des règles d'inférence suivantes (X, Y, Z, W sont des sous-ensembles de l'ensemble des attributs de R, et pour deux ensembles X et Y, XY désigne par abus de notation l'union de X et Y ($XY = X \cup Y$)) :

$$\text{Réflexivité } (\sigma_{Re}) : \frac{Y \subseteq X}{X \to Y}$$

$$\text{Augmentation } (\sigma_A) : \frac{X \to Y}{WX \to WY}$$

$$\text{Transitivité } (\sigma_T) : \frac{X \to Y \qquad Y \to Z}{X \to Z}$$

Ces trois règles d'inférences forment le système d'inférence d'Armstrong. Le but de cette partie est d'étudier et d'utiliser le système d'inférence d'Armstrong.

3. **Expliquer ce qu'est une anomalie de mises à jour.**
   Réponse : Une anomalie de mise à jour a lieu lorsque, à la suite d'une modification de la base, des contraintes sémantiques valides se trouvent violées. Bien sûr, des mécanismes de contrôle sont intégrés aux SGBDR pour éviter ce genre de problèmes. Mais cela suppose une implémentation rigoureuse de toutes les contraintes et une gestion assez lourde de la base, avec certaines vérifications qui peuvent être assez coûteuses. Un compromis est alors atteint en faisant l'hypothèse suivante : le concepteur n'implémente que les clés et les clés étrangères. Le contrôle automatique de ces contraintes est peu coûteux par le SGBD et leur implémentation est toujours intégrées dans les SGBDR. Ainsi, on considère que toute mise à jour respecte les clés. r a une anomalie de mise à jour par rapport à un ensemble de contraintes F s'il est possible de modifier un n-uplet t tel que :
   — $r \cup \{t\}$ vérifie l'ensemble des clés induites par F.
   — $r \cup \{t\}$ viole une dépendance de F.
   Ainsi une insertion ou une modification d'un n-uplet peut entrainer une anomalie de m.à.j. Une suppression n'entraine pas une telle anomalie, toutefois elle peut entrainer une perte d'information. **En résumé : une anomalie de mise-à-jour se produit lorsqu'à la suite d'une insertion ou d'une modification d'un n-uplet, les contraintes de clés sont bien vérifiées mais d'autres contraintes (non vérifiées par le SGBD) se retrouvent violées.**

4. **Rappeler ce qu'est une *clé* dans une base de données. En quoi une clé détermine-t-elle fonctionnellement tous les autres attributs d'un schéma relationnel ?**
   Réponse :
   Une clé est un groupe d'attributs qui permet d'identifier de façon univoque un enregistrement (n-uplet) dans une relation. Elle permet ainsi d'éviter les doublons.
   Pour qu'une dépendance fonctionnelle $X \to Y$ ne soit pas satisfaite sur une instance de relation r, il faut deux n-uplets t_1, t_2 de r tels que $t_1[X] = t_2[X]$ et $t_1[Y] \neq t_2[Y]$. Or, il n'est pas possible que deux n-uplets aient la même valeur sur la clé, donc impossibilité de construire de tels contre-exemples. Ainsi pour X une clé, on a $X \to R$.

5. **Montrer que la règle d'inférence suivante est fausse.**

$$\frac{X \to Y \qquad Y \to Z}{Z \to X}$$

   Réponse :
   il suffit d'exhiber un contre-exemple avec l'instance suivante satisfaisant bien $X \to Y$ et $Y \to Z$ mais pas $Z \to X$. Les deux n-uplets suivants suffisent :

X	Y	Z
x_0	y_0	z_0
x_1	y_1	z_0

   Remarque : On peut aussi passer par les fermetures (abordées plus loin dans le sujet). Ainsi la fermeture de Z, $Z^+ = Z$. Elle ne contient pas X donc $Z \to X$ est fausse.

   On dit qu'un système d'inférence est correct s'il ne permet pas de produire des dépendances fonctionnelles non valides.

6. **Est-ce qu'un système d'inférence contenant la règle d'inférence de la question 5 est correct ?**
   Réponse :
   Nous avons montré que la règle d'inférence de la question 5 est fausse. Un système d'inférence contenant une telle règle ne peut donc pas être correct.

7. **Démontrer que les règles du système d'Armstrong (Réflexivité, Augmentation et Transitivité) sont correctes en exploitant la notion de dépendance fonctionnelle. Conclure sur le système d'inférence d'Armstrong.**
   Réponse :
   Soit R un schéma de relation et X, Y, Z trois sous-ensembles de R.
   (a) *réflexivité : soient $t_1, t_2 \in r$ où r une relation quelconque. Supposons $t_1[X] = t_2[X]$. Si $Y \subseteq X$, alors $t_1[Y] = t_2[Y]$ et on a $X \to Y$.*
   (b) *transitivité : c'est essentiellement la transitivité de l'implication logique. Soit r une relation quelconque sur R telle que $X \to Y$ et $Y \to Z$. Soient $t_1, t_2 \in r$ deux tuples quelconques tels que $t_1[X] = t_2[X]$. Puisque $X \to Y$ on a $t_1[Y] = t_2[Y]$. Puisque $Y \to Z$ on a $t_1[Z] = t_2[Z]$.*
   (c) *augmentation : on suppose $X \to Y$ et $t_1[WX] = t_2[WX]$. On a donc $t_1[W] = t_2[W]$ d'un part et $t_1[X] = t_2[X]$ d'autre part. Comme $X \to Y$ on déduit que $t_1[Y] = t_2[Y]$ et ainsi $t_1[WY] = t_2[WY]$.*

Nous avons prouvé les trois règles d'inférence du Système d'Armstrong. Ce système est donc correct.

8. **Soit le schéma de relation $R(A, B, C, D, E, F)$ (A, B, C, D, E et F étant les attributs de la relation R). En utilisant le système d'inférence d'Armstrong $\{\sigma_{Re}, \sigma_A, \sigma_T\}$, montrer que** :

   (a) $CE \rightarrow E$.
   Réponse :
   Par application directe de la réfléxivité : $E \subseteq CE$ donc $CE \rightarrow E$.

   (b) La dépendance fonctionnelle $BD \rightarrow C$ peut être inférée à partir de l'ensemble de dépendances fonctionnelles $\{AB \rightarrow BC, D \rightarrow A\}$.
   Réponse :
   Par augmentation de B sur $D \rightarrow A$, on a $BD \rightarrow AB$.
   Par transitivité ($BD \rightarrow AB$ et $AB \rightarrow BC$), on obtient $BD \rightarrow BC$.
   Par réflexivité, $C \subseteq BC$, donc $BC \rightarrow C$.
   Par transitivité ($BD \rightarrow BC$ et $BC \rightarrow C$), on conclut $BD \rightarrow C$.

   Sous forme arborescente, cela donne :

$$\dfrac{\sigma_T\ \dfrac{\sigma_A\ \dfrac{D \rightarrow A}{BD \rightarrow AB} \qquad AB \rightarrow BC}{BD \rightarrow BC} \qquad \dfrac{C \subseteq BC}{BC \rightarrow C}\ \sigma_{Re}}{BD \rightarrow C}$$

   (c) La dépendance fonctionnelle $AB \rightarrow F$ peut être inférée à partir de à partir de l'ensemble de dépendances fonctionnelles $\{AB \rightarrow C, A \rightarrow D, CD \rightarrow EF\}$.
   Réponse :
   Par augmentation par A sur $AB \rightarrow C$, on a $AB \rightarrow AC$.
   Par augmentation par C sur $A \rightarrow D$, on a $AC \rightarrow CD$.
   Par transitivité ($AB \rightarrow AC$ et $AC \rightarrow CD$) on a $AB \rightarrow CD$.
   $F \subseteq EF$ donc par réflexivité $EF \rightarrow F$.
   Par transitivité ($CD \rightarrow EF$ et $EF \rightarrow F$) , $CD \rightarrow F$.
   Par transitivité ($AB \rightarrow CD$ et $CD \rightarrow F$), on conclut que $AB \rightarrow F$.

   Sous forme arborescente, cela donne :

$$\dfrac{\sigma_T\ \dfrac{\sigma_A\ \dfrac{AB \rightarrow C}{AB \rightarrow AC} \qquad \sigma_A\ \dfrac{A \rightarrow D}{AC \rightarrow CD}}{AB \rightarrow CD} \qquad \dfrac{F \subseteq EF}{EF \rightarrow F}\ \sigma_{Re}}{AB \rightarrow F}$$

3 Fermeture transitive

On s'intéresse dans la suite de l'exercice à déterminer l'ensemble des dépendances fonctionnelles que l'on peut dériver à partir des règles d'inférences du système d'Armstrong. Pour cela, on va s'intéresser à la notion de fermeture transitive.
Soient X un ensemble d'attributs et Σ un ensemble de dépendances fonctionnelles sur le schéma de relation R. $\Sigma \models X \rightarrow Y$ signifie que l'ensemble de dépendances fonctionnelles Σ implique la dépendance fonctionnelle $X \rightarrow Y$. La fermeture transitive *de X par rapport*

à Σ, notée $X^+_{\Sigma,R}$ (ou simplement X^+ lorsque Σ et R sont évidents) est définie de la façon suivante :

$$X^+_{\Sigma,R} = \{A \in R \text{ tq } \Sigma \models X \rightarrow A\}.$$

Intuitivement, la fermeture transitive d'un ensemble d'attributs X correspond à tous les attributs qui sont déterminés fonctionnellement par X.

9. Soit l'algorithme 1 qui calcule la fermeture transitive d'un ensemble d'attributs par rapport à un ensemble de dépendances fonctionnelles.

Algorithme 1 : *Fermeture*(Σ, R, X)

Entrées : Σ un ensemble de dépendances fonctionnelles, X un ensemble d'attributs, le schéma R
Sortie : X^+ (la fermeture de X par Σ)

```
1 Cl := X;
2 done := false;
3 while (¬done) do
4 |   done := true;
5 |   forall W → Z ∈ Σ do
6 |   |   if W ⊆ Cl and Z ⊈ Cl then
7 |   |   |   Cl := Cl ∪ Z;
8 |   |   |   done := false;
9 return Cl
```

Etant donnés $\Sigma = \{A \rightarrow D; AB \rightarrow E; BF \rightarrow E; CD \rightarrow F; E \rightarrow C\}$ et le schéma de relation $R(A, B, C, D, E, F)$ (A, B, C, D, E et F étant les attributs de la relation R), que retourne l'algorithme pour les trois cas suivants ? :

(a) $X = F$
Réponse :
$F^+ = F$ (aucune Dépendance Fonctionnelle n'est exploitée).

(b) $X = BF$
Réponse :
$BF^+ = BCEF$

(c) $X = ABF$
Réponse :
$ABF^+ = ABCDEF$

Que peut-on dire de l'ensemble d'attributs ABF ?
Réponse :
A partir des attributs ABF, on peut retrouver tous les autres attributs de R, c'est-à-dire $ABF \rightarrow ABCDEF$. ABF est donc une clé de R.

10. Soit le lemme suivant : $\Sigma \models X \rightarrow Y$ si et seulement si $Y \subseteq X^+$.
On considère l'ensemble Σ de dépendances fonctionnelles suivant défini sur le schéma de relation $R(A, B, C, D, E, F, G)$:

$$\Sigma = \{A \rightarrow BCD; ABE \rightarrow G; CD \rightarrow E; EG \rightarrow ABD; BE \rightarrow F; FG \rightarrow A\}.$$

(a) **Montrer que $\Sigma \models A \to F$.**
Réponse :
Calculons la fermeture de A : $A^+ = ABCDEFG$
$F \in A^+$ donc on a bien $\Sigma \models A \to F$.
Remarque : il est aussi possible de prouver la dépendance fonctionnelle en passant le système d'inférence d'Armstrong ou en passant par une preuve s'appuyant sur la définition de satisfaction d'une dépendance fonctionnelle (i.e., poser deux n-uplets quelconques qui ont la même valeur sur A et montrer qu'ils ont la même valeur sur F) mais c'est inutilement fastidieux.

(b) **Montrer que BEF n'est pas une clé de R.**
Réponse :
Calculons la fermeture de BEF : $BEF^+ = BEF$
On ne retrouve pas tous les attributs de R dans la fermeture de BEF donc BEF n'est pas une clé.
Remarque : là aussi il est possible d'exhiber un contre-exemple : une instance de relation qui vérifie toutes les dépendances fonctionnelles de Σ où BEF n'est pas clé. Encore une fois, une telle solution prendrait plus de temps et serait plus difficile (assurer que toutes les dépendances de Σ sont valides sur l'exemple).

On s'intéresse maintenant aux propriétés algébriques de la fermeture. En algèbre, on appelle fermeture *une application $\phi : \wp(E) \to \wp(E)$ où $\wp(E)$ désigne l'ensemble des parties de E, qui est :*

Extensive : $X \subseteq \phi(X)$

Croissante : $X \subseteq Y \Rightarrow \phi(X) \subseteq \phi(Y)$

Idempotente : $\phi(\phi(X)) = \phi(X)$

11. **Montrer que la fermeture X^+ est bien une fermeture au sens algébrique du terme.**
Réponse :

Extensive *Soit $A \in X$, par l'axiome de réflexivité on a $X \to A$ et donc $A \in \{A \mid \Sigma \models X \to A\}$, soit $A \in X^+$.*

Croissante *Soit $X \subseteq Y$, il faut montrer que $X^+ \subseteq Y^+$. Considérons $A \in X^+$, par définition, il existe une preuve de $\Sigma \models X \to A$. Comme $X \subseteq Y$, par l'axiome de réflexivité on a $Y \to X$. Par transitivité on a une preuve de $\Sigma \models Y \to A$, c'est-à-dire $A \in Y^+$.*

Idempotente *On montre la double inclusion. Pour la première direction, on a $X \subseteq X^+$ et par monotonie $X^+ \subseteq (X^+)^+$. Pour la seconde direction, il faut prouver que $(X^+)^+ \subseteq X^+$. On prouve d'abord que $\Sigma \models X \to X^+$. Supposons sans perte de généralités que $X^+ = A_1 \dots A_n$, pour tout indice $1 \leq i \leq n$. Par définition, si $A_i \in X^+$ alors on a une preuve que $\Sigma \models X \to A_i$. Or on peut concaténer toutes ces preuves puis répéter la règle de composition[1] $\{X \to Y, X \to Z\} \models X \to YZ$ pour obtenir une preuve de $\Sigma \models X \to X^+$. Considérons $A \in (X^+)^+$, par définition, on a une preuve $\Sigma \models X^+ \to A$. Comme on a prouvé que $\Sigma \models X \to X^+$, on peut conclure par transitivité.*

1. cette dernière étant simplement une combinaison d'augmentation et de transitivité.

4 Equivalence de systèmes d'inférence

On dit qu'un système d'inférence est complet si toutes les dépendances valides peuvent être inférées par ce système. Le système d'inférence d'Armstrong est complet.
L'objectif de cette partie est de trouver un système d'inférence correct et complet de plus petite taille que le système d'Armstrong.

Considérons la nouvelle règle d'inférence suivante :

$$\text{Pseudo-transitivité } (\sigma_P) : \frac{X \to Y \qquad WY \to Z}{WX \to Z}$$

12. **Soit Σ un ensemble de dépendances fonctionnelles, montrer que toute preuve de $\Sigma \models X \to Y$ utilisant la règle σ_P peut être transformée en une preuve n'utilisant que σ_A et σ_T.**
Réponse :
Il s'agit de montrer que l'on peut prouver $WX \to Z$ à partir de $X \to Y$ et $WY \to Z$ en utilisant uniquement σ_A et σ_T :

$$\dfrac{\dfrac{X \to Y}{WX \to WY}\,\sigma_A \qquad WY \to Z}{WX \to Z}\,\sigma_T$$

13. **Montrer que toute preuve de $\Sigma \models X \to Y$ utilisant les règles σ_{Re}, σ_A et σ_T peut être transformée en une preuve n'utilisant que σ_{Re} et σ_P.**
Réponse :
Comme σ_{Re} appartient aux deux ensembles, il suffit de montrer la transitivité et l'augmentation à l'aide de la réflexivité et la pseudo-transitivité seulement. La transitivité est en fait un cas dégénéré de la pseudo-transitivité avec $W = \emptyset$:

$$\frac{X \to Y \qquad Y \to Z}{X \to Z}\,\sigma_P$$

L'augmentation s'obtient en posant $Z = WY$ dans la règle de pseudo-transitivité :

$$\dfrac{X \to Y \qquad \dfrac{WY \subseteq WY}{WY \to WY}\,\sigma_{Re}}{WX \to WY}\,\sigma_P$$

14. **En déduire que le système $\{\sigma_{Re}, \sigma_P\}$ est correct et complet pour l'inférence des dépendances fonctionnelles.**
Réponse :
Le système d'Armstrong $\{\sigma_{Re}, \sigma_A, \sigma_T\}$ est correct et complet (cf énoncé). Nous venons de montrer que toute preuve $F \models X \to Y$ utilisant les règles σ_{Re}, σ_A et σ_T peut être transformée en une preuve n'utilisant que σ_{Re} et σ_P. Donc le système $\{\sigma_{Re}, \sigma_P\}$ est correct et complet par équivalence à $\{\sigma_{Re}, \sigma_A, \sigma_T\}$.