Écrit Blanc d'informatique
Préparation au CAPES de Mathématiques
19 décembre 2018
Durée 4h

Instructions :

1. Les problèmes sont indépendants et *doivent* être traités sur des copies séparées.
2. La clarté et la précision des réponses font partie intégrante de l'évaluation.
3. Des graphes sans dessin ...

1 Problème 1 : Coloriage de Graphes

Préliminaires

Ce sujet est adapté du sujet X-ENS 2018, dont il reprend une partie de l'énoncé.

Solution: Problème et correction rédigés par L. Gonnord

Complexité Par **complexité en temps** d'un algorithme A, on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc) nécessaires à l'exécution de A dans le cas le pire.

Lorsque la complexité en temps ou en espace dépend d'un ou plusieurs paramètres $k_0, \dots, k_{r-1}$, on dit que A a une complexité $O(f(k_0, \dots, k_{r-1}))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $k_0, \dots, k_{r-1}$ suffisamment grandes (c'est à dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $k_0, \dots, k_{r-1}$, la complexité est au plus $Cf(k_0, \dots, k_{r-1})$.

On dit que la complexité en temps est **linéaire** quand f est une fonction linéaire des paramètres $k_0, \dots, k_{r-1}$, **polynomiale** quand f est une fonction polynomiale des paramètres $k_0, \dots, k_{r-1}$ et enfin **exponentielle** quand $f = 2^g$ où g est une fonction polynomiale des paramètres $k_0, \dots, k_{r-1}$.

Les complexités (en temps) des algorithmes **devront être justifiées**.

Graphes Rappelons qu'un graphe non-orienté est la donnée (S, A) de deux ensembles finis :

- un ensemble S de **sommets**, et
- un ensemble $A \subset S \times S$ d'**arêtes**, tel que pour tout couple de sommets (s, t), $s \neq t$ et, on a $(s, t) \in A$ si et seulement si $(t, s) \in A$.

Etant donné un graphe $G = (S, A)$, le **sous-graphe induit** par un ensemble de sommets $T \subset S$ est $(T, A \cap (T \times T))$.

Soit $G = (S, A)$ un graphe et soit $s \in S$ un sommet de G. Un **voisin** de s est un sommet t de G qui est relié à s par une arête, c'est à dire tel que $(s, t) \in A$. On note $V(s)$ l'ensemble des voisins de s. Le **degré** $d(s)$ de s est le cardinal de $V(s)$. Le **degré** $d(G)$ de G est le maximum des degrés de ses sommets.

Un graphe est dit **étiqueté** lorsque l'on dispose d'une fonction, dite d'étiquetage, de l'ensemble de ses sommets vers un ensemble non vide arbitraire, que l'on appelle ensemble des étiquettes. Les étiquettes peuvent par exemple être des entiers, des listes ou des chaînes de caractères.

On dit qu'une fonction d'étiquetage L est un **coloriage** des sommets de $G = (S, A)$ lorsque deux sommets voisins ont toujours deux étiquettes distinctes (alors appelées **couleurs**), c'est à dire lorsque L vérifie la condition

$$\forall s, t \in S,\ (s, t) \in A \Rightarrow L(s) \neq L(t)$$

Un graphe est dit k-coloriable s'il admet un coloriage avec au plus k couleurs. Un graphe est dit colorié s'il est k-coloriable pour un $k > 0$.

Le **nombre chromatique** d'un graphe non orienté G, noté $\chi(G)$, est le nombre minimal k tel que G est k-coloriable. Cet énoncé porte sur le calcul de nombres chromatiques et de coloriages.

Représentation des graphes étiquetés On se fixe dans cet énoncé une représentation des graphes par matrices d'adjacence (avec 0/1). On se fixe également comme convention que les étiquetages des graphes sont tous à valeurs entières. L'étiquetage d'un graphe sera donné par une liste d'entiers. Un graphe non orienté $G = (S, A)$ avec $S = \{0, \ldots, n-1\}$ est représenté par une valeur `gphe` de type `graphe` telle que pour $i, j \in S$, `gphe[i,j]=1` si et seulement si $(i, j) \in A$. Le graphe G étant supposé non orienté, on a alors également par symétrie `gphe[j,i]=1`. Pour un étiquetage `etiq` de `gphe`, l'étiquette du sommet i de `gphe` est donnée par `etig[i]`.

En Python/igraph, on crée une matrice d'adjacence comme ceci :

```
# ceci est un graphe avec 6 sommets
adj = np.array([[0, 1, 1, 0, 0, 0],
                [1, 0, 0, 1, 0, 0],
                [1, 0, 0, 0, 1, 0],
                [0, 1, 0, 0, 1, 0],
                [0, 0, 1, 1, 0, 1],
                [0, 0, 0, 0, 1, 0]])
```

et un étiquetage comme ceci :

```
etiq = [1, 1, 2, 2, 1, 0]
```

1.1 Coloriage

Question #1

Indiquer, pour chacun des graphes de la figure 1, si l'étiquetage proposé est un coloriage.

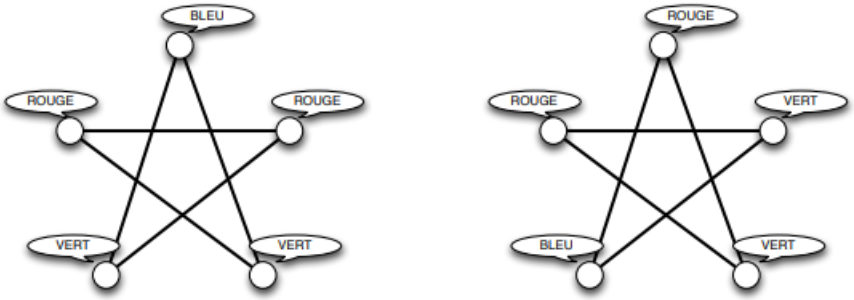

FIGURE 1 – Graphes (bien) coloriés ?

Solution: non (car liaison 2,5 avec deux sommets rouges) - oui (en énumérant les liaisons)

Question #2

Donner le nombre chromatique, ainsi qu'un exemple de coloriage pour le **graphe de Petersen** de la Figure 2.

Solution:
Le graphe contient des cycles de longueur impaire, par exemple (0, 4, 3, 9, 6) : il n'est pas 2-coloriable. En effet il faudrait que les couleurs soient alternées dans le cycle et on arriverait à deux couleurs égales pour les sommets 0 et 6 qui sont reliés.

Par contre il est 3-coloriable (donner le coloriage)

La vérification de la propriété de coloriage est le problème suivant.
— Entrée : un graphe G et un étiquetage L de G.
— Question : L est-il un coloriage de G ?

Question #3

Ecrire une fonction `est_col`, telle que `est_col(gphe,etiq, taille)` renvoie `True` si et seulement si `etiq` est un coloriage de `gphe` (taille est le nombre de sommets).

Dans le cas où la taille de l'étiquetage est strictement inférieure au nombre de sommets du graphe, la fonction renvoie `False`. On demande une complexité quadratique en le nombre de sommets du graphe.

Solution:

```
def est_col(graphe, etiquetage, taille):
    """ retourne True si la table d'étiquetage donnée en paramè
       tre
    est bien un coloriage correct du graphe donné par la matrice
    d'adjacence -- cas d'erreur non traité
    """
    for i in range(0, taille):
        colori = etiquetage[i]
        for j in range(0, i):
            if graphe[i, j] == 1 and graphe[j] == colori:
                return False
    return True
```

Question #4

Démontrer que le calcul du nombre chromatique d'un graphe peut s'effectuer en temps exponentiel en le nombre de sommets. *indication : on se demandera combien il existe de coloriages à k couleurs, pour k inférieur au nombre de sommets du graphes.*

Solution: Un graphe de n sommet possède un n-coloriage, il suffit de donner une couleur distincte à chaque sommet.

Pour tester son nombre chromatique il suffit de tester, pour k variant de 2 à $n-1$ s'il admet un coloriage à k couleurs.

Pour cela on peut tester tous les coloriages : il y en a k^n.

La complexité est alors majorée par

$$An^2\left(\sum_{k=2}^{n-1} k^n\right) \leq An^2.n.n^n = A2^{(n+3)\log_2(n)} \leq B2^{n^2}$$

la complexité est exponentielle.

1.2 2-coloriage

Nous avons vu à la question 4 que le calcul du nombre chromatique peut s'effectuer en temps exponentiel en le nombre de sommets du graphe. Dans le cas général, on ne sait aujourd'hui pas faire mieux. Pour obtenir de meilleures bornes de complexité, il faut donc se limiter à des sous-problèmes. On considère dans cette partie le cas du 2-coloriage.

Graphe biparti. Un graphe G est **biparti** lorsque l'ensemble de ses sommets S peut être divisé en deux sous-ensembles disjoints T et U (non vides), tels que chaque arête a une extrémité dans T et l'autre dans U.

Question #5

Démontrer (proprement) qu'un graphe G est biparti si et seulement s'il est 2-coloriable.

Solution: Si G est biparti, alors ses sommets peuvent se diviser en deux sous-ensembles T et U. On attribue alors la couleur 1 aux sommets de T, et 2 aux sommets de U. La propriété de coloration est alors instantanément vérifiée.

Inversement, si G possède une 2-coloration, alors on peut appeler T les sommets recevant la couleur 1 et U les sommets recevant la couleur 2. Aucune arête ne peut relier deux sommets de couleur 1 (ou 2), et les arêtes vont donc d'un sommet de T vers un sommet de U.

On se propose de programmer la vérification de la 2-colorabilité des graphes en procédant comme suit. On effectue un parcours du graphe en profondeur au cours duquel on construit une 2-coloration du graphe. On se donne pour ce faire trois étiquettes, disons -1, 0 et 1. L'étiquetage est initialisé à -1 pour tous les sommets, et on teste la 2-colorabilité avec 0 et 1. Le principe de l'algorithme est le suivant.

(1) On choisit un sommet s d'étiquette -1.
(2) On colorie les sommets rencontrés lors du parcours en profondeur à partir de s, en alternant entre les couleurs 0 et 1 à chaque incrémentation de la profondeur, et en vérifiant si les sommets déjà coloriés rencontrés sont d'une couleur compatible.
(3) Enfin, s'il reste des sommets d'étiquette -1, alors on revient au point (1).

Question #6

Écrire une fonction récursive `explo(gr, i, k, taille, etiq)` qui réalise le parcours en profondeur du graphe `gr` à partir du sommet `i`, en fixant la couleur de ce sommet à `k` dans `etiq`. Avec quelle couleur doivent être coloriés les voisins du sommet k ?

Comme les paramètres des fonctions python sont mutables, toute modification apportée à `etiq` *sera (automatiquement) enregistrée à la sortie de la fonction.* -1 *dans le tableau* `etiq` *dénote le fait qu'un sommet n'a pas encore été vu.*

Solution: pas testée.

```
def explo(gr, i, k, taille, etiq):  # etiq est modifiée à la fin
    de la fonction
    etiq[i] = k
    for j in range(0, taille):  # (jusqu'à taille-1, donc)
        if gr[i, j] == 1 and etiq[j] == -1:  # je n'ai pas
            encore vu!
```

```
            explo(gr, j, 1-k, taille, etiq)
    return
```

Question #7

En utilisant la fonction précédente, écrire une fonction `deuxcol(gphe, taille)` qui calcule et retourne un 2 coloriage (0/1) du graphe si celui-ci est 2-coloriable. Le sommet 0 sera colorié en 0. Faire un exemple.

On demande une complexité quadratique en le nombre de sommets du graphe (justifier !). Le comportement de la fonction est laissé au choix du candidat lorsque le graphe n'est pas 2-coloriable.

Indication : l'initialisation des étiquettes peut être réalisée avec : `etiq = [-1] * taille`

Solution:

```
def deuxcol(graphe, taille):
    etiq = [-1] * taille
    for i in range(0, taille):
        if etiq[i] == -1:
            explo(graphe, i, 0, taille, etiq)
    return etiq
```

Ici, si le graphe n'est pas 2-coloriable, alors le programme renvoie une coloration fausse (on ne détecte pas les erreurs si le sommet j est déjà colorié).

Le programme `explo` ne peut être lancé qu'une seule fois par sommet au maximum, et sa complexité est en $O(n)$. On arrive donc à une complexité en $O(n^2)$ dans le pire des cas.

1.3 Glouton

Dans cette partie, nous allons étudier un algorithme permettant de colorier un graphe en temps polynomial, mais donnant en général un coloriage sous-optimal : le coloriage obtenu peut dans certains cas utiliser plus de couleurs que le coloriage optimal.

Cet algorithme prend en paramètre un ordre sur les sommets du graphe, que l'on appelera **ordre de numérotation**.
Par exemple, $1 < 3 < 4 < 0 < 2 < 6 < 5 < 9 < 8 < 7$ et $0 < 7 < 2 < 5 < 4 < 6 < 8 < 1 < 3 < 9$ sont deux ordres de numérotation des sommets du graphe de Petersen (Figure 2).

Pour un graphe `gphe` à n sommets, on implémente un ordre de numérotation de ses sommets par un tableau `num` de n valeurs entières, tel que `num[k]`$=j$ si et seulement si le sommet j apparaît en $(k+1)$-ième position dans l'ordre.
L'**algorithme glouton** construit un coloriage L d'un graphe G en utilisant au plus $d(G)+1$ couleurs. Son principe est le suivant :
On parcourt la liste des sommets du graphe, dans l'ordre de numérotation des sommets donné. Pour chaque sommet s parcouru :
(1) On calcule l'ensemble $C(s) = \{L(t)/\ t \in V(s)\}$ des couleurs déjà données aux voisins de s.
(2) On cherche le plus petit entier naturel c qui n'appartient pas à $C(s)$.
(3) On pose $L(s) = c$.

Question #8

Considérons le graphe de Petersen (Figure 2) et les deux ordres de numérotation :

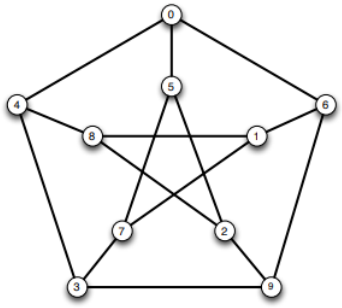

FIGURE 2 – Le graphe de Peterson à 10 sommets

```
num1 = [1, 3, 4, 0, 2, 6, 5, 9, 8, 7]
num2 = [0, 7, 2, 5, 4, 6, 8, 1, 3, 9]
```

Donner les coloriages obtenus par l'algorithme glouton décrit ci-dessus pour le graphe de Petersen et chacun de ces deux ordres de numérotation, ainsi que les nombres de couleurs correspondants.

Solution: Avec le premier ordre, on trouve comme coloration pour les sommets : $(0, 0, 0, 0, 1, 1, 1, 2, 2, 2)$, donc trois couleurs.

Avec le second, on trouve comme coloration : $(0, 3, 0, 2, 1, 1, 1, 0, 2, 3)$, donc quatre couleurs.

Question #9

Écrire une fonction `min_couleur_possible(gr, etiquetage, taille, sommet)` qui pour un graphe `gphe` à *taille* sommets, un étiquetage `etiquetage` à valeurs dans $\{-1, \ldots, n-1\}$, renvoie le plus petit entier naturel n'appartenant pas à l'ensemble $\{\texttt{etiq}[t] / \ t \in V(\texttt{s})\}$. On demande une complexité $O(n)$.

Solution:

```
def min_couleur_possible (gr, etiquetage, taille, sommet):
    coul = [False] * taille
    for i in range(0, taille):
        if gr[sommet][i] == 1 and etiquetage[i] != -1:
            coul[etiquetage[i]] = True
    cpt = 0
    while coul[cpt]:
        cpt = cpt+1
    return cpt
```

Question #10

Écrire une fonction `colore_glouton` : pour un graphe `gphe` de taille *taille* et un ordre de numérotation `num` de ses sommets, l'appel `colore_glouton(gphe, num, taille)` renvoie le coloriage glouton de `gphe` selon l'ordre, avec au plus $d+1$ sommets, où d est le degré de `gphe`. On demande une complexité $O(n^2)$, où n est le nombre de sommets de `gphe`.

Dans le cas où le tableau `num` contient autre chose qu'un ordre de numérotation des sommets de `gphe`, le résultat de la fonction est laissé au choix, mais il faudra préciser.

Solution:

```
def colore_glouton(gr, numerotation, taille):
    couleurs = [-1] * taille
    for i in range(0, taille):
        k = numerotation[i]
        couleurs[k] = min_couleur_possible(gr, couleurs, taille,
            k)
    return couleurs
```

Question #11

Montrer que l'algorithme de coloriage glouton construit toujours un coloriage, et que ce coloriage utilise au plus $d+1$ couleurs, où d est le degré du graphe en entrée.

Solution: La fonction `min_couleur_possible` renvoie une couleur qui n'a pas été affectée aux voisins du sommet passé en paramètre. Lorsqu'une couleur est affectée à un sommet elle ne pourra plus être affectée aux voisins qui suivent dans l'ordre de numération. De plus chaque sommet reçoit une couleur lorsque `num` est un ordre de numération.

On a bien construit un coloriage du graphe.

Dans l'algorithme glouton chaque sommet est colorié par une couleur non employée par ses voisins déjà coloriés, comme il admet au plus $d(G)$ voisins, la couleur choisie est la plus petite parmi une ensemble d'au plus $d(G)$ entiers positif : elle est donc majorée par $d(G)$.
Ainsi la coloration construite admet au plus $d(G)+1$ couleurs.

Question #12

Soit G un graphe. Montrer que pour tout coloriage L de G, il existe un ordre de numérotation des sommets tel que le coloriage glouton L' associé vérifie $L'(s) \leq L(s)$ pour tout sommet s de G. En déduire qu'il existe une numérotation des sommets telle que l'algorithme glouton renvoie un coloriage optimal.

Solution: Soit L un coloriage de G. On considère une numération des sommets qui les classe par numéro de couleur croissante.
Comme deux sommets de même couleur ne sont pas adjacents, lors de l'appel de `min_couleur_possible` les seuls sommets voisins de s qui seront considérés auront une couleur strictement inférieure à celle de s, $L(s)$. La couleur choisie, $L'(s)$, ne pourra donc pas être strictement supérieure à $L(s)$: on a $L'(s) \leq L(s)$ pour tout s.
Si on part d'un coloriage optimal pour construire la numération des sommets on aboutit donc à un coloriage dont le maximum est majoré par celui du coloriage optimal : ce sera donc aussi un coloriage optimal.

Les questions 7 et 11 indiquent que l'efficacité de l'algorithme glouton est en grande partie dépendante de l'ordre dans lequel on choisit de parcourir les sommets du graphe. L'ordre correspondant à la représentation choisie du graphe (dans notre cas, les indices de la matrice d'adjacence, c'est à dire la permutation identité) est le plus simple à calculer, mais a peu de chances d'être efficace. A contrario, on pourrait essayer de déterminer l'ordre optimal, dont on a prouvé l'existence à la question 11, mais cela n'apporte aucun bénéfice vis-à-vis de la complexité temporelle du problème.

Une alternative est donnée par l'optimisation de Welsh-Powell. L'idée est de parcourir l'ensemble des sommets du graphe par ordre de degré décroissant. Le tri des sommets par degré décroissant ne prend pas plus de temps que le parcours glouton, mais permet d'obtenir un algorithme raisonnablement efficace en pratique.

Question #13

Écrire une fonction de tri **décroissant** d'un tableau / d'une liste d'entiers en Python, et donner sa complexité.

Solution: Un joli tri en $O(n^2)$, le tri sélection qui sélectionne le max du sous-tableau à droite de i...

```
def tri_selection(inputt, taille):  # tri decroissant
for i in range(0, taille -1):
    imax = i
    dmax = inputt[i]
    for j in range (i+1, taille):
        dcur = inputt[j]
        if dcur > dmax:
            imax = j
```

```
                dmax = dcur
        echanger(inputt, i, imax)
    return inputt
```

Question #14

Écrire une fonction `degres(gr, taille)` qui retourne la liste des degrés des sommets du graphe.

Solution: J'ai choisi de faire des listes de paires (numéro, degré)

```
    deg = [0] * taille
    for i in range(0, taille):
        for j in range(0, taille):
            if gr[i, j] == 1:
                deg[i] = deg[i]+1
    return [(i, deg[i]) for i in range(0, taille)]
```

Question #15

En déduire une fonction `welsh_powell` qui implémente l'optimisation de Welsh-Powell. *Une modification des deux algorithmes précédents pourra être utile.* Pourquoi un tri quadratique est-il suffisant ?

Solution: Comme l'algorithme `colore_glouton` est de complexité quadratique on peut choisir un algorithme lui-même quadratique pour construire une numération des sommets sans augmenter la complexité.

Je décide ensuite de modifier pour trier selon le deuxième élément des paires, et ensuite je filtre et je recolle avec la fonction glouton :

```
def tri_selection2(inputt, taille):  # tri decroissant selon le
   second du tuple
    for i in range(0, taille -1):
        imax = i
        dmax = inputt[i][1]
        for j in range(i+1, taille):
            dcur = inputt[j][1]
            if dcur > dmax:
                imax = j
                dmax = dcur
        echanger(inputt, i, imax)
    return inputt

def welsh_powell(gr, taille):
    deg = degres_graphe(gr, taille)
    apres_tri = tri_selection2(deg, taille)
    num = [un for (un, deux) in apres_tri]
    return(colore_glouton(gr, num,  taille))
```

2 Problème 2 : Bases de données relationnelles

On considère à titre d'exemple la base de données *Stanford* illustrée dans la table 1.

Student

sID	sName	GPA	sizeHS
123	Amy	3,9	1.000
234	Bob	3,6	1.500
345	Craig	3,5	500
456	Doris	3,9	1.000
567	Edward	2,9	2.000
678	Fay	3,8	200
789	Gary	3,4	800
987	Helen	3,7	800
876	Irene	3,9	400
765	Jay	2,9	1.500
654	Amy	3,9	1.000
543	Craig	3,4	2.000

Apply

sID	cName	major	dec.
123	Stanford	CS	Y
123	Stanford	EE	N
123	Berkeley	CS	Y
123	Cornell	EE	Y
234	Berkeley	biology	N
345	MIT	bioeng.	Y
345	Cornell	bioeng.	N
345	Cornell	CS	Y
345	Cornell	EE	N
678	Stanford	history	Y
987	Stanford	CS	Y
987	Berkeley	CS	Y
876	Stanford	CS	N
876	MIT	biology	Y
876	MIT	marine bio.	N
765	Stanford	history	Y
765	Cornell	history	N
765	Cornell	psych.	Y
543	MIT	CS	N

College

cName	state	enrollment
Stanford	CA	15.000
Berkeley	CA	36.000
MIT	MA	10.000
Cornell	NY	21.000

TABLE 1 – Base de données *Stanford* d'exemple

2.1 Bases de SQL

Solution: Problème et correction rédigés par R. Thion.

Question #1

Donner une phrase simple en français pour dire ce que calcule la requête suivante et donner le résultat.

```
SELECT DISTINCT sName, GPA
FROM Student INNER JOIN Apply ON Student.sID = Apply.sID
WHERE major = 'CS' AND decision = 'Y';
```

Solution: Les noms et les moyennes (GPA) des étudiants qui ont accepté une candidature en informatique.

```
sName       GPA
----------  ----------
Amy         3.9
Craig       3.5
Helen       3.7
```

Question #2

Donner une phrase simple en français pour dire ce que calcule la requête suivante et donner le résultat.

```
SELECT sName, COUNT(*)
FROM Student INNER JOIN Apply ON Student.sID = Apply.sID
GROUP BY Student.sID, Student.sName;
```

Solution: Le nombre de candidatures de chaque étudiant.

```
sName       COUNT(*)
----------  ----------
Jay         3
Irene       3
Helen       2
Craig       4
Fay         1
Bob         1
Craig       1
Amy         4
```

Question #3

Écrire une requête qui calcule, les couples d'établissements qui ont l'informatique *'CS'* en commun.

Solution:

```
SELECT DISTINCT A1.Cname, A2.cname
FROM Apply A1 INNER JOIN Apply A2 on A1.major = A2.major
WHERE A1.cname < A2.cname AND A1.major = 'CS';
```

Question #4

Écrire une requête qui calcule, pour formation (établissement et majeure) le nombre de candidatures acceptées.

Solution:

```
SELECT cName, major, count(*)
FROM Apply
WHERE decision = 'Y'
GROUP BY Apply.cName, Apply.major;
```

Question #5

Écrire une requête qui calcule les noms des établissement avec la plus grande capacité d'accueil (*enrollment*). Utiliser une requête imbriquée dans le `FROM`.

Solution:

```
SELECT cName
```

```
FROM (SELECT max(enrollment) as em
      FROM College) C INNER JOIN College on C.em = College.enrollment;
```

Question #6

On considère la requête suivante. Que calcule-t-elle ? Justifier. La réécrire en une requête plus élégante qui calcule la même chose.

```
SELECT sName, GPA, sizeHS
FROM Student
GROUP BY sName, GPA, sizeHS;
```

Solution: En faisant tous les `GROUP BY`, on assure l'unicité, mais il y a un opérateur spécifique pour ça.

```
SELECT DISTINCT sName, GPA, sizeHS
FROM Student;
```

2.2 Notions de clef

On appelle *clef* un ensemble *minimal* d'attributs d'une relation qui permet d'identifier *uniquement* un tuple dans une instance de relation.

Question #7

Pour chaque relation de la base de données *Stanford*, donner ses clefs en vous appuyant sur la sémantique intuitive des attributs.

Solution:
- *Student* : sID est naturelle, pas d'autres combinaisons
- *College* : $cName$ est naturelle, *enrollment* l'est aussi, mais par hasard.
- *Apply* : là c'est le triplet $sID, cName, major$.

Question #8

Par rapport à la notion de clef, que calcule la requête suivante ?

```
SELECT sName, GPA, count(*)
FROM Student
GROUP BY sName, GPA
HAVING count(*) > 1;
```

Solution: Les contre-exemples au fait que le couple $sName, GPA$ soit clef.

Question #9

L'attribut *enrollment* de la relation *College* est-il clef ? Qu'en pensez-vous ?

Solution: C'est un hasard que les établissements aient des capacités toutes différentes qui permet donc de les identifier. Il ne faudrait pas mettre cette contrainte dans la base.

Question #10

Justifier de l'adjectif *minimal* dans la définition de clef, autrement dit, que se passe-t-il si on dit qu'une clef « est un ensemble d'attributs d'une relation qui permet d'identifier uniquement un tuple. »

Solution: Sinon, tout sur-ensemble le serait.

2.3 Formalisation logique de SQL

On s'intéresse maintenant au fragment restreint de SQL représenté ci-dessous, où les conditions booléennes écrites `EQx` sont toutes des conjonctions de tests d'égalités entre des attributs ou des constantes. Le but de l'exercice est de donner une sémantique formelle logique à ces requêtes.

```
SELECT DISTINCT A1, A2, ..., An
FROM R1
  INNER JOIN R2 ON (EQ2)
  INNER JOIN R3 ON (EQ3)
  ...
  INNER JOIN Rn ON (EQn)
WHERE EQ
```

Par exemple, la requêtes suivante respecte ces contraintes.

```
SELECT DISTINCT sName, decision, state
FROM Student
  INNER JOIN Apply ON Student.sID = Apply.sID
  INNER JOIN College ON Apply.cName = College.cName
WHERE Major = 'CS';
```

On veut montrer qu'à chaque requête SQL correspond une formule de logique du premier ordre associée qui a la même sémantique et vice-versa. Soit $\mathbb{D}$ un ensemble de constantes et $\mathcal{V}$ un ensemble de variables, tous deux supposés infinis dénombrables. Les formules considérées sont définies comme suit :

- si R est un symbole (de relation) d'arité n, alors $R(x_1, \ldots, x_n)$ où les x_i sont des variables ou des constantes constitue *une formule atomique* ;
- un *corps* est un ensemble *fini* de formules atomiques noté $FA_1, \ldots, FA_n$
- une *requête logique* est une expression de la forme $q(\bar{x}) \leftarrow B$ où B est un corps $B = FA_1, \ldots, FA_n$ avec $\bar{x}$ est une liste de variables, chacune apparaissant dans au moins une formule atomique du corps. Les variables de $\bar{x}$ sont appelées variables de tête.

A titre d'exemple, voici la représentation en requête logique de la requête SQL précédente :

$$q(s, d, st) \leftarrow \text{Student}(s, n, g, h) \wedge \text{Apply}(s, c, \text{'CS'}, d) \wedge \text{College}(c, st, e)$$

Soit $\mathcal{I}$ *une instance de base de données*, c'est-à-dire une application qui à chaque symbole de relation R d'arité n fait correspondre une relation $\mathcal{I}(R) \subseteq \mathbb{D}^n$. Soit $\mu : \mathcal{V} \to \mathbb{D}$ une fonction qui à chaque variable fait correspondre une constante. La fonction μ est étendue respectivement :

- aux constantes par l'identité, $\mu(c) = c$ si $c \in \mathbb{D}$;
- aux formules atomiques par $\mu(R(x_1, \ldots, x_n)) = R(\mu(x_1), \ldots, \mu(x_n))$;
- aux corps $\mu(FA_1, \ldots, FA_n) = \mu(FA_1), \ldots, \mu(FA_n)$;
- aux listes de variables $\mu(\bar{x}) = \mu(x_1, \ldots, x_n) = (\mu(x_1), \ldots, \mu(x_n))$.

L'ensemble des réponses à une requête logique $Q = q(\bar{x}) \leftarrow B$ sur une instance est défini ainsi :

$$\mathsf{Ans}(Q, \mathcal{I}) = \{\mu(\bar{x}) \mid \exists \mu, \bigwedge_{R_i(x^i_1, \ldots, x^i_{n^i}) \in B} \mu(x^i_1, \ldots, x^i_{n^i}) \in \mathcal{I}(R_i)\}$$

Question #1

Pourquoi le mot-clef `DISTINCT` est-il obligatoire dans le fragment de SQL considéré pour la correspondance avec le formalisme des requêtes logiques ?

Solution: Car la sémantique est ensembliste et SQL ne l'est pas (*bag semantics* voire même *liste* de résultats).

Question #2

Exprimer formellement (avec les symboles consacrés du formalisme) que si le corps d'une requête logique est inclus dans une autre qui a les même variables de tête, alors quelle que soit l'instance, tous les résultats de la requête incluante sont des résultats de la requête incluse.

Solution: Soient $Q_1 = q_1(\bar{x}) \leftarrow FA^1_1, \ldots, FA^n_1$ et $Q_2 = q_2(\bar{x}) \leftarrow FA^1_2, \ldots, FA^m_2$. Si $FA^1_1, \ldots, FA^n_1 \subseteq FA^1_2, \ldots, FA^m_2$, alors $\forall \mathcal{I}, \mathsf{Ans}(Q_2, \mathcal{I}) \subseteq \mathsf{Ans}(Q_1, \mathcal{I})$.

Question #3

Le prouver.

Solution: Si μ est un témoin de l'occurrence du corps de Q_2 dans $\mathcal{I}$, i.e., tel que $\mu(R_i(x^i_1, \ldots, x^i_{n^i})) \in \mathcal{I}(R_i)$ quand $R_i(x^i_1, \ldots, x^i_{n^i}) \in FA^i_1, \ldots, FA^n_1$, alors, comme toute formule atomiques de Q_2 est aussi dans Q_1, μ est aussi un témoin de l'occurrence du corps de Q_1 dans $\mathcal{I}$ et les projections sur $\bar{x}$ sont les mêmes.

Question #4

Une requête dont la liste des variables de tête est vide est dite *booléenne*. Justifier l'emploi de l'adjectif.

Solution: Il n'y a que deux résultats possibles à une telle requête, quelle que soit l'instance $\mathcal{I}$, soit $\emptyset$ (quand le corps n'a pas d'occurrence dans $\mathcal{I}$) soit $\{\emptyset\}$ (quand il en a une).

Question #5

Le formalisme des requêtes logiques est *positionnel* alors que SQL est dit *nommé* : les attributs sont référencés par leurs positions (au sein des formules atomiques) dans le premier et par leurs noms dans le second. Expliquer de quelle structure de données $\mathcal{M}$ vous avez besoin pour passer des requêtes logiques aux requêtes SQL. Donner un exemple sur la base *Stanford* de cette correspondance.

Solution: Il faut un dictionnaire qui à chaque symbole de relation et position associe un nom d'attribut. Par exemple $\mathcal{M}$:

$$\begin{aligned}\mathcal{M}(\text{Student}, 1) &= \text{sID}\\ \mathcal{M}(\text{Student}, 2) &= \text{sName}\\ &\dots\\ \mathcal{M}(\text{Apply}, 2) &= \text{major}\\ &\dots\\ \mathcal{M}(\text{College}, 1) &= \text{cName}\end{aligned}$$

Question #6

Expliquer le rôle du partage de variables dans le formalisme des requêtes logiques.

Solution: C'est le test d'égalité.

Question #7

Expliquer, avec un algorithme de haut-niveau, comment générer une requête SQL à partir d'une requête logique $Q = q(\bar{x}) \leftarrow R_1(x_1^1, \dots, x_{n^1}^1), \dots, R_m(x_1^m, \dots, x_{n^m}^m)$ telle que ses résultats soient les mêmes. On supposera l'existence de la structure de donnée $\mathcal{M}$ discutée avant.

Solution:

Écrit Blanc d'informatique
Préparation au CAPES de Mathématiques
19 décembre 2018
Représentation et codage des données, l'algèbre de Boole

Exercice, 1. Répondez aux questions de cours suivantes : pb et correction par H. Ladjal

Question 1 (1 pt)**.** Pourquoi les ordinateurs actuels sont-ils binaires ?

Réponse :
- Le binaire est utilisé en informatique car il permet de modéliser le fonctionnement des composants de commutation comme le TTL ou le CMOS. Les processeurs des ordinateurs actuels sont composés de transistors ne gérant chacun que deux états.
- Par exemple le chiffre 0 sera utilisé pour signifier une absence de tension, et le chiffre 1 pour signifier sa présence. Cette marge de tolérance permet de pousser les cadences des microprocesseurs à des valeurs atteignant sans problème plusieurs gigahertz.
- En n'utilisant que 2 chiffres (base 2), les processeurs effectuent des calculs très rapidement et très simplement sur des nombres comportant uniquement des 0 et des 1.

Question 2 (1 pt)**.** Expliquez une méthode permettant de coder en machine des nombres entiers négatifs. Donnez un exemple pour illustrer cette méthode.
Réponse : par exemple le complément à 2 : CA2 (nombre)= CA1(nombre) +1 (Voir le cours)

Exercice, 2. En utilisant exlusivement l'algèbre de Boole :

Question 3 (1 pt)**.** Démonterez que : $A.(\overline{A}+\overline{B}).(A+B) = A.\overline{B}$
Réponse : $A.(\overline{A}+\overline{B}).(A+B) = (A.\overline{A}+A.\overline{B}).(A+B)$ Avec $A.\overline{A}=0$)
$= A.\overline{B}.A + A.\overline{B}.B = A.\overline{B}$

Question 4 (1 pt)**.** Simplifiez au maximum la fonction : $F = \overline{(A \bigoplus B) + \overline{(\overline{A}+B)}}$
Réponse : $F = \overline{(A.\overline{B}+\overline{A}.B)} + A.\overline{B}$ avec $(A.\overline{B}+A.\overline{B}=A.\overline{B})$

$F = \overline{A.\overline{B}+\overline{A}.B} = \overline{A \bigoplus B}$

Exercice, 3. Effectuez les conversions suivantes.

Question 5 (0.5 pt)**.** Convertir le nombre hexadécimal $F1ACF2B$ en binaire :
Réponse : $(F1ACF2B)_{16} = (11110001101011001111001010 11)_2$

Question 6 (0.5 pt)**.** Convertir le nombre binaire à virgule $101,101$ en décimal :
Réponse : $(101,101)_2 = (2^0 + 2^2 + 2^{-1} + 2^{-3})_{10}$

Question 7 (0.5 pt)**.** Convertir le nombre décimal 31 en base 7 :
Réponse : division par 7 $(31)_{10} = (43)_7$

Question 8 (0.5 pt)**.** Convertir le nombre décimal 57 en octal :
Réponse : division par 8 $(57)_{10} = (71)_8$

Question 9 (1 pt)**.** Représentation binaire des entiers négatifs : Coder sur 4 bits, les entiers 7, -2, -7 avec les représentations suivantes : Réponse :

- Signe et valeur absolue : (7)=0111, (-2)=1010 ,(-7)=1111
- Complément à 1 : (7)=0111, (-2)=1101 ,(-7)=1000
- Complément à 2 : (7)=0111, (-2)=1110 ,(-7)=1001

Question 10 (1 pt)**.** Effectuer l'opération $(0,14)_8 + (0,014)_8$ sur la machine suivante :

Signe mantisse	Exposant biaisé (décalé)	Mantisse normalisée
1 bit	4 bits	6 bits

Réponse :
$(0.14)_8 = (0.001100)_2 = +0.1100.2^{-2}$
$(0.014)_8 = (0.0000001100)_2 = +0.1100.2^{-5}$
$(0,14)_8 + (0,014)_8 = (0.1100).2^{-2} + (0.1100).2^{-5}$
$(0,14)_8 + (0,014)_8 = +(0.1101100).2^{-2}$

Signe de la mantisse = positif= 0

Mantisse normalisée sur 6 bits = $(110110)_2$ *Exposant*= -2

Biais = $2^{4-1} = 2^3 = 8$

Exposant biaisé = $-2 + 8 = 6 = (0110)_2$

Donc :

Signe	Exposant biaisé (4 bits)	Mantisse normalisée (6 bits)
0	**0110**	**110110**