Concours d'admission de l'École polytechnique et des ENS (X-ENS), France

Session 2026, filière MPI

Informatique B

Durée : 2 heures

Ce que traite le sujet

- Partie I. Détection de cycles
- Partie II. Unification
- Partie III. Représentation relationnelle

Sujet reproduit sans retouche à partir de la page suivante.

Résumé des parties, chapitres de cours liés et autres années :
prepahelp.com/annales/x-ens/mpi/2026/info-b

Prepa Help ne revendique aucun droit sur le sujet, propriété de l'organisateur du concours.

ECOLE POLYTECHNIQUE - ESPCI
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2026

JEUDI 16 AVRIL 2026
16h30 - 18h30

FILIERES MP-PC-PSI
Epreuve n° 8
INFORMATIQUE B

Durée : 2 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Cycles et unification

Contexte. Soit G un graphe orienté dont les arcs sont étiquetés par des entiers naturels. On suppose que :

A0. Les arcs sortants d'un sommet donné sont étiquetés par des entiers consécutifs en partant de 0.

A1. G est acyclique.

Le graphe G est représenté en Python par une variable globale `G`. C'est un dictionnaire dont les clés sont les sommets et les valeurs sont des listes représentant les arcs sortants. L'étiquette d'un arc correspond à son indice dans la liste. En particulier, les étiquettes des arcs sont implicites et l'hypothèse A0 est automatiquement satisfaite. La figure 1 présente un exemple.

Dans ce problème, certaines des fonctions étudiées modifient G en ajoutant de nouveaux arcs. *L'hypothèse A1 est un invariant qui sera maintenu en toute circonstance.*

Organisation du sujet. La première partie étudie finement un parcours en profondeur pour pouvoir vérifier efficacement qu'un nouvel arc ne crée pas de cycles dans G. La deuxième partie étudie les notions de *forme normale* et d'*unification* dans G. Cette partie peut être traitée indépendamment de la première. La troisième partie étudie la notion de forme normale dans le contexte d'une base de données relationnelle.

Rappels de Python. On rappelle ici quelques opérations sur les listes et les dictionnaires de Python, avec leur complexité. Si `a` désigne une liste en Python de longueur n :

— L'expression `len(a)` renvoie la longueur n.

— L'expression `a[i]` désigne le `i`-ième élément de la liste, pour $0 \leq$ `i` $< n$.

— L'instruction `a[i] = v` affecte la valeur `v` au `i`-ième élément de la liste, pour $0 \leq$ `i` $< n$.

— L'instruction `a.append(e)` ajoute l'élément `e` à la fin de la liste `a`.

— L'expression `a.pop()` supprime et renvoie le dernier élément de la liste `a`.

Toutes ces opérations sont de complexité $O(1)$. Par ailleurs, la boucle `for x in a` parcourt tous les éléments de `a`, dans l'ordre des indices croissants. La complexité est $O(n)$.

Si `d` désigne un dictionnaire en Python :

— Le test `(k in d)` renvoie `True` si `k` est une clé du dictionnaire `d`, et `False` sinon.

— L'expression `d[k]` renvoie la valeur associée à la clé `k` dans le dictionnaire `d`, le cas échéant.

— L'instruction `d[k] = v` affecte la valeur `v` à la clé `k` dans le dictionnaire `d`.

Toutes ces opérations sont de complexité $O(1)$.

Partie I. Détection de cycles

On note $s \rightsquigarrow t$ l'existence d'un chemin de s vers t dans G, et $s \not\rightsquigarrow t$ sa négation. Avant d'ajouter un arc de liaison $s \to t$, on souhaite vérifier que $t \not\rightsquigarrow s$. On introduit pour cela des variables globales :

— `époque` et `sortie` sont des dictionnaires associant un entier à chaque sommet de G,

— `présent` et `compteur` sont des entiers.

Ces variables globales sont mises à jour par une fonction `pp`, donnée en figure 2, qui réalise un parcours en profondeur.

En plus de l'invariant A1, on considère les invariants suivants. Ils seront maintenus en toute circonstance, sauf pendant l'exécution de la fonction `pp`.

A2. `époque`$[s] \leqslant$ `présent`, pour tout sommet s de G.

A3. `époque`$[s] \leqslant$ `époque`$[t]$ pour tout arc $s \to t$ de G.

A4. `époque`$[s] =$ `époque`$[t] \Rightarrow$ `sortie`$[s] \geqslant$ `sortie`$[t]$ pour tout arc $s \to t$ de G.

On commence par l'étude de la fonction `pp`.

Question 1. En supposant que `G` est dans l'état de la figure 1, donner, sans justification, l'état des dictionnaires `sortie` et `époque` après exécution du code suivant.

```
for s in G.keys():
    sortie[s] = 0
    époque[s] = 0

compteur = 0
présent = 1
pp('a')
```

Indication : `sortie['a'] = 4` et `sortie['X'] = 0`.

Question 2. On considère l'exécution du code suivant (à l'extérieur de la fonction `pp`) :

```
présent += 1
```

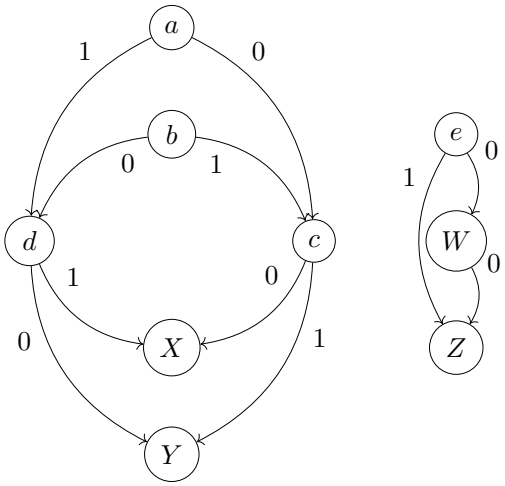
Montrer que, pour tout sommet t de G, on a `époque`$[t] \neq$ `présent` après l'exécution de cette incrémentation.

Question 3. Soit s un sommet de G. On considère l'exécution du code suivant (toujours à l'extérieur de la fonction `pp`) :

```
présent += 1
pp(s)
```

Soit t un sommet de G différent de s. Montrer que $s \rightsquigarrow t$ si et seulement si `époque`$[t] =$ `présent` après l'exécution. On admettra que la fonction `pp(s)` réalise un parcours en profondeur à partir de s et affecte l'époque `présent` à tous les sommets visités.

Question 4. Soit s un sommet de G et soit $u \to v$ un arc de G tel que $s \rightsquigarrow u$. Après un appel à `pp`(s), montrer que `sortie`$[u] \geqslant$ `sortie`$[v]$.




```python
G = {
    'a': ['c', 'd'],
    'b': ['d', 'c'],
    'c': ['X', 'Y'],
    'd': ['Y', 'X'],
    'e': ['W', 'Z'],
    'W': ['Z'],
    'X': [],
    'Y': [],
    'Z': [],
}
```

FIGURE 1 – Exemple de graphe avec sa représentation Python.

```python
# Variables globales.
# L'initialisation (cachée) satisfait les invariants.
G = { ... }
sortie = { ... }
époque = { ... }
présent = 0
compteur = 0

def pp(s):
    """
    Effectue un parcours en profondeur à partir de `s`,
    affecte l'époque `présent` à tous les sommets visités,
    ainsi qu'une certaine numérotation dans `sortie`.
    """
    global compteur    # époque, sortie et présent sont implicitement globales
    époque[s] = présent
    for t in G[s]:
        if époque[t] != présent:
            pp(t)

    sortie[s] = compteur
    compteur += 1
```

FIGURE 2 – Parcours en profondeur.

En s'appuyant sur la fonction `pp`, on définit les fonctions `accessible` et `lier` (figure 3).

Question 5. Montrer qu'`accessible(s, t)` renvoie `True` si et seulement si $s \rightsquigarrow t$.

Question 6. Montrer qu'`accessible(s, t)` préserve les invariants A2, A3 et A4.

Question 7. Montrer que la fonction `lier(u, s)` préserve les invariants A1, A2, A3 et A4.

Partie II. Unification

On suppose désormais que :

A5. Le degré sortant des sommets de G est au plus 2.

Les sommets de degré 0 sont appelés *sommets libres*. Les sommets de degré 1 sont appelés *sommets liés*. Les sommets de degré 2 sont appelés *sommets internes*.

Si s est un sommet lié, on note s^* la cible de l'unique arc sortant de s. Si s est un sommet interne, on note s_0 la cible de l'arc sortant de s étiqueté 0, et s_1 la cible de l'arc sortant de s étiqueté 1.

Question 8. Dans le graphe donné en figure 1, quels sont les sommets internes ? les sommets liés ? les sommets libres ?

On définit de manière récursive la *forme normale* d'un sommet s, notée $\mathrm{fn}(s)$, par :

$$\mathrm{fn}(s) = \begin{cases} s & \text{si } s \text{ est libre,} \\ \mathrm{fn}(s^*) & \text{si } s \text{ est lié,} \\ (\mathrm{fn}(s_0), \mathrm{fn}(s_1)) & \text{si } s \text{ est interne.} \end{cases}$$

Dans l'exemple de la figure 1, $\mathrm{fn}(a) = ((X, Y), (Y, X))$ et $\mathrm{fn}(e) = (Z, Z)$. En Python, la définition se traduit comme suit.

```
def fn(s):
    if len(G[s]) == 0:    # sommet libre
        return s
    elif len(G[s]) == 1:  # sommet lié
        return fn(G[s][0])
    else:                 # sommet interne
        return (fn(G[s][0]), fn(G[s][1]))
```

Question 9. Démontrer la terminaison de `fn` en exhibant une quantité $N(s)$ qui décroit strictement lors des appels récursifs.

Question 10. Montrer que la complexité dans le cas le pire de la fonction `fn` est au moins exponentielle en le nombre de sommets de G.

```python
def accessible(s, t):
    """Renvoie True si, et seulement si, il existe un chemin de s à t."""
    global présent, compteur

    if (époque[s] > époque[t] or
            (époque[s] == époque[t] and sortie[s] < sortie[t])):
        return False

    présent += 1
    pp(s)

    return époque[t] == présent

def lier(u, s):
    """
    Ajoute l'arc u -> s s'il ne crée pas de cycle.
    Signale l'ajout réussi en renvoyant True.
    """
    if not accessible(s, u):
        G[u].append(s)
        return True
    return False
```

FIGURE 3 – Ajout d'arcs de liaison avec détection de cycle.

Question 11. Écrire une fonction Python `fn_efficace(s)` qui calcule la même chose que `fn(s)`, mais en complexité linéaire par rapport au nombre de sommets de G.

Soit $T(s)$ le nombre de chemins dans G partant d'un sommet s et aboutissant à un sommet libre. Par exemple, dans la figure 1, on vérifie que $T(a) = 4$ et $T(W) = 1$.

Question 12. Écrire une fonction Python `taille(s)` prenant en argument un sommet s et renvoyant $T(s)$.

Une *extension de* G est un graphe H acyclique obtenu par ajout, pour certains sommets libres distincts de G, d'un arc sortant étiqueté 0 (ces sommets deviennent donc liés dans H). En particulier, une extension satisfait bien les invariants A0, A1 et A5.

Un *problème d'unification* est un ensemble P de paires de sommets de G. Une *solution* d'un problème d'unification P est une extension de G telle que $\mathrm{fn}(s) = \mathrm{fn}(t)$ dans cette extension, pour tout $(s, t) \in P$.

Par exemple, dans le contexte de la figure 1, le problème d'unification $\{(a, b)\}$ admet comme solution l'extension de G obtenue par ajout de l'arc $X \to Y$.

Question 13. Dans le contexte de la figure 1, montrer que le problème d'unification $\{(c, X)\}$ n'a pas de solution.

Soit P un problème d'unification, (s, t) un élément de P, et $P' = P \setminus \{(s, t)\}$ l'ensemble obtenu en retirant (s, t) de P. On admet que :

T1. $P' \cup \{(t, s)\}$ a les mêmes solutions que P.

T2. Si s est lié, alors $P' \cup \{(s^*, t)\}$ a les mêmes solutions que P.

T3. Si s et t sont internes, alors $P' \cup \{(s_0, t_0), (s_1, t_1)\}$ a les mêmes solutions que P.

T4. Si s est libre, t n'est pas lié et $s \neq t$, alors

$$P \text{ admet une solution } \Leftrightarrow \ P' \text{ admet une solution avec l'arc } s \to t.$$

En s'appuyant sur ces équivalences, on propose la fonction `unif(P)` (figure 4). Elle prend en argument un problème d'unification P (comme une liste de paires de sommets) et renvoie `True` si et seulement si G admet une extension solution de P. De plus, si `unif(P)` renvoie `True`, alors la variable globale `G` contient désormais une solution de P. Autrement dit, les arcs d'une solution sont ajoutés directement à G. Par exemple, dans le contexte de la figure 1, `unif([('X', 'Y')])` renvoie `True` et ajoute l'arc $X \to Y$ (ou $Y \to X$).

Question 14. Compléter la partie manquante de `unif` (à l'endroit marqué `à compléter`) en insérant une ou plusieurs lignes de code.

On pourra utiliser la fonction `lier` et s'appuyer sur le résultat obtenu en première partie : la fonction `lier(u, s)` ajoute un arc $u \to s$ dans G si et seulement si l'ajout de cet arc ne crée pas de cycle.

```python
def unif(P):
    """
    Ajoute des arcs dans G jusqu'à obtenir une extension solution de P.
    Renvoie False si P n'a pas de solution. Renvoie True sinon.
    """
    while len(P) > 0:
        s, t = P.pop()

        if len(G[s]) == 1:
            P.append((G[s][0], t))
        elif len(G[t]) == 1:
            P.append((s, G[t][0]))
        elif len(G[s]) == 2 and len(G[t]) == 2:
            P.append((G[s][0], G[t][0]))
            P.append((G[s][1], G[t][1]))
        elif len(G[s]) == 0:
            ...  # à compléter
        else:
            P.append((t, s))

    return True
```

FIGURE 4 – Fonction `unif`.

Partie III. Représentation relationnelle

On suppose maintenant que le graphe G est représenté dans une base de données relationnelle avec le schéma suivant :

- une table *sommets* dont les colonnes sont
  - *id*, clé primaire entière,
  - *degré*, entier, représente le degré sortant,
  - *sortie*, entier,
  - *époque*, entier ;
- une table *arcs* dont les colonnes sont
  - *étiquette*, entier,
  - *source*, clé étrangère vers *sommets*,
  - *cible*, clé étrangère vers *sommets*.

Question 15. Écrire une requête SQL permettant de vérifier les invariants A3 et A4, introduits dans la partie I.

Question 16. Écrire une requête SQL renvoyant, s'il en existe, une paire de sommets (s, t) telle que $s \neq t$ et $\mathrm{fn}(s) = \mathrm{fn}(t)$. (La fonction *forme normale* « fn » est définie dans la partie II.)

Fin du sujet.

Travailler les chapitres de ce sujet

Prepa Help suit le programme de MPI chapitre par chapitre : cours détaillé, méthodes, pièges, exercices et problèmes corrigés pas à pas, tirés des concours. L'essai gratuit donne accès aux chapitres offerts de ta filière, sans carte bancaire.

Chapitres liés à ce sujet

- Spécifier, tester, prouver : terminaison et correction
- Récursivité
- Tuples, dictionnaires et ensembles
- Algorithmes, discipline de programmation, validation et tests

Ouvrir l'essai gratuit

Scanne le code ou clique sur le bouton.

prepahelp.com/#essai

Une question sur le concours ou sur ta filière :
écrire sur WhatsApp

Les autres années et épreuves : prepahelp.com/annales/x-ens/mpi