

CYCLES ET UNIFICATION

I Détection de cycles

Q1 le code proposé commence par affecter 0 à chacun des sommets du graphe des dictionnaires dictionnaire `sortie` et `époque`, puis respectivement 0 et 1 aux variables `compteur` et `présent`.

Après l'exécution de `pp('a')` :

— Le code proposé en FIGURE 2 donne la valeur 1 aux sommets visités par le parcours en profondeur à partir de 'a'. Après exécution du code, l'état du dictionnaire `époque` est donc :

{ 'a': 1, 'b': 0, 'c': 1, 'd': 1, 'e': 0, 'W': 0, 'X': 1, 'Y': 1, 'Z': 0 }

— Le code proposé en FIGURE 2 indique dans quel ordre s'achèvent les traitements `pp(s)` où `s` sont les sommets vus lors du parcours en profondeur. Le premier traitement à s'achever (ici `pp('X')`) prend le numéro 0. Après exécution du code, l'état du dictionnaire `sortie` vaut donc :

{ 'a': 4, 'b': 0, 'c': 2, 'd': 3, 'e': 0, 'W': 0, 'X': 0, 'Y': 1, 'Z': 0 }

Q2 Au départ, on a $\text{époque}[t] \leq \text{présent}$ pour tout t (**invariant A2**).

Après `présent += 1`, la valeur de `présent` est donc augmentée de 1.

Avec cette nouvelle valeur, on a $\text{époque}[t] < \text{présent}$

et, a fortiori, $\text{époque}[t] \neq \text{présent}$ pour tout sommet t de G .

Q3 Soit t un sommet différent de s .

Après l'exécution des instructions `présent += 1`; `pp(s)`, le parcours en profondeur affecte la valeur de `présent` à chaque sommet t atteignable depuis s . Ainsi, si $s \rightsquigarrow t$ alors $\text{époque}[t] = \text{présent}$.

Réciproquement, d'après la question 2, juste avant l'appel de `pp(s)`, on a $\text{époque}[t] \neq \text{présent}$. Ainsi, si $\text{époque}[t] = \text{présent}$ après l'exécution de `pp(s)`, il s'agit forcément d'une mise à jour effectuée par `pp(s)`. Ainsi, le sommet t est atteignable depuis s , i.e. $s \rightsquigarrow t$. Finalement, **pour tout sommet t différent de s , on a $s \rightsquigarrow t$ si et seulement si $\text{époque}[t] = \text{présent}$.**

Q4 Soit s un sommet de G et soit $u \rightarrow v$ un arc de G tel que $s \rightsquigarrow u$. Supposons l'exécution de `pp(s)`.

Comme $s \rightsquigarrow u$, le sommet u est atteignable depuis s et donc `sortie[u]` va être mis à jour suite à l'appel de `pp(s)`, à la fin de l'exécution de `pp(u)`.

En particulier, lors de l'exécution de `pp(u)`, on rencontre v lors du parcours de $G[u]$ (car $u \rightarrow v$) et tant que l'appel `pp(v)` n'a pas eu lieu, on a $\text{époque}[u] < \text{présent}$ (cf. question 2) :

— soit $\text{époque}[v] \neq \text{présent}$, ce qui signifie que l'on rencontre le sommet v pour la première fois.

Alors `pp(v)` va s'exécuter. Suite au parcours de tous les éléments de $G[v]$, on met à jour `sortie[v]` avec la valeur de `compteur` et cette valeur est ensuite incrémentée. On revient alors dans `pp(u)` qui n'a pas fini de s'exécuter. Suite au parcours des éléments de $G[u]$ qui suivent v (éventuellement aucun), on met à jour `sortie[u]` avec la valeur de `compteur` qui aura été a minima incrémenté 1 fois depuis la mise à jour de `sortie[v]`.

Ainsi, forcément, $\text{sortie}[u] \geq \text{sortie}[v]$.

- soit $\text{époque}[v] = \text{présent}$, ce qui signifie que le sommet v a déjà visité au moment où $\text{pp}(u)$ traite l'arc $u \rightarrow v$. Comme $\text{époque}[u] \neq \text{époque}[v]$, on en déduit que $u \neq v$. D'après la question 3, on sait qu'il existe au moins un chemin $s \rightsquigarrow v$: soit il en existe au moins un qui ne passe pas par u , soit il s'agit du chemin $s \rightsquigarrow u \rightarrow v$.
- S'il existe un chemin $s \rightsquigarrow v$ qui ne passe pas par u , comme $\text{pp}(s)$ est le seul traitement à mettre à jour époque et que $\text{époque}[u] < \text{présent} = \text{époque}[v]$, c'est que $\text{pp}(v)$ a été déclenché depuis $\text{pp}(s)$ avant le début de $\text{pp}(u)$. Alors, $\text{sortie}[v]$ a été mis à jour avant que $\text{pp}(u)$ ne commence, donc avant $\text{sortie}[u]$.
- Soit il s'agit du chemin $s \rightsquigarrow u \rightarrow v$.
 - Notons que le sous-cas où $\text{pp}(v)$ est en cours d'exécution (non terminé) quand $\text{pp}(u)$ traite $u \rightarrow v$ est impossible. En effet, si $\text{pp}(v)$ était en cours, v serait un ancêtre de u dans la pile d'appels, donc $v \rightsquigarrow u$, et l'arc $u \rightarrow v$ formerait un cycle, contredisant A1.
 - On en conclut que l'exécution de $\text{pp}(v)$ a été déclenché par un appel récursif à l'intérieur de $\text{pp}(u)$, via un autre successeur w de u traité avant v dans la boucle (i.e. w apparaît avant v dans $G[u]$, et le parcours de w a conduit à visiter v). Alors, $\text{sortie}[v]$ a été affecté pendant l'exécution de $\text{pp}(u)$, mais avant que $\text{pp}(u)$ n'atteigne ses lignes 21-22.

Dans les deux sous-cas possibles, $\text{sortie}[v]$ reçoit une valeur de **compteur** strictement inférieure à celle utilisée pour $\text{sortie}[u]$ (car **compteur** est strictement croissant et $\text{sortie}[u]$ est affecté après $\text{sortie}[v]$).

On conclut que, si s est un sommet de G et $u \rightarrow v$ un arc de G tel que $s \rightsquigarrow u$, on a toujours $\text{sortie}[u] \geq \text{sortie}[v]$.

Q5 Lemme : Les invariants A3 et A4 se généralisent à tout **chemin** $s \rightsquigarrow t$ de G .

Preuve du lemme :

- Preuve pour A3 :
Soit un chemin $s = u_0 \rightarrow u_1 \rightarrow \dots \rightarrow u_p = t$ (avec $p \geq 1$).
Par une récurrence immédiate, il s'ensuit de A3 que la fonction **époque** est croissante le long des arcs du chemins.
Ainsi $\text{époque}[s] \leq \text{époque}[t]$ pour tout $s \rightsquigarrow t$ de G .
- Preuve pour A4 :
Soit un chemin $s = u_0 \rightarrow u_1 \rightarrow \dots \rightarrow u_p = t$ (avec $p \geq 1$).
Si $\text{époque}[s] = \text{époque}[t]$, alors, comme la fonction **époque** est croissante le long des arcs du chemin, c'est donc qu'elle est constante sur chaque arc, càd.
 $\text{époque}[u_i] = \text{époque}[u_{i+1}]$ pour tout $0 \leq i \leq p-1$.
Par une récurrence immédiate, il s'ensuit de A4 que la fonction **sortie** est décroissante le long des arcs du chemins.
Ainsi, $\text{époque}[s] = \text{époque}[t] \implies \text{sortie}[s] \geq \text{sortie}[t]$ pour tout $s \rightsquigarrow t$ de G .

Montrons à présent qu' $\text{accessible}(s, t)$ renvoie **True** si et seulement si $s \rightsquigarrow t$.

- On suppose qu' $\text{accessible}(s, t)$ renvoie **True**.
D'une part, ceci suppose qu' $\text{accessible}(s, t)$ ne renvoie pas **False** aux lignes 5 à 7.
Or, d'après le lemme, on sait que, lorsque A3 ou A4 n'est pas vérifié, alors il ne peut pas y avoir de chemin $s \rightsquigarrow t$ (et donc s n'est pas « avant » t selon un ordre topologique croissant donné par $(\text{époque}, -\text{sortie})$).
D'autre part, après incrémentation de **présent** et exécution de $\text{pp}(s)$, comme $\text{accessible}(s, t)$ renvoie **True**, il s'ensuit que $\text{époque}[t] = \text{présent}$.
Si $s \neq t$, d'après la question 3, il existe donc un chemin $s \rightsquigarrow t$.

Notons que, si $s = t$, `accessible(s,t)` renvoie toujours `True`. En effet, dans ce cas, `époque[s] = époque[t]` et `sortie[s] = sortie[t]`, donc les lignes 5–6 ne renvoient pas `False`; puis, après exécution de `pp(s)`, on a `époque[s] = présent`, i.e. `époque[t] = présent`. Ceci est cohérent avec le fait que l'on a toujours un chemin $s \rightsquigarrow s$ (chemin vide).

- Réciproquement, on suppose l'existence d'un chemin $s \rightsquigarrow t$.

Notons que l'on peut supposer $s \neq t$, car on a vu qu'en cas d'égalité, `accessible(s,t)` renvoie toujours `True`.

D'après le lemme, on sait que A3 et A4 sont donc vérifiés. Ainsi les lignes 5 à 7 de la `accessible(s, t)` ne renvoient pas `False`.

Ensuite, la fonction `accessible` incrémente `présent` et exécute `pp(s)`. Comme $s \neq t$, d'après la question 3, comme $s \rightsquigarrow t$, après exécution, on a `époque[t] = présent`. Ainsi, `return époque[t] == présent` renvoie `True`.

Ainsi, `accessible(s, t)` renvoie `True`.

Finalement, `accessible(s,t)` renvoie `True` si et seulement si $s \rightsquigarrow t$.

Q6 Montrons que la fonction `accessible` préserve les invariants A2, A3 et A4 :

- Tout d'abord, les lignes 5 à 7, ne modifient ni `époque`, ni `présent`, donc A2 est préservé.

Ensuite, les lignes 5 à 7 vérifient que A3 et A4 sont vérifiés pour tout chemin $s \rightsquigarrow t$. Si tel est le cas, ce sera forcément vrai pour tout arc $s \rightarrow t$.

Avant l'exécution des lignes 9 à 12 (incrément de `présent`, exécution de `pp(s)`, puis test `return époque[t] == présent`), les invariants A2, A3 et A4 sont donc vérifiés.

- Invariant A2 :

Après l'exécution de `pp(s)`, seuls les sommets t pour lesquels il existe un chemin $s \rightsquigarrow t$ voient `époque[t]` incrémenté via `époque[t] = présent`. Donc, en fin d'exécution de `pp(s)`, A2 est bien vérifié. Le test `return époque[t] == présent` n'a ensuite aucun impact sur A2.

Finalement, `accessible(s, t)` préserve l'invariant A2.

- Invariant A3 :

Comme A3 est vérifié avant l'exécution de `pp(s)`, on a `époque[u] ≤ époque[v]` pour tout arc $u \rightarrow v$. Considérons l'un de ces arcs $u \rightarrow v$:

- Soit u et v sont accessibles depuis s . L'exécution de `pp(s)` affecte à `époque[u]` et `époque[v]` le même contenu, à savoir celui de `présent` incrémenté avant l'exécution de `pp(s)`. Ainsi, l'invariant A3 est préservé.

- Soit u n'est pas accessible depuis s . Alors, `époque[u]` n'est pas modifié et `époque[v]` est soit inchangé (v non accessible), soit mis à `présent` (v accessible). Si `époque[v]` est inchangé, alors A3 est préservé.

Sinon, `époque[u] ≤ présent` (cf. A2) avant l'appel de `pp(s)`. Avant cet appel, `présent` est incrémenté, donc la mise à jour de `époque[v]` avec la valeur incrémentée de `présent` donne `époque[u] < époque[v]`. Ainsi, l'invariant A3 est préservé.

- Notons que le cas où u est accessible depuis s et v ne l'est pas n'existe pas car $s \rightsquigarrow u \rightarrow v$ est un chemin de s vers v .

Le test `return époque[t] == présent` n'a ensuite aucun impact sur A3.

Finalement, `accessible(s, t)` préserve l'invariant A3.

— Invariant A4 :

Comme A4 est vérifié avant l'exécution de `pp(s)`, on a `époque[u] = époque[v]`
 $\implies \text{sortie}[u] \geq \text{sortie}[v]$ pour tout arc $u \rightarrow v$.

Soit $u \rightarrow v$ un arc de G . On a les cas suivants :

— Soit il existe un chemin $s \rightsquigarrow u$.

D'une part, comme $s \rightsquigarrow u \rightarrow v$, après l'exécution de `pp(s)`, on a `époque[u] = présent = époque[v]` car u et v sont sur un chemin visité par `pp(s)`.

D'autre part, d'après la question 4, après l'exécution de `pp(s)`, on a toujours `sortie[u] ≥ sortie[v]`. On en conclut que l'invariant A4 est préservé.

— Soit il n'existe **pas** de chemin $s \rightsquigarrow u$ (ce qui entraîne $s \neq u$).

D'après la question 3, on a alors `époque[u] ≠ présent` après l'exécution de `pp(s)`.

— Si $v = s$, alors après l'exécution de `pp(s)`, on a `époque[s] = présent`.
 Or `époque[u] ≠ présent`. On n'est donc jamais dans un cas d'égalité `époque[u] = époque[v]`.

— Si $v \neq s$ et `époque[u] = époque[v]` après l'exécution de `pp(s)`, d'après la question 3, cela signifie qu'il n'existe pas de chemin $s \rightsquigarrow v$. Ainsi, `pp(s)` n'a pas appelé `pp(u)` et `pp(v)`. Donc, ni `sortie[u]`, ni `sortie[v]` n'a été modifié. Donc, l'invariant A4 est préservé.

Le test `return époque[t] == présent` n'a ensuite aucun impact sur A4.

Finalement, `accessible(s, t)` préserve l'invariant A4.

Q7 Montrons que la fonction `lier(u, s)` préserve les invariants A1, A2, A3 et A4 :— Invariant A1 :

On a deux cas :

— soit u est accessible de s .

L'appel `lier(u, s)` renvoie directement `False` et n'ajoute **pas** l'arc $u \rightarrow s$ à G .
 En effet, l'ajout de $u \rightarrow s$ serait un chemin $u \rightarrow s \rightsquigarrow u$, donc un cycle.

— soit u n'est **pas** accessible de s .

L'appel `lier(u, s)` ajoute s à la liste d'adjacence de u et renvoie `True`.

En effet, l'ajout de $u \rightarrow s$ ne crée pas de cycle, sinon il existerait déjà un chemin $s \rightsquigarrow u$, contredisant `not accessible(s, u)`.

Si le graphe était acyclique au départ, il reste donc acyclique.

Ainsi, `lier(u, s)` préserve l'invariant A1.

— Invariants A2 et A3 :

Reprenons les deux cas possibles :

— soit u est accessible de s .

Comme aucun arc est ajouté à G (car A1 ne serait plus préservé), on sait de la question 6 que A2 et A3 sont préservés.

Ainsi, `lier(u, s)` préserve les invariants A2 et A3.

— soit u n'est **pas** accessible de s .

Nommons présent^- et présent^+ les valeurs de `présent` respectivement avant et après l'exécution de `accessible(s, u)`. D'après la ligne 9 de la fonction `accessible(s, u)`, on a $\text{présent}^- < \text{présent}^+$.

Avant l'exécution de `accessible(s, u)`, on a l'invariant A2, et donc `époque[s] ≤ présent-` et `époque[u] ≤ présent-`.

Après l'exécution de `accessible(s, u)`, la variable `présent` vaut présent^+ et

donc $\text{époque}[s] = \text{présent}^+$, alors que $\text{époque}[u] = \text{présent}^-$ (car u est non accessible de s). De plus, l'ajout de l'arête $u \rightarrow s$ par un `append` n'influence ni $\text{époque}[u]$, ni $\text{époque}[s]$.

Ainsi, après exécution de `lien(u, s)` et ajout de l'arc $u \rightarrow s$ à G :

- d'une part $\text{époque}[s] = \text{présent}^+ = \text{présent}$ et $\text{époque}[u] = \text{présent}^- < \text{présent}^+ = \text{présent}$, càd. l'invariant A2 est préservé pour l'arc $u \rightarrow s$.
- d'autre part, $\text{époque}[u] = \text{présent}^- < \text{présent}^+ = \text{époque}[s]$, càd. l'invariant A3 est préservé pour l'arc $u \rightarrow s$.

Par ailleurs, l'invariant A3 est préservé par `accessible(s, u)` pour tous les autres arcs de G . Dans ce cas, `lier(u, s)` préserve aussi les invariants A2 et A3.

Ainsi, `lier(u, s)` préserve les invariants A2 et A3.

— Invariant A4 :

Si u est accessible de s , aucun arc n'est ajouté à G , donc A4 est préservé d'après la question 6.

Si u n'est **pas** accessible de s , on vient de montrer qu'après ajout de l'arc $u \rightarrow s$ à G , on a $\text{époque}[u] < \text{époque}[s]$. Donc A4 est vérifié par défaut pour cet arc et préservé par `accessible(s, u)` pour tous les autres arcs de G .

Ainsi, `lier(u, s)` préserve l'invariant A4.

II Unification

Q8 D'après la définition donnée,

- les **sommets internes** de la FIGURE 1 sont ceux de degrés sortants égal à 2, donc 'a', 'b', 'c', 'd', 'e',
- les **sommets liés** de la FIGURE 1 sont ceux de degrés sortants égal à 1, donc 'W',
- les **sommets libres** de la FIGURE 1 sont ceux de degrés sortants égal à 0, donc 'X', 'Y', 'Z'.

Q9 On considère $N(s)$ égale au **nombre de sommets t accessibles depuis s** , i.e. les sommets $t \in G$ tels que la fonction `accessible` de la question 5 renvoie **True**. Ainsi, $N(s) = \text{card}\{t \in G, \text{accessible}(s, t)\}$.

Montrons la terminaison de l'exécution de la fonction `fn(s)` en utilisant la quantité $N(s)$.

Il y a 3 possibilités :

- Soit s est un sommet libre.
 s est le seul sommet accessible depuis s (chemin vide) et $N(s) = 1$.
- Soit s est un sommet lié.
Alors, comme s est de degré 1, on obtient $N(s) = 1 + N(s^*)$, i.e. $N(s^*) = N(s) - 1$. Or `fn(s)` exécute `fn(G[s][0])`, i.e. `fn(s*)`. Ainsi, lors de l'appel récursif de `fn(s*)`, la quantité $N(s)$ décroît strictement pour le sommet appelé s^* .
- Soit s est un sommet interne.
Alors, comme s est de degré 2, on a $N(s_0) < N(s)$ et $N(s_1) < N(s)$. En effet, s_0 et s_1 sont accessibles depuis s , mais s ne l'est pas depuis s_0 ou s_1 car le graphe est **acyclique**.
Or, `fn(s)` exécute d'abord `fn(G[s][0])`, i.e. `fn(s_0)`, donc la quantité $N(s)$ décroît strictement lors de cet appel récursif.

Plus tard, la fonction $\text{fn}(s)$ exécute $\text{fn}(G[s][1])$, i.e. $\text{fn}(s_1)$, donc la quantité $N(s)$ décroît également strictement lors de cet appel récursif.

On conclut que les quantités $N(s)$ forment une suite d'entiers naturels strictement décroissante le long des appels récursifs. La quantité exhibée permet de conclure à la terminaison de l'exécution de $\text{fn}(s)$.

Q10 Version 1 : le graphe G est une suite de sommets **internes**

$s = u_0 \rightarrow u_1 \rightarrow \dots \rightarrow u_p = t$, à l'exception de t qui est libre.

Dans ce cas, pour tout $0 \leq i \leq p-1$, on a $(u_i)_0 = (u_i)_1 = u_{i+1}$. Le calcul $\text{fn}((u_i)_0)$ est donc **dupliqué**.

Ainsi, si $T(s)$ est le nombre d'appels de $\text{fn}(s)$, on a :

$$— T(t) = 1$$

$$— T(u_i) = 2 \times T(u_{i+1})$$

Donc, $T(s) = T(u_0) = 2^p = 2^{n-1} = \frac{1}{2} \times 2^n$ où n est le nombre de sommets de G .

Dans ce cas, le nombre d'appels est proportionnel à 2^n et la complexité de fn est donc exponentielle en le nombre n de sommets de G .

Ainsi, dans le pire des cas, la complexité de fn est au moins exponentielle en le nombre de sommets de G .

Version 2 : le graphe G est une suite de sommets définis comme suit :

u_0 et v_0 libres et pour tout $1 \leq i \leq p$, u_i a pour enfants u_{i-1} et v_{i-1} ; de même v_i a pour enfants u_{i-1} et v_{i-1} (*On partage donc tous les sous-graphes*)

Ainsi, si $T(s)$ est le nombre d'appels de $\text{fn}(s)$, on a :

$$— T(u_0) = T(v_0) = 1$$

$$— T(u_i) = T(v_i) = T(u_{i-1}) + T(v_{i-1})$$

Posons, pour tout $1 \leq i \leq p$, $a_i = T(u_i) = T(v_i)$ avec $a_0 = T(u_0) = T(v_0) = 1$.

On a donc $a_0 = 1$ et pour tout $1 \leq i \leq p$, $a_i = 2 \times a_{i-1}$. Donc, $a_i = T(u_i) = T(v_i) = 2^i$.

Or, si le graphe contient les sommets u_0 à u_p et v_0 à v_p , il contient $n = 2(p+1)$ sommets. Ainsi, $T(u_p) = 2^{\frac{n}{2}-1} = \frac{1}{2} \times \sqrt{2}^n$ où n est le nombre de sommets de G .

Dans ce cas, le nombre d'appels est proportionnel à $\sqrt{2}^n$ et la complexité de fn est donc exponentielle en le nombre n de sommets de G .

Ainsi, dans le pire des cas, la complexité de fn est au moins exponentielle en le nombre de sommets de G .

Q11 On peut obtenir **une complexité linéaire par rapport au nombre de sommets par mémoïsation**.

Soit `memo` un dictionnaire associant à chaque sommet s la valeur $\text{fn}(s)$.

La variable `memo` est définie comme variable globale et au lancement de `fn`, on a `memo = {}`.

On peut écrire alors le code de la fonction `fn_efficace` en **programmation dynamique descendante**.

```

1 def fn_efficace(s):
2     if s in memo:
3         return memo[s]
4     if len(G[s]) == 0:           # sommet libre, feuille de l'arbre
5         m = s
6     elif len(G[s]) == 1:        # sommet lié
7         m = fn_efficace(G[s][0])
8     else:                       # sommet interne
9         m = (fn_efficace(G[s][0]), fn_efficace(G[s][1]))
10    memo[s] = m
11    return m

```

Q12 On a la relation de récurrence suivante :

- quand s est un sommet libre, $T(s) = 1$,
- quand s est un sommet lié, $T(s) = T(s^*)$,
- quand s est un sommet interne, $T(s) = T(s_0) + T(s_1)$.

On en déduit une version en programmation dynamique descendante de la fonction `taille(s)` qui renvoie $T(s)$. On suppose défini comme variable globale initialement vide, le dictionnaire `memo_taille` associant à chaque sommet s la valeur $T(s)$:

```

1 def taille(s):
2     if s in memo_taille:
3         return memo_taille[s]
4     if len(G[s]) == 0:      # sommet libre, feuille de l'arbre
5         t = 1
6     elif len(G[s]) == 1:   # sommet lié
7         t = taille(G[s][0])
8     else:                  # sommet interne
9         t = taille(G[s][0]) + taille(G[s][1])
10    memo_taille[s] = t
11    return t

```

Q13 Dans G , on a $\text{fn}('c') = ('X', 'Y')$ et $\text{fn}('X') = 'X'$.

Les sommets libres de G sont $'X'$, $'Y'$, $'Z'$.

- Envisageons les différents cas d'ajout d'arcs sortants de $'X'$ avec les sommets de G .
Tout d'abord, seuls les sommets $'e'$, $'W'$, $'Y'$, $'Z'$ peuvent être ajoutés à $'X'$ en conservant les propriétés A0, A1 et A5.

En ajoutant $X \rightarrow e$, on obtient $\text{fn}('c') = (('Z', 'Z'), 'Y')$ et $\text{fn}('X') = ('Z', 'Z')$.

En ajoutant $X \rightarrow W$, on obtient $\text{fn}('c') = ('Z', 'Y')$ et $\text{fn}('X') = 'Z'$.

En ajoutant $X \rightarrow Y$, on obtient $\text{fn}('c') = ('Y', 'Y')$ et $\text{fn}('X') = 'Y'$.

En ajoutant $X \rightarrow Z$, on obtient $\text{fn}('c') = ('Z', 'Y')$ et $\text{fn}('X') = 'Z'$.

- Envisageons les différents cas d'ajout d'arcs sortants de $'Y'$ ou $'Z'$ avec les sommets de G .

On constate que $'X'$ reste un sommet libre donc $\text{fn}('X') = 'X'$. De son côté, $\text{fn}('c')$ est de type `tuple`. Par conséquent, on a toujours $\text{fn}('c') \neq \text{fn}('X')$.

Pour toute extension de G , on a $\text{fn}('c') \neq \text{fn}('X')$, donc le problème d'unification $\{(c, X)\}$ n'a pas de solution.

Q14 On complète la fonction `unif` :

```

1  def unif(P):
2      while len(P) > 0:
3          s, t = P.pop()
4
5          if len(G[s]) == 1: # T2 appliqué à s
6              P.append((G[s][0], t))
7          elif len(G[t]) == 1: # T2 appliqué à t
8              P.append((s, G[t][0]))
9          elif len(G[s]) == 2 and len(G[t]) == 2: # T3
10             P.append((G[s][0], G[t][0]))
11             P.append((G[s][1], G[t][1]))
12          elif len(G[s]) == 0: # T4
13             if len(G[t]) != 1 and s != t: # t n'est pas lié et s est
différent de t
14                 if not lier(s, t):
15                     return False
16             else: # l'arc s->t est ajouté à G quand il n'engendre
pas de cycle
17                 P.append((s, t))
18          else:
19             P.append((t, s)) # T1
20
21     return True

```

III Représentation rationnelle

Q15 La requête SQL suivante ne renvoie pas de résultat si et seulement si les invariants A3 et A4 sont vérifiés. Dans le cas contraire, elle renvoie les arcs concernés.

En effet, on détermine s'il existe des arcs ne vérifiant pas les invariants A3 et A4. Il n'y en a pas si et seulement si tous les arcs vérifient A3 et A4.

```

SELECT source, cible
FROM arcs
JOIN sommets AS S1 ON S1.id = source
JOIN sommets AS S2 ON S2.id = cible
WHERE (S1.époque > S2.époque) OR
      (S1.époque = S2.époque AND S1.sortie < S2.sortie)

```

Q16 On suppose la fonction normale `fn` déjà écrite car l'écriture récursive en SQL n'est pas au programme.

On détermine toutes les paires de sommets (s, t) tels que $s \neq t$ et $\text{fn}(s) = \text{fn}(t)$.

Si l'on souhaite une seule paire de sommets, il suffit de rajouter `LIMIT 1` en fin de requête.

```

SELECT S1.id AS s, S2.id AS t
FROM sommets AS S1
JOIN sommets AS S2 ON S1.id < S2.id
WHERE fn(S1.id) = fn(S2.id)

```