TD XXXIX

CLÔTURE TRANSITIVE

Ce sujet est directement adapté d'un problème posé au concours Mines-Ponts en 2014 (reformulé de manière parfaitement équivalente en termes de graphes, et traduit en C.)

On considère un graphe orienté $G = (V, E)$, et l'on note $n = |V|$ son nombre de sommets. On se fixe une numérotation $x_0, \dots, x_{n-1}$ des sommets.

- Pour $x, y \in V$, on note $x \Rightarrow_k y$ s'il existe $k+1$ sommets $y_0, \dots, y_k$ tels que :
  - $y_0 = x$;
  - $y_k = y$;
  - $(y_i, y_{i+1}) \in E$ pour tout $i \in [0 \dots k-1]$.

  On a donc $x \Rightarrow_0 y$ si et seulement si $x = y$.
- On suppose que pour tout sommet x, la *boucle* (x, x) fait partie de l'ensemble E des arcs. On a donc également $x \Rightarrow_1 x$ pour tout sommet x.
- On note $x \Rightarrow y$ s'il existe un entier k tel que $x \Rightarrow_k y$.

Exemples On se réfèrera aux deux graphes ci-dessous dans le problème :

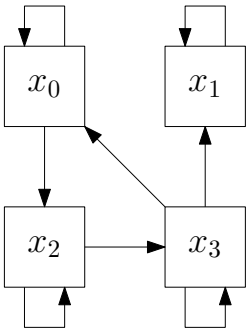

FIGURE XXXIX.1 – Le graphe G_1.

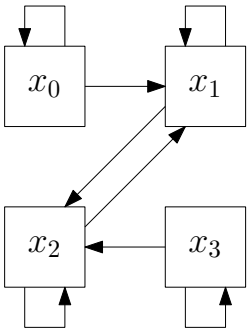

FIGURE XXXIX.2 – Le graphe G_2.

► **Question 1** Soient $x, y \in V$ et $h \leqslant k$ deux entiers naturels. Montrer que si $x \Rightarrow_h y$, alors $x \Rightarrow_k y$.

► **Question 2** Soient $x, y \in V$. Montrer que l'on a $x \Rightarrow y$ si et seulement si $x \Rightarrow_{n-1} y$.

Matrice booléenne Une matrice booléenne est une matrice dont les coefficients sont à valeurs dans $\{\mathtt{vrai}, \mathtt{faux}\}$. Le produit de deux matrices booléennes s'obtient selon la formule habituelle du produit matriciel, en prenant pour somme de deux valeurs booléennes le « ou logique » (noté $\vee$) et pour produit de deux valeurs booléennes le « et logique » (noté $\wedge$). Dans toute la suite, le produit matriciel, et les puissances d'une matrice, sont à comprendre avec cette définition.

On a par exemple :

$$\begin{pmatrix} \mathtt{vrai} & \mathtt{vrai} \\ \mathtt{faux} & \mathtt{vrai} \end{pmatrix} \cdot \begin{pmatrix} \mathtt{faux} & \mathtt{vrai} \\ \mathtt{vrai} & \mathtt{faux} \end{pmatrix} = \begin{pmatrix} \mathtt{vrai} & \mathtt{vrai} \\ \mathtt{vrai} & \mathtt{faux} \end{pmatrix}$$

Programmation En C, une matrice booléenne M de dimensions (n, p) sera représentée par un `M` de type `bool**`. `M` pointera vers un bloc alloué de n pointeurs de type `bool*` : `M[i]` sera un pointeur vers un bloc alloué de p booléens correspondant à la ligne i de la matrice (les lignes étant numérotées de 0 à $n-1$).

▸ **Question 3** Écrire une fonction `matrix_new` renvoyant une nouvelle matrice de la taille spécifiée, initialisée à `false`.

```
bool **matrix_new(int n, int p);
```

▸ **Question 4** Écrire une fonction `matrix_free` qui libère toute la mémoire associée à une matrice. On pourra supposer que la matrice a été créée par un appel à `matrix_new` (on a ensuite pu modifier ses coefficients, mais pas les pointeurs). Seul le nombre de lignes est pris en paramètre, le nombre de colonnes étant superflu.

```
void matrix_free(bool **m, int n);
```

▸ **Question 5** Écrire une fonction `identity` renvoyant la matrice carrée de taille n contenant des `true` sur la diagonale et des `false` ailleurs.

```
bool **identity(int n);
```

▸ **Question 6** Écrire une fonction `product` renvoyant le produit des deux matrices passées en argument.

```
bool **product(bool **a, bool **b, int n, int p, int q);
```

Préconditions : on pourra supposer (sans le vérifier) que `a` est de dimensions (n, p) et `b` de dimensions (p, q) (et que tous ces entiers sont strictement positifs).

On associe à un graphe G sa matrice d'adjacence, vue comme une matrice booléenne, que l'on notera A. On a donc $A[i, j]$ qui vaut *vrai* si et seulement si $x_i \Rightarrow_1 x_j$ (on rappelle que $x_i \Rightarrow_1 x_i$ pour tout i).

▸ **Question 7** Montrer que pour tous $i, j \in [0 \dots n-1]$ et pour tout $k > 0$, on a $A^k[i, j] = \textit{vrai}$ si et seulement si $x_i \Rightarrow_k x_j$.

▸ **Question 8** Montrer que pour $k \geqslant n-1$, on a $A^k = A^{n-1}$.

Clôture transitive On appelle *clôture transitive* de la matrice A la matrice $\mathrm{CT}(A) = A^{n-1}$.

▸ **Question 9** Écrire une fonction `closure` qui prend en entrée une matrice carrée A de taille n et renvoie sa clôture transivite, calculée à l'aide de multiplications de matrices.

```
bool **closure(bool **a, int n);
```

▸ **Question 10** Déterminer la complexité de la fonction `closure`.

▸ **Question 11** Écrire une fonction `accessible` qui prend en entrée une matrice carrée A de taille n et un entier $i \in [0 \dots n-1]$ et renvoie un pointeur `arr` vers un bloc alloué de n booléens tel que `arr[j]` soit égal à `true` si $x_i \Rightarrow x_j$, `false` sinon.

```
bool *accessible(bool **a, int n, int i);
```

On exige une complexité temporelle en $O(n^2)$, que l'on justifiera.

▶ **Question 12** Écrire une nouvelle version de la fonction `closure` ayant une complexité temporelle en $O(n^3)$.

```
bool **closure(bool **a, int n);
```

Axiome On dit qu'un sommet x du graphe est un *axiome* s'il possède la propriété suivante : pour tout sommet y tel que $y \Rightarrow x$, on a $x \Rightarrow y$.

▶ **Question 13** Donner tous les axiomes du graphe G_1.

▶ **Question 14** Donner tous les axiomes du graphe G_2.

▶ **Question 15** Écrire une fonction `is_axiom` qui prend en entrée la matrice $B = CT(A)$ et un sommet `i`, et indique si ce sommet est un axiome.

```
bool is_axiom(bool **b, int n, int i);
```

Suite unidirectionnelle de sommets On appelle *suite unidirectionnelle de sommets* une suite finie $y_0, \dots, y_h$ (avec $h \in \mathbb{N}$) de sommets vérifiant :

- pour $i \in [0, h-1]$, $y_i \Rightarrow y_{i+1}$;
- pour $i \in [0, h-1]$, $y_{i+1} \not\Rightarrow y_i$ (y_i n'est pas accessible depuis y_i).

▶ **Question 16** Montrer que les sommets d'une suite unidirectionnelle sont deux à deux distincts.

▶ **Question 17** Soit y un sommet. Montrer qu'il existe un axiome x tel que $x \Rightarrow y$.

▶ **Question 18** Donner les composantes fortement connexes du graphe G_2.

▶ **Question 19** On considère une composante fortement connexe C contenant un axiome. Montrer que tous les sommets de C sont des axiomes.

Composante source On dit qu'une composante fortement connexe est une *composante source* si elle contient un axiome (ce qui revient à dire que tous ses éléments sont des axiomes, d'après la question précédente).

Système d'axiomes On dit qu'une partie X de V est un *système d'axiomes* si, pour tout sommet $y \in V$, il existe un sommet $x \in X$ tel que $x \Rightarrow y$.

▶ **Question 20** Montrer qu'on obtient un système d'axiomes de cardinal minimum en choisissant un et un seul sommet dans chacune des composantes source.

▶ **Question 21** Écrire une fonction `axiom_system` prenant en entrée la matrice $B = CT(A)$ et renvoyant un système d'axiome de cardinal minimum. On renverra ce système sous forme d'un bloc alloué `s` de n booléens, tel que `s[i]` soit vrai si et seulement si le sommet `i` fait partie du système d'axiomes.

```
bool *axiom_system(bool **b, int n);
```

Solutions

▶ **Question 1** On peut compléter le chemin de longueur h dont on dispose par $k - h \geqslant 0$ arcs $y \Rightarrow_0 y$. On obtient ainsi $\boxed{x \Rightarrow_k y.}$

▶ **Question 2** Si $x \Rightarrow_{n-1} y$, on a bien sûr $x \Rightarrow y$.

Si $x \Rightarrow y$, considérons le plus petit h tel que $x \Rightarrow_h y$.

- Si $h \leqslant n-1$, alors on conclut par la question précédente.
- si $h \geqslant n$, alors on remarque que le chemin fait intervenir $h+1 \geqslant n+1$ sommets. Comme $|V| = n$, il est nécessairement de la forme $x = y_0 \to \cdots y_j \to \cdots y_k \to \cdots y_h = y$ avec $y_j = y_k$ et $j < k$.
  Mais alors $x = y_0 \to \cdots y_j \to y_{k+1} \to \cdots y_h = y$ est un chemin de longueur $h - (k-j) < h$ de x à y, ce qui contredit la minimalité de h.

Donc $\boxed{x \Rightarrow y \text{ si et seulement si } x \Rightarrow_{n-1} y.}$

▶ **Question 3**

```c
bool **matrix_new(int n, int p){
    bool **m = malloc(n * sizeof(bool*));
    for (int i = 0; i < n; i++){
        m[i] = malloc(p * sizeof(bool));
        for (int j = 0; j < p; j++) {
            m[i][j] = false;
        }
    }
    return m;
}
```

▶ **Question 4**

```c
void matrix_free(bool **m, int n){
    for (int i = 0; i < n; i++) {
        free(m[i]);
    }
    free(m);
}
```

▶ **Question 5**

```c
bool **identity(int n){
    bool **m = matrix_new(n, n);
    for (int i = 0; i < n; i++) {
        m[i][i] = true;
    }
    return m;
}
```

▸ **Question 6**

```c
bool **product(bool **a, bool **b, int n, int p, int q){
    bool **c = matrix_new(n, q);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < q; j++) {
            for (int k = 0; k < p; k++) {
                c[i][j] = c[i][j] || (a[i][k] && b[k][j]);
            }
        }
    }
    return c;
}
```

▸ **Question 7** On procède par récurrence sur k.

- Pour $k = 1$, la propriété demandée est exactement la définition de A.
- On suppose la propriété vérifiée pour $k \geqslant 1$. On a (pour tout i, j) :

$$
\begin{aligned}
x_i \Rightarrow_{k+1} x_j \text{ ssi } & \exists 0 \leqslant i_1, \dots, i_{k-1} < n,\ x_i \Rightarrow_1 x_{i_1} \dots \Rightarrow_1 x_{i_{k-1}} \Rightarrow_1 x_j \\
\text{ssi } & \exists 0 \leqslant l < n,\ x_i \Rightarrow_k x_l \wedge x_l \Rightarrow_1 x_j \\
\text{ssi } & \exists 0 \leqslant l < n,\ A^k[i,l] \wedge A[l,j] && \text{hypothèse de récurrence} \\
\text{ssi } & \bigvee_{l=0}^{n-1} \left(A^k[i,l] \wedge A[l,j]\right) \\
\text{ssi } & A^{k+1}[i,j] && \text{par définition du produit matriciel}
\end{aligned}
$$

On a donc bien $\boxed{A^k[i,j] = \top \text{ ssi } x_i \Rightarrow_k x_j.}$

▸ **Question 8** Soit $k \geqslant n-1$ et $0 \leqslant i, j < n$.

- Si $A^k[i,j] = \top$, alors $x_i \Rightarrow_k x_j$ d'après la question 7 et donc $x_i \Rightarrow x_j$. Donc d'après 2, on a $x_i \Rightarrow_{n-1} x_j$ et donc $A^{n-1}[i,j] = \top$ d'après 7.
- Si $A^{n-1}[i,j] = \top$, alors $x_i \Rightarrow_{n-1} x_j$ et, comme $k \geqslant n-1$, $x_i \Rightarrow_k x_j$ d'après 1. Donc $A^k[i,j] = \top$.

▸ **Question 9** On procède par exponentiation rapide (ce n'était pas nécessaire) et l'on traite correctement le cas $n = 1$ (probablement pas utile non plus). Petite difficulté supplémentaire liée au fait qu'on fait du C : ne pas oublier de libérer les matrices intermédiaires crées.

```c
bool **power(bool **m, int n, int pow){
    if (pow == 0) return identity(n);
    bool **m2 = product(m, m, n, n, n);
    bool **b = power(m2, n, pow / 2);
    if (pow % 2 == 0) {
        matrix_free(m2, n);
        return b;
    } else {
        bool **result = product(b, m, n, n, n);
        matrix_free(m2, n);
        matrix_free(b, n);
        return result;
    }
}

bool **closure(bool **a, int n){
    return power(a, n, n - 1);
}
```

▶ **Question 10** La fonction `product` a clairement une complexité en $O(npq)$, donc $O(n^3)$ ici. Sachant qu'on effectue $O(\log n)$ multiplications avec notre méthode, on obtient une complexité en $O(n^3 \log n)$ pour `closure`.

▶ **Question 11** On pourrait extraire la ligne i de la matrice $CT(A)$, mais on ne respecterait pas la contrainte sur la complexité. On fait donc un parcours en profondeur :

```
void explore(bool **a, int n, int i, bool *known){
    known[i] = true;
    for (int j = 0; j < n; j++){
        if (a[i][j] && !known[j]) explore(a, n, j, known);
    }
}

bool *accessible(bool **a, int n, int i){
    bool *known = malloc(n * sizeof(bool));
    for (int i = 0; i < n; i++){
        known[i] = false;
    }
    explore(a, n, i, known);
    return known;
}
```

▶ **Question 12** On calcule simplement ligne par ligne.

```
bool **closure_dfs(bool **a, int n){
    bool **tc = malloc(n * sizeof(bool*));
    for (int i = 0; i < n; i++){
        tc[i] = accessible(a, n, i);
    }
    return tc;
}
```

Remarque

Si l'on procède par exponentiation de matrice, on obtient une complexité en $O(f(n) \ln n)$, où $f(n)$ est la complexité du calcul d'un produit de matrices $n \times n$. Avec la méthode naïve que nous avons employée, on a $f(n) = n^3$ et la complexité est donc moins bonne que celle obtenue ici. En utilisant, par exemple, l'algorithme diviser pour régner de Strassen en $O\left(n^{2.81}\right)$, on obtient en revanche une meilleure complexité que celle de `closure_dfs`. La situation est différente si le graphe est creux et donné par un tableau de listes d'adjacence.

▶ **Question 13** Dans le graphe G_1, $\boxed{x_0, x_2 \text{ et } x_3 \text{ sont des axiomes}}$ (depuis chacun de ces sommets, on peut atteindre tous les sommets). En revanche, $\boxed{x_1 \text{ n'est pas un axiome}}$ (car $x_0 \Rightarrow x_1$ mais $x_1 \not\Rightarrow x_0$).

▶ **Question 14** Dans le graphe G_2 :

- depuis x_0, tout le monde est accessible sauf x_3, et $x_3 \not\Rightarrow x_0$, donc x_0 est un axiome ;
- on a $x_0 \Rightarrow x_1$ et $x_0 \Rightarrow x_2$ mais x_0 n'est accessible ni depuis x_2 ni depuis x_2, donc x_1 et x_2 ne sont pas des axiomes ;
- pour x_3, la situation est symétrique de celle de x_0.

$\boxed{\text{Les axiomes du graphe } G_2 \text{ sont } x_0 \text{ et } x_3.}$

▶ **Question 15** En partant de la clôture transitive, il n'y a aucune difficulté : on vérifie que « $x_j \Rightarrow x_i$ implique $x_i \Rightarrow x_j$ » est vrai pour tout j.

```
bool is_axiom(bool **b, int n, int i){
    for (int j = 0; j < n; j++) {
        if (b[j][i] && !b[i][j]) return false;
    }
    return true;
}
```

Remarque

On n'utilise en fait qu'une ligne et une colonne de la matrice B, donc si l'on part de A il est plus efficace de ne calculer que cela. En termes de graphes, cela revient à calculer les sommets accessibles depuis i et ceux depuis lesquels i est accessible .

▶ **Question 16** Soit une suite $y_0 \Rightarrow y_1 \cdots \Rightarrow y_h$, supposons qu'on ait $y_i = y_j$ avec $i \leqslant j-1$. On a $y_i \Rightarrow y_{j-1}$ par transitivité de $\Rightarrow$ et donc $y_j \Rightarrow y_{j-1}$. La suite n'est donc pas unidirectionnelle (point 3).

Dans une suite unidirectionnelle, les propositions sont deux à deux distinctes.

▶ **Question 17** On considère l'algorithme suivant :

Entrée : une sommet x

Sortie : un axiome y tel que $y \Rightarrow x$

Initialisation : $y \leftarrow x$

Tant que y n'est pas un axiome :

Trouver z tel que $z \Rightarrow y$ et $y \not\Rightarrow z$.

$y \leftarrow z$

Renvoyer y

On prouve la correction et la terminaison, ce qui donne le résultat demandé :

- l'étape « trouver z » ne peut échouer car y n'est pas, à cet instant, un axiome ;
- la suite $y_h, \ldots, y_1, y_0 = x$ des valeurs successives de y (après h passages dans la boucle) est unidirectionnelle par construction. Donc :
  - cette suite est de longueur au plus n car constituée de sommets distincts, ce qui prouve la terminaison.
  - à tout instant, on a $y = y_h \Rightarrow x$;
- quand on sort de la boucle, on a donc bien y axiome et $y \Rightarrow x$.

▶ **Question 18** Les composantes fortement connexes sont $\{0\}, \{1,2\}$ et $\{3\}$.

▶ **Question 19** Soit $x \in C$ un axiome, y un élément de C et $z \in V$ tel que $z \Rightarrow y$. On a $y \Rightarrow x$ car $y \in C$, donc $z \Rightarrow x$. Comme x est un axiome, on en déduit $x \Rightarrow z$ et donc $y \Rightarrow x \Rightarrow z$. Donc y est un axiome.

▶ **Question 20**

Minimalité : soit X une axiomatique, C une classe source et $y \in C$. Il existe $x \in X$ tel que $x \Rightarrow y$. Mais y est un axiome (il est dans une classe source), donc $y \Rightarrow x$. Donc x et y sont dans la même composante connexe : $x \in C$. Ainsi, X contient au moins un élément de chaque classe source.

Caractère suffisant : soit X constituée d'un élément de chaque classe source, et soit $y \in C$. D'après 17, il existe un axiome x tel que $x \Rightarrow y$. En notant z l'élément de X appartenant à la classe source contenant x, on a $z \Rightarrow x \Rightarrow y$, ce qui montre que X est une axiomatique.

▶ **Question 21** Pas de difficulté particulière, on utilise le fait que si x_i est un axiome et $x_j \Rightarrow x_i$, alors $x_j \Leftrightarrow x_i$.

```
bool *axiom_system(bool **b, int n){
    bool *system = malloc(n * sizeof(bool));
    for (int i = 0; i < n; i++){
        system[i] = true;
    }
    for (int i = 0; i < n; i++){
        if (system[i] && is_axiom(b, n, i)) {
            // eliminate all the other elements in the scc of i
            for (int j = i + 1; j < n; j++){
                if (b[j][i]) system[j] = false;
            }
        } else {
            system[i] = false;
        }
    }
    return system;
}
```