Université d'Aix-Marseille - Master Informatique 1^{ere} année

UE Complexité - TD 2 - Rappels d'algorithmique de base sur les graphes

Préambule : Dans cette planche, nous abordons la manipulation de graphe en utilisant les deux représentations classiques, soit par matrices, soit par listes d'adjacence. Certaines questions seront à traiter avec les deux représentations. Dans tous les cas, il faudra fournir une évaluation de la complexité. Voici un rappel des types C utilisés (matrices et listes) pour le cas de graphes simples sans valuation :

```
#define NMAX .../* nombre maximum de sommets */

typedef int SOMMET; /* indice des sommets */

/* representation par matrices d'adjacence */
typedef struct {
                    int A[NMAX][NMAX]; /* matrice carree 0/1 */;
                    int n; /* valeur comprise entre 0 et NMAX */
} GRAPHEMAT;

/* representation par listes d'adjacence avec en premier definition des listes */
typedef struct maillon {
                          SOMMET st;
                          struct maillon *suivant;
} CHAINON;
typedef CHAINON *PTR_CHAINON;

typedef struct {
                    PTR_CHAINON A[NMAX]; /* tableau de listes */;
                    int n; /* valeur comprise entre 0 et NMAX */
} GRAPHELIST;
```

Cette "implémentation" en C des structures de données est proposée à titre indicatif et pour rappeler précisément le cadre. Mais comme il s'agit ensuite d'écrire des algorithmes, il va de soi qu'il n'est pas obligé de les reprendre dans les exercices.

Avant-propos de la correction. Dans ce corrigé, nous allons utiliser les notations suivantes, identiques à celles proposées en cours :

- un graphe G, qu'il soit orienté ou non sera noté par un couple $G = (S,A)$ où S est l'ensemble de sommets et A l'ensemble d'arcs ou d'arêtes (on utilise parfois $G = (V,E)$ pour *Vertices* et *Edges*..., mais ici, nous utiliserons les notations francophones)
- le nombre de sommets est $|S| = n$
- le nombre d'arêtes est $|A| = m$
- pour le cas des graphes orientés le nombre de successeurs d'un sommet x est noté $d^+(x)$
- pour le cas des graphes orientés le nombre d'arcs est donc $m = \Sigma_{x \in S} d^+(x)$

Exercice 1. Algorithmes de base sur les graphes

Il est proposé ici d'écrire plusieurs algorithmes de graphes. Il est suggéré de les écrire pour chacune des représentations (matrice et listes).

Question 1. Test d'existence d'un arc (graphes orientés). Donnez un algorithme qui prend en entrées un graphe orienté G et deux sommets x et y et qui vérifie si l'arc (x,y) est présent dans le graphe G. Donnez une évaluation de sa complexité.

Solution. Si le graphe est représenté avec une matrice d'adjacence, pour savoir s'il y a un arc (x,y) il suffit de vérifier l'entrée correspondante de la matrice :

```
algorithme arc(G, x, y):
    retourner G.A[x][y]
```

Cela ne prend que du temps constant $\Theta(1)$, indépendamment du nombre de sommets ou d'arcs du graphe.

Pour ce qui concerne la représentation en listes d'adjacence, il faut vérifier la présence du sommet y dans la liste d'adjacence du sommet x. Un pointeur p est utilisé pour parcourir cette liste.

```
algorithme arc(G, x, y):
    p := G.A[x]                    # pointeur vers la liste d'adjacence de x
    tant que p != nil faire        # il y a encore des maillons a verifier
        if p.st = y alors
            retourner vrai         # on a trouve y dans la liste de x
        sinon
            p := p.suivant         # on continue avec le maillon suivant
    retourner faux                 # y n'est pas dans la liste de x
```

Dans le pire des cas, cet algorithme prend un temps proportionnel à la longueur de la liste d'adjacence de x puisque celle-ci pourra être intégralement parcourue si y ne s'y trouve pas. Le coût est alors proportionnel au demi-degré extérieur de x, soit $d^+(x)$. On a donc une complexité en $\Theta(d^+(x))$. Sachant que pour tout sommet du graphe, $d^+(x) \leq n$, on pourrait écrire que la complexité est $O(n)$ et cela ne serait pas faux car la notation "O" exprime une majoration. Mais cela serait imprécis et conduirait à évaluer avec imprécision des algorithmes ayant à parcourir toutes les listes d'adjacence (cf. certaines questions ultérieures).

Question 2. Calcul des successeurs d'un sommet (graphes orientés). Donnez un algorithme qui prend en entrées un graphe orienté G et un sommets x et qui calcule l'ensemble E des sommets successeurs de x dans le graphe G. Dans une première version de l'algorithme, vous utiliserez une représentation d'ensembles de sommets par tableau de booléens, et dans une seconde, vous utiliserez une représentation d'ensembles de sommets par liste simplement chaînée. Donnez une évaluation de la complexité pour chaque version de l'algorithme.

Solution 1. Si le graphe est codé en **matrice d'adjacence** et la sortie en **tableau de booléens**, il suffit de retourner la x^{eme} rangée de la matrice d'adjacence puisqu'elle contient les successeurs de x :

```
algorithme successeurs(G, x):
    Succ : tableau de booleens de longueur G.n
    pour y := 1 a G.n faire
        Succ[y] := G.A[x][y]
    retourner Succ
```

Cela prend du temps $\Theta(n)$, le temps pour parcourir et recopier une rangée de la matrice.

Solution 2. Si le graphe est codé en **matrice d'adjacence** et la sortie en **liste simplement chaînée**, on peut écrire un algorithme ayant le même temps de calcul, soit $\Theta(n)$, car il faut parcourir la x^{eme} rangée de la matrice et parceque l'insertion d'un élément en tête d'une liste chaînée prend du temps constant :

```
algorithme successeurs(G, x):
    Succ := nil                     # liste initialement vide
    pour y := 1 a G.n faire
        si G.A[x][y] alors        # y est un successeur de x
            p := nouveau maillon  # insertion de y en tête de la liste pointee par Succ
            p.st := y
            p.suivant := Succ
            Succ := p             # le nouveau maillons devient le premier
    retourner Succ
```

Solution 3. Si le graphe est codé en **listes d'adjacence** et la sortie en **liste simplement chaînée**, il s'agit de recopier la liste d'adjacence de x, qui contient déjà ses successeurs. Notons que l'algorithme suivant a pour résultat la liste d'adjacence recopiée en ordre inverse, puisque les insertions sont réalisées en tête de liste. Cela étant, toutes les permutations d'une liste d'adjacence contiennent les mêmes informations, car par principe, une liste d'adjacence est un ensemble non ordonné de sommets :

```
algorithme successeurs(G, x):
    Succ := nil                      # liste initialement vide
    p := G.A[x]                    # tête de la liste de x
    tant que m != nil faire
        p' := nouveau maillon     # insertion de p.y en tête de la liste pointee par Succ
        p'.st := p.y
        p'.suivant := Succ
        Succ := p'                   # le nouveau maillons devient le premier
        p := p.suivant
    retourner Succ
```

Le temps de calcul de cet algorithme est $\Theta(d^+(x))$, puisqu'il parcourt la liste d'adjacence de x en effectuant des opérations qui prennent un temps constant (voir la deuxième partie de la solution à la question 1).

Solution 4. Enfin, si le graphe est codé en **listes d'adjacence** et la sortie en **tableau de booléens**, on peut écrire un algorithme ayant un temps de calcul en $\Theta(n)$ car il est nécessaire d'initialiser le tableau avec des 0, indépendamment du nombre de sommets successeurs, et comme le parcours de la liste prend toujours un temps en $\Theta(d^+(x))$ et que $d^+(x) \leq n$, on obtient bien une complexité en $\Theta(n)$:

```
algorithme successeurs(G, x):
    Succ : tableau de booleens de longueur G.n
    pour y := 1 a G.n faire
        Succ[y] := 0
    p := G.A[x]
    tant que p != nil faire
        Succ[p.st] := 1
        p := p.suivant
    retourner Succ
```

Question 3. Calcul des prédécesseurs d'un sommet (graphes orientés). Donnez un algorithme qui prend en entrées un graphe orienté G et un sommets x et qui calcule l'ensemble E des sommets prédécesseurs de x dans le graphe G. Dans une première version de l'algorithme, vous utiliserez une représentation d'ensembles de sommets par tableau de booléns, et dans une seconde, vous utiliserez une représentation d'ensembles de sommets par liste simplement chaînée. Donnez une évaluation de la complexité pour chaque version de l'algorithme.

Solution 1. Si le graphe est codé en **matrice d'adjacence** et la sortie en **tableau de booléens**, il suffit de fournir en résultat la x^{eme} colonne de la matrice d'adjacence, comme elle contient les sommets y qui sont reliés par un arc sortant à x, i.e. un arc de la forme (y,x), donc ses prédécesseurs :

```
algorithme  predecesseurs(G, x):
    Pred : tableau de booleens de longueur G.n
    pour y := 1 a G.n faire
        Pred[y] := G.A[y][x]
    retourner Pred
```

Ce traitement est en $\Theta(n)$, le temps pour parcourir et recopier une colonne de la matrice.

Solution 2. Si le graphe est codé en **matrice d'adjacence** et la sortie en **liste simplement chaînée**, on peut écrire un algorithme ayant le même temps de calcul, soit $\Theta(n)$, car si l'insertion d'un élément en tête d'une liste chaînée prend un temps constant, il faut malgré tout parcourir la x^{eme} colonne de la matrice d'adjacence :

```
algorithme predecesseurs(G, x):
    Pred := nil
    pour y := 1 a G.n faire
        si G.A[y][x] alors        # y est un predecesseur de x
            p := nouveau maillon
            p.st := y
            p.suivant := Pred
            Pred := p
    retourner Pred
```

Solution 3. Si le graphe est codé en **listes d'adjacence** et la sortie en **liste simplement chaînée**, il s'agit de parcourir *toutes* les listes d'adjacence pour chercher les occurrences de x, puisque ce sommet peut apparaître dans la liste d'adjacence de n'importe quel sommet y, et potentiellement de plusieurs :

```
algorithme predecesseurs(G, x):
    Pred := nil
    pour y := 1 a G.n faire
        p := G.A[y]
        tant que p != nil et p.st != x faire
            p := p.suivant
        if p != nil alors                    # on a trouve x
            p' := nouveau maillon
            p'.st := y
            p'.suivant := Pred
            Pred := p'
    retourner Pred
```

Le temps de calcul de cet algorithme est $\Theta(n+m)$, puisqu'il parcourt la totalité du tableau `G.A` en n étapes donc, et, pour chaque case `G.A[y]` du tableau `G.A`, il parcourt dans le pire des cas la totalité de la liste d'adjacence correspondante, soit en $\Theta(d^+(y))$ opérations ; la somme des longueurs des listes n'est rien d'autre que le nombre d'arcs du graphe car $m = \Sigma_{y \in S} d^+(y)$. Remarquez que ce temps de calcul reste linéaire par rapport à la taille t_G du graphe en entrée car $t_G \in \Theta(n+m)$.

Solution 4. Enfin, si le graphe est codé en **listes d'adjacence** et la sortie en **tableau de booléens**, on peut écrire un algorithme ayant le même temps de calcul $\Theta(n+m)$, puisque il est nécessaire, dans le pire des cas, de parcourir la totalité des n listes d'adjacence. On note que s'il est toujours nécessaire d'initialiser le tableau de booléens en sortie avec des 0, cette fois-ci ce temps total est dominé par d'autres opérations :

```
algorithme predecesseurs(G, x):
    Pred : tableau de booleens de longueur G.n
    pour y := 1 a G.n faire
        Pred[y] := 0
        p := G.A[y]
        tant que p != nil et p.st != x faire
            p := p.suivant
        if p != nil alors                    # on a trouve x
            Pred[y] := 1
    retourner Pred
```

Question 4. Graphe réciproque (graphes orientés). Donnez un algorithme qui prend en entrée un graphe orienté $G = (S,A)$ et calcule son graphe réciproque $G^{-1} = (S,A^{-1})$. Donnez une évaluation de sa complexité.

Solution 1. Si le graphe est représenté par une **matrice d'adjacence**, son graphe réciproque aura pour matrice d'adjacence la matrice transposée de G:

```
algorithme reciproque(G):
    H : nouveau graphe
    H.n := G.n
    pour x := 1 a G.n faire
        pour y := 1 a G.n faire
            H.A[x][y] = G.A[y][x]
    retourner H
```

Le temps de calcul est $\Theta(n^2)$, qui est du temps linéaire, vu que la taille de l'entrée t_G vérifie $t_G \in \Theta(n^2)$.

Solution 2. Si le graphe est représenté par des **listes d'adjacence**, il faudra toutes les parcourir et pour chaque arc (x,y) trouvé, ajouter l'arc (y,x) au résultat (c'est-à-dire, ajouter le sommet x à la liste d'adjacence de y) :

```
algorithme reciproque(G):
    H := nouveau graphe
    H.n := G.n
    pour x := 1 a G.n faire
        H.A[x] := nil
    pour x := 1 a G.n faire
        p := G.A[x]
        tant que p != nil faire         # (x,p.st) est un arc de G
            p' := nouveau maillon
            p'.st := x
            p'.suivant := H.A[p.st]     # (p.st,x) est maintenant un arc de H
            H.A[p.st] := p'
    retourner H
```

Le complexité en temps de cet algorithme est $\Theta(n + m)$, puisque il parcourt la totalité du tableau `G.A` et, pour chaque case `G.A[x]` du tableau, il parcourt la totalité de la liste d'adjacence correspondante, donc globalement, la totalité des arcs du graphe en effectuant des opérations en temps constant pour chaque arc. On a donc un temps en $\Theta(n + \Sigma_{x \in S} d^+(x)) = \Theta(n + m)$. En fait, c'est le même raisonnement que pour le calcul des prédécesseurs dans la question 3.

Question 5. Symétrisation (graphes orientés). Donnez un algorithme qui prend en entrée un graphe orienté $G = (S,A)$ et calcule son graphe symétrisé $G_{Sym} = (S,A \cup A^{-1})$. Donnez une évaluation de sa complexité. Pour le cas des listes d'adjacence, il existe un algorithme linéaire... Il faudrait le trouver, sachant qu'il est interdit de construire un multigraphe, c'est-à-dire, un graphe avec potentielle duplication d'arcs.

Solution 1. Si G est représenté par une **matrice d'adjacence**, son graphe symétrisé aura un arc (x,y) si et seulement si soit l'arc (x,y), soit l'arc (y,x) existent dans G. Cela se traduit assez directement en pseudo-code :

```
algorithme symétrise(G):
    H := nouveau graphe de G.n sommets
    pour x := 1 à G.n faire
        pour y := 1 à G.n faire
            H.A[x][y] := G.A[x][y] ou G.A[y][x]
    retourner H
```

Le temps de calcul est $\Theta(n^2)$, qui est du temps linéaire, vu que la taille de l'entrée t_G vérifie $t_G \in \Theta(n^2)$.

Solution 2. Si le graphe est représenté par des **listes d'adjacence**, on commence par initialiser G_{Sym} en dupliquant G (ligne 2 du pseudo-code qui suit), ce qui prend du temps $\Theta(m+n)$. Il faudra ajouter à G_{Sym} les arcs du réciproque de G (appelé `G'` dans le pseudo-code) sans créer un multigraphe. On calcule le réciproque de G (ligne 3) en temps $\Theta(m+n)$ (question 4). On initialise également à 0 un tableau de booléens `B`, qui servira à garder trace des arcs qui apparaissent déjà dans G_{Sym} (lignes 4–6), ce qui prend du temps $\Theta(n)$. Pour chaque sommet x du graphe (ligne 7) :

- On parcourt sa liste d'adjacence dans G_{Sym}, en affectant 1 à `B[y]` si l'arc (x,y) y apparaît (lignes 8–11), avec un coût de $\Theta(d^+(x))$.
- Maintenant, on parcourt la liste d'adjacence de x dans le réciproque `G'` pour trouver les arcs de G entrant dans x et les ajouter à G_{Sym} (lignes 12–19); cependant, on ne fait pas cela si l'arc existe déjà, ce qu'on peut vérifier en regardant la case correspondante du tableau `B` (ligne 14). Les lignes 12–19 ont donc un coût de $\Theta(d^-(x))$.
- Il faut remettre à 0 le tableau `B`. Pour éviter de le parcourir entièrement, il suffit d'affecter 0 aux cases correspondantes au sommets qui apparaissent dans la liste d'adjacence de x dans G (lignes 21–23), ce qui ne demande que du temps $\Theta(d^+(x))$.

On peut enfin renvoyer le résultat G_{Sym}.

```
1     algorithme symetrise(G):
2         Gsym := dupliquer(G)
3         G' := reciproque(G)
4         B := tableau de longueur G.n
5         pour x := 1 à G.n faire
6             B[x] := 0
7         pour x := 1 à G.n faire
8             p := Gsym.A[x]
9             tant que p != nil faire
10                B[p.st] := 1
11                p := p.suivant
12            p := G'.A[x]
13            tant que p != nil faire
14                si B[p.st] = 0 alors
15                    p' := nouveau maillon
16                    p'.st = p.st
17                    p'.suivant := Gsym.A[x]
18                    Gsym.A[x] := p'
19                p := p.suivant
20            p := G.A[x]
21            tant que p != nil faire
22                B[p.st] := 0
23                p := p.suivant
24        retourner Gsym
```

Le temps de calcul total de la boucle des lignes 7–23 est donc $\Theta(n)$ (puisque on parcourt tous les sommets) et, pour chaque sommet x, du temps proportionnel à $2d^+(x) + d^-(x)$, ce qui donne du temps $3m \in \Theta(m)$ sur tous les sommets (puisque $\sum_{x\in S} d^+(x) = \sum_{x\in S} d^-(x) = m$) et donc $\Theta(n+m)$ pour la boucle. Donc cet algorithme a un coût total de $\Theta(n+m)$, qui est du temps linéaire.

Question 6. Complémentaire (graphes non-orientés). Donnez un algorithme qui prend en entrée un graphe non-orienté $G = (S,A)$ et calcule son graphe complémentaire $\overline{G}$. Donnez une évaluation de sa complexité.

Solution 1. Si le graphe est représenté par une **matrice d'adjacence**, il suffit de calculer le complémentaire de chaque bit, sauf pour ceux dans la diagonale (puisque on ne veut pas ajouter de boucles).

```
algorithme complementaire-matrice(G):
    H := nouveau graphe de G.n sommets
    pour x := 1 à G.n faire
        pour y := 1 à G.n faire
            si x != y alors
                H.A[x][y] := non G.A[x][y]
            sinon
                H.A[x][y] := 0
    retourner H
```

Cet algorithme a un coût de $\Theta(n^2)$, qui est du temps linéaire.

Solution 2. Si le graphe est représenté par des **listes d'adjacence**, on peut d'abord calculer sa représentation en matrice d'adjacence, puis utiliser l'algorithme de la solution 1, et enfin le reconvertir en listes d'adjacence.

```
algorithme complementaire-listes(G):
    G' := nouveau graphe de n sommets en matrice d'adjacence
    G'.n := G.n
    pour x := 1 à G'.n faire
        pour y := 1 à G'.n faire
            G'.A[x][y] := 0
    pour x := 1 à G.n faire
        p := G.A[x]
        tant que p != nil faire
            G'.A[x][p.st] := 1
            p := p.suivant
    H := complementaire-matrice(G')
    H' := nouveau graphe de n sommets en listes d'adjacence
    pour x := 1 à H.n faire
        pour y := 1 à H.n faire
            si H.A[x][y] alors
                p := nouveau maillon
                p.st = y
                p.suivant = H'.A[x]
                H'.A[x] := p
    retourner H'
```

Le temps d'exécution de cet algorithme est dominé par les parcours des graphes représentés en matrice d'adjacence, ce qui donne un temps de calcul de $\Theta(n^2)$. Remarquez qu'il ne s'agit pas du temps linéaire par rapport à la taille de l'entrée $n + m$ (par exemple, le graphe pourrait avoir $m = 0$ arcs).

On pourrait se demander si, en évitant la conversion en matrice d'adjacence, on peut trouver un algorithme de temps linéaire $\Theta(n + m)$ pour calculer le complémentaire d'un graphe en listes d'adjacence. Néanmoins, il est toujours nécessaire de parcourir chaque arête potentielle $\{x,y\} \subset S$, soit parce que $\{x,y\}$ fait partie du graphe d'entrée, soit parce qu'il n'en fait pas partie et il faut donc l'ajouter au résultat. Par conséquent, tout algorithme correct pour ce problème doit effectuer au moins de l'ordre de n^2 opérations. Si vous avez des difficultés pour vous en assurer, on sait que le graphe complet contient $\frac{n(n-1)}{2}$ arêtes. Si $m \geq \frac{n(n-1)}{4}$, i.e. si G contient au moins la moitié des arêtes possibles, alors on a $m \in \Theta(n^2)$, et le simple accès à ces arêtes est en $\Theta(m) = \Theta(n^2)$. Si par contre, si G contient moins de la moitié des arêtes possibles, i.e. si $m < \frac{n(n-1)}{4}$, nécessairement, le complémentaire de G contiendra un nombre d'arêtes m' tel que $m' > \frac{n(n-1)}{4}$ qu'il faudra construire et le coût sera donc aussi en $\Theta(n^2)$.

Question 7. Test de stable (graphes non-orientées). Donnez un algorithme qui prend en entrées un graphe non-orienté $G = (S,A)$ et un sous-ensemble K de ses sommets, et vérifie si cet ensemble K est un stable du graphe G (un *stable* est un ensemble de sommets deux à deux non adjacents). Donnez une évaluation de sa complexité.

Solution 1. Il suffit de vérifier que l'arête $\{x,y\}$ n'existe pas, pour tout $x,y \in K$. Si le graphe est représenté par une **matrice d'adjacence** et l'ensemble K par un vecteur de booléens, cela ne prend que du temps constant de vérifier, pour un couple $x,y \in K$, si l'arête correspondante existe (exercice 1, question 1). En revanche, il faut traverser tout le tableau K pour trouver quels sommets il contient :

```
algorithme test-stable(G, K):
    pour x := 1 à G.n faire
        pour y := x + 1 à G.n faire
            si K[x] et K[y] et arc(G, x, y) alors
                retourner faux
    retourner vrai
```

Le temps de calcul de cet algorithme est donc $\Theta(n^2)$, ce qui est atteint dans le cas où K soit effectivement un stable et donc on ne termine pas l'exécution en avance. La taille de l'entrée étant $n^2 + n$, il s'agit de temps linéaire.

Solution 2. Si le graphe est représenté par des **listes d'adjacence** et l'ensemble K par une liste chaînée, on peut d'abord transformer K en tableau de booléens K', ce qui prend du temps $\Theta(n)$ dominé par l'initialisation à 0. Ensuite, on traverse la liste K et, pour chaque sommet y qui y apparaît, on transforme également sa liste d'adjacence en tableau de booléens T, ce qui prend $\Theta(n)$. On peut donc comparer les deux sous-ensembles K' et T pour vérifier qu'aucun sommet du stable potentiel soit adjacent à y ; si ce n'est pas le cas, on a vérifié que K n'est finalement pas un stable, sinon on continue avec le sommet suivant de K. Si on ne trouve aucune arête, cela signifie que K est tout à fait un stable.

```
algorithme test-stable(G, K):
    K' := tableau de longueur G.n
    pour x := 1 à G.n faire
        K'[x] := 0
    p := K
    tant que p != nil faire
        K'[p.st] := 1
        p := p.suivant
    p := K
    tant que p != nil faire
        y := p.st
        T := tableau de longueur G.n
        pour x := 1 à G.n faire
            T[x] := 0
        q := G.A[y]
        tant que q != nil faire
            T[q.st] := 1
            q := q.suivant
        pour x := 1 à G.n faire
            si K'[x] et T[x] faire
                retourner faux
        p := p.suivant
    retourner vrai
```

Le temps de calcul de cet algorithme est $\Theta(kn)$, où $k = |K|$.

Question 8. Test de stable maximal (graphes non-orientés). Donnez un algorithme qui prend en entrées un graphe non-orienté $G = (S,A)$ et un sous-ensemble K de ses sommets, et vérifie si cet ensemble K est un stable maximal du graphe G. On dit qu'un stable de G est *maximal* s'il n'est inclus strictement dans aucun autre stable de G. Donnez une évaluation de sa complexité.

Solution. Un stable K n'est pas maximal si et seulement si il existe un sommet $x \notin K$ tel que l'arête $\{x,y\}$ n'existe pas pour aucun $y \in K$.

```
algorithme test-stable-maximal(G, K):
    if non test-stable(G, K):
        retourner faux
    pour x := 1 à G.n faire
        si non K[x] alors
            y := 1
            tant que y <= G.n et (non K[y] ou non arc(G, x, y)) faire
                y := y + 1
            si y > G.n alors
                retourner faux
    retourner vrai
```

Le temps de calcul de cet algorithme est $\Theta(n^2)$ si G est représenté par sa matrice d'adjacence.

Question 9. Stable maximum (graphes non-orientés). Que faudrait-il faire pour pour s'assurer qu'étant donné un graphe non-orienté $G = (S,A)$ et un sous-ensemble K de ses sommets, K est un stable maximum du graphe G (i.e. il n'existe aucun autre stable de G qui contienne strictement plus de sommets que K)? Attention, il n'est pas demandé ici de donner un algorithme.

Solution. Il faut, à priori, vérifier pour tout ensemble de sommets K' de taille $|K| + 1$ que K' n'est pas un stable. Cela inclut les ensembles K' qui ne contiennent pas K.

Question 10. Existence d'un circuit (graphes orientés). Donnez un algorithme qui prend en entrée un graphe orienté $G = (S,A)$ et calcule un circuit de G s'il en existe un.

Solution 1. Il est possible de détecter la présence d'un circuit en parcourant en profondeur la graphe, en gardant trace des sommets déjà visités ; si a un moment donné on retrouve un sommet déjà visité pendant la descente courante, cela indique la présence d'un circuit. L'algorithme suivant, qui prend en entrée un graphe sous forme de **listes d'adjacence**, colorie les sommets en fonction de leurs état : blanc (pas encore visité), gris (en cours de visite, c'est-à-dire avec des arcs sortants pas encore parcourus) et noir (sommet visité). On utilise une procédure auxiliaire `circuit-aux` pour effectuer la visite en profondeur de façon récursive.

L'algorithme construit également un tableau `prédécesseur`, qui contient les prédécesseurs des sommets visités pendant le parcours en profondeur (la valeur -1 est utilisé pour les sommets pas encore visités ou visités en premiers). Ce tableau est renvoyé comme résultat, avec le sommet x où on découvre en premier l'existence d'un circuit, et permet de reconstruire le circuit lui-même en suivant en arrière récursivement les prédécesseurs à partir de x. La valeur `nil` est renvoyée si le graphe ne contient pas de circuit.

```
algorithme circuit(G):
    couleur := tableau de longueur G.n
    pour x := 1 à n faire
        couleur[x] := blanc
    prédécesseur := tableau de longueur G.n
    pour x := 1 à n faire
        prédécesseur[x] := -1
    pour x := 1 à n faire
        si couleur[x] = blanc alors
```

```
            résultat := circuit-aux(G, x, couleur, prédécesseur)
            si résultat != nil alors
                retourner résultat
    retourner nil

algorithme circuit-aux(G, x, couleur, prédécesseur):
    couleur[x] := gris
    p := G.A[x]
    tant que p != nil faire
        si couleur[p.st] = gris alors
            retourner (x, prédécesseur)
        sinon si couleur[p.st] = blanc alors
            prédécesseur[p.st] := x
            résultat := circuit-aux(G, p.st, couleur, prédécesseur)
            si résultat != nil alors
                retourner résultat
        p := p.suivant
    couleur[x] := noir
    retourner nil
```

Le temps de calcul est le même que pour une visite en profondeur, sauf qu'on peut l'interrompre dès qu'on trouve le circuit. La procédure `circuit` initialise les tableaux `couleur` et `prédécesseur` pour chaque sommet en temps $\Theta(n)$ et ensuite appelle `circuit-aux` sur chaque sommet pas encore visité. La visite en soi, dans le pire des cas (où on ne trouve pas de circuit) traverse chaque arc du graphe exactement une fois, et donc prend du temps $\Theta(m)$. En total on a donc du temps $\Theta(n+m)$, ce qui est linéaire par rapport à la taille de l'entrée.

Solution 2. Si on utilise une représentation en **matrice d'adjacence**, la procédure `circuit` reste la même, mais il faut modifier `circuit-aux` pour chercher les sommets adjacents à chaque sommet dans la matrice :

```
algorithme circuit-aux(G, x, couleur, prédécesseur):
    couleur[x] := gris
    for y := 1 à G.n faire
        si couleur[p.st] = gris alors
            retourner (x, prédécesseur)
        sinon si G.A[x][y] et couleur[y] = blanc alors
            prédécesseur[y] := x
            résultat := circuit-aux(G, y, couleur, prédécesseur)
            si résultat != nil alors
                retourner résultat
    couleur[x] := noir
    retourner nil
```

Donc, dans cette version de l'algorithme on prend du temps $\Theta(n)$ pour analyser (dans le pire des cas) tout les sommets adjacents aux sommet courant, ce qui donne un temps de calcul $\Theta(n^2)$. Pour une représentation en matrice d'adjacence, il s'agit quand même de temps linéaire.

Exercice 2. Problème du transversal

Un *transversal* dans un graphe non-orienté $G = (S,A)$, est un sous-ensemble X de S tel que pour toute arête $\{x,y\} \in A$, alors $x \in X$ ou $y \in X$ (*ou* non exclusif) . En d'autres termes, un transversal est un sous-ensemble de sommets partageant au moins un sommet avec chaque arête du graphe. Seule la représentation de graphes

par matrices d'adjacence sera considérée ici.

Question 1. Donnez un algorithme qui prend en entrées un graphe non-orienté G et un ensemble de sommets X, et qui vérifie si X est un transversal de G. Donnez une évaluation de sa complexité.

Solution. Pour chaque arête du graphe, qu'on peut repérer en parcourant la moitié supérieure de la matrice d'adjacence (qui est symétrique dans le cas d'un graphe non orienté), il faut vérifier qu'au moins l'une des deux extrémités appartienne à X, ici représenté par un tableau de booléens :

```
algorithme test-transversal(G, X):
    pour x := 1 à G.n faire
        pour y := x + 1 à G.n faire
            si G.A[x][y] et non X[x] et non X[y] alors
                retourner faux
    retourner vrai
```

Cet algorithme prend du temps proportionnel à $\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$ dans le pire des cas, quand X est effectivement un transversal, et donc du temps $\Theta(n^2)$ en général ; il s'agit d'un temps linéaire par rapport à la taille $n^2 + n$ de l'entrée.

Question 2. Donnez un algorithme qui prend en entrées un graphe non-orienté G et un ensemble de sommets X, et qui vérifie si X est un transversal minimal de G, c'est-à-dire que X ne possède aucun sous-ensemble strict qui soit aussi un transversal. Donnez une évaluation de sa complexité.

Solution. Pour que X soit un transversal minimal de G, il faut que l'ensemble X soit bien évidemment un transversal, et qu'il n'existe pas de sommet $x \in X$ tel que $X \backslash \{x\}$ soit un transversal de G. On peut donc essayer retirer de X chaque sommet (mais un seul à la fois) et vérifier si le sous-ensemble considéré est ou non un transversal :

```
algorithme test-transversal-minimal(G, X):
    si non test-transversal(G, X):
        retourner faux
    pour x := 1 à G.n faire
        si X[x] alors
            trouvé := faux
            pour y := 1 à G.n faire
                si G.A[x][y] et non X[y] alors
                    trouvé := vrai
            si non trouvé alors
                retourner faux
    retourner vrai
```

Cet algorithme applique la procédure `test-transversal` de la question 1 à X pour vérifier s'il est un transversal. Pour vérifier s'il existe a un sous-ensemble propre de X qui est un transversal, pour chaque $x \in X$ on cherche un sommet $y \notin X$ tel que $\{x,y\} \in A$. Si un tel y existe, alors on ne peut pas supprimer x de X sans perdre la proprieté de transversal ; en revanche, si on trouve un tel y, l'ensemble $X \setminus \{x\}$ est un transversal contenu dans X, et donc X n'est pas minimal. Le temps d'exécution de cet algorithme est $\Theta(n^2)$.

Question 3. Donnez un algorithme qui prend en entrée un graphe non-orienté G et qui fournit en résultat un transversal minimal de G. Donnez une évaluation de sa complexité.

Solution. Il est possible de trouver un transversal minimal avec un algorithme glouton qui, a partir d'un transversal X quelconque, essaye de retirer un par un les éléments de X, si c'est possible de le faire sans perdre la propriété de transversal. Comme transversal initial, on peut toujours choisir $X = S$, vu que l'ensemble des sommets du graphe contient au moins l'une des extrémités (en effet, les deux) de chaque arête.

```
algorithme trouver-transversal-minimal(G):
    X := tableau de longueur G.n
    pour x := 1 à n faire
        X[x] := vrai
    pour x := 1 à G.n faire
        X[x] := faux
        si non test-transversal(G, X) alors
            X[x] := vrai
    retourner X
```

Le temps de calcul de cet algorithme est $\Theta(n)$ pour initialiser l'ensemble X, puis on applique l'algorithme `test-transversal` de la question 1 à n ensembles de taille bornée par n. Mais comme le coût de `test-transversal` est $\Theta(n^2)$ dans tous les cas, on obtient une complexié de $\Theta(n^3)$, super-linéaire par rapport à la taille n^2 de l'entrée.

Question 4. Donnez un algorithme qui prend en entrées un graphe non-orienté G et un entier k, et teste si le graphe G possède un transversal de taille k ou moins. Donnez une évaluation de sa complexité.

Solution. Pour vérifier l'existence d'un transversal de taille k il semble être nécessaire de tester, dans le pire des cas, tout les sous-ensembles de k sommets (il n'est pas strictement nécessaire de tester les sous-ensembles de moins de k sommets, puisque s'il existe un transversal X de taille $\ell < k$ alors il en existe aussi un de taille k, obtenu en ajoutant n'importe quel ensemble de $k - \ell$ sommets de $S \setminus X$).

Il y a $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ sous-ensembles de taille k. On peut montrer que, en fonction de k, cette valeur peut être exponentiel par rapport à n. Plus spécifiquement, en choisissant $k = n/2$ on obtient

$$\begin{aligned}\binom{n}{k} = \binom{n}{n/2} = \frac{n!}{(\frac{n}{2})!(\frac{n}{2})!} &= \frac{n \times (n-1) \times \cdots \times (\frac{n}{2}+1) \times \frac{n}{2} \times (\frac{n}{2}-1) \times \cdots \times 2 \times 1}{\left(\frac{n}{2} \times (\frac{n}{2}-1) \times \cdots \times 2 \times 1\right)^2} \\ &= \frac{n \times (n-1) \times \cdots \times (\frac{n}{2}+1)}{\frac{n}{2} \times (\frac{n}{2}-1) \times \cdots \times 2 \times 1} \\ &= \frac{n}{\frac{n}{2}} \times \frac{n-1}{\frac{n}{2}-1} \times \frac{n-2}{\frac{n}{2}-2} \times \cdots \times \frac{\frac{n}{2}+2}{2} + \frac{\frac{n}{2}+1}{1}\end{aligned}$$

Chaque terme du produit à la forme $\frac{n-i}{\frac{n}{2}-i}$ pour $i = 0, \ldots, \frac{n}{2} - 1$, et on a toujours $\frac{n-i}{\frac{n}{2}-i} \geq 2$, puisque

$$\frac{n-i}{\frac{n}{2}-i} \geq 2 \iff n - i \geq 2(\frac{n}{2} - i) \iff n - i \geq n - 2i \iff 2i \geq i.$$

Comme il y a $n/2$ termes dans le produit, on obtient donc $\binom{n}{n/2} \geq 2^{n/2}$, ce qui nous montre que le nombre de sous-ensembles d'une certaine taille k peut être exponentiel par rapport à n. Par conséquent, pour une raison de simplicité on peut définir un algorithme qui, pour tout sous-ensemble $X \subseteq S$, vérifie d'abord si la taille de X est k, et ensuite s'il s'agit d'un transversal. Cela nous demande de tester, dans le pire des cas, tous les 2^n sous-ensembles de S, mais en tout cas on ne connaît pas d'algorithme polynomial pour ce problème.

Pour générer tous les sous-ensembles $X \subseteq S$ sous forme de tableau de booléens, on traverse le tableau et, pour chaque case, on essaye les valeurs vrai et faux, en remplissant récursivement le reste du tableau. Quand on a atteint la fin du tableau, on vérifie si le sous-ensemble obtenu a taille k et, si c'est le cas, on vérifie si c'est un transversal en utilisant la fonction `test-transversal` de la question 1. Cela conduit donc au schéma d'algorithme suivant :

```
algorithme transversal(G, k):
    X := tableau de longueur G.n
    return transversal-aux(G, k, X, 0)
```

```
algorithme transversal-aux(G, k, X, x):
    si x < G.n alors
        X[x] := faux
        si transversal-aux(G, k, X, x+1) alors
            retourner vrai
        X[x] := vrai
        si transversal-aux(G, k, X, x+1) alors
            retourner vrai
        retourner faux
    sinon
        c := 0
        pour y := 1 à n faire
            si X[y] alors
                c := c + 1
        si c = k alors
            retourner test-transversal(G, X)
```

La procédure `transversal` ne prend que du temps constant (excepté bien sûr l'appel à `transversal-aux`) vu qu'elle n'initialise pas le contenu du tableau X. La procédure `transversal-aux` prend du temps constant pour remplir une case du tableau, plus le coût de deux appels récursifs sur le reste du tableau, ou bien le coût de parcourir le tableau et d'appeler `test-transversal` si le tableau est déjà rempli. Cela revient à générer en temps $\Theta(n)$ chacun des 2^n sous-ensembles de S, à mesurer sa taille en temps $\Theta(n)$, et à vérifier si c'est un transversal en temps $\Theta(n^2)$. Cela nous donne un temps de calcul exponentiel $O(2^n n^2)$.