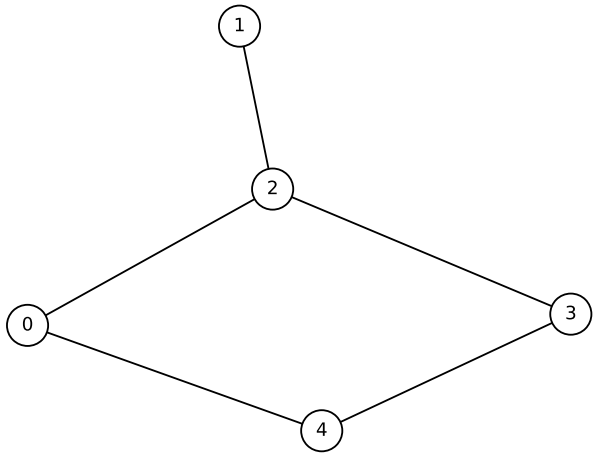
Comptage de triangles dans un graphe

1.1

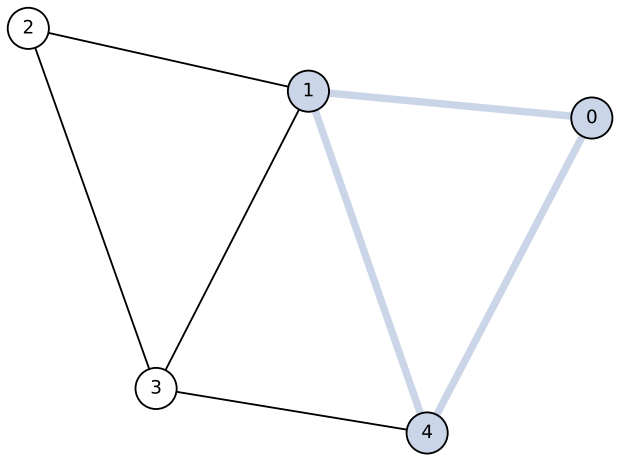
On obtient pour M1:



,

et donc M_1 n'admet **pas de triangle**.

On obtient pour M2:



,

et donc M_2 admet **un triangle**. par exemple $0, 1, 4$.

1.2

On parcourt les triplets de sommets. Plutôt que de considérer les triplets i, j, k quelconques, on considère les triplets i, j, k tels que $i < j < k$.

Si on avait pris i, j, k quelconques, on aurait compté chaque triangle 6 fois (une fois pour chaque permutation de i, j, k).

In [ ]:
```python
def recherche_triangle_brute_force(M):
    n, p = np.shape(M)
    num_triangles = 0
    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                if M[i, j] == 1 and M[j, k] == 1 and M[k, i] == 1:
                    num_triangles = num_triangles + 1
    return num_triangles
```

1.3

On a trois boucles `for` imbriquées, et dans le corps de la boucle des opérations en $O(1)$.

Par conséquent la complexité est en $O(\sum_{0 \leq i < j < k < |S|} 1) = \boxed{O(|S|^3)}$.

1.4

De même pour éviter de compter deux fois chaque arête, on se contente des arêtes i, j telles que $i < j$.

In [ ]:
```python
def liste_aretes(M):
    n, p = np.shape(M)
    A = []
    for i in range(n):
        for j in range(i + 1, n):
            if M[i, j] == 1:
                A.append((i, j))
    return A
```

1.5

On parcourt les arêtes puis les sommets pour vérifier si on a un triangle.

On remarque que chaque triangle est compté 3 fois (une fois pour chaque "base" du triangle).

In [ ]:
```python
def recherche_triangle_variante(M):
    n, p = np.shape(M)
    A = liste_aretes(M)
    num_triangles = 0
    for a in A:
        i, j = a
        for k in range(n):
            if k != i and k != j and M[i, k] == 1 and M[j, k] == 1:
                num_triangles += 1
    # Avec cette méthode, chaque triangle est compté trois fois
    return num_triangles // 3
```

1.6

- La fonction `liste_aretes` a une complexité en $O(|S|^2)$ car elle parcourt tous les couples de sommets (i, j) avec $i < j$.
- on effectue ensuite une boucle sur les arêtes, et pour chaque arête on parcourt les sommets pour vérifier si on a un triangle. D'où une complexité en $O(|A| \times |S|)$.

Au final on a donc une complexité en $\boxed{O(|S| \times (|A| + |S|))}$.

Dans le cas où le graphe a "beaucoup" d'arêtes (le maximum étant $\binom{|S|}{2}$), on a $|A| = O(|S|^2)$, et donc la complexité devient $O(|S|^3)$.

Dans le cas où le graphe a "peu" d'arêtes, et plus précisément si $|A| = O(|S|)$, on a une complexité en $O(|S|^2)$.

1.7

En notant $M = (m_{ij})_{i,j \in S}$ on a pour tout $i \in S$,

$$\begin{aligned}(M^3)_{ii} &= (MM)M \\ &= \sum_{j \in S} (MM)_{ij} (M)_{ji} \\ &= \sum_{j \in S} \sum_{k \in S} m_{ik} m_{kj} m_{ji} \\ &= \sum_{(j,k) \in S} m_{ik} m_{kj} m_{ji}\end{aligned}$$

Or pour $i, j, k \in S$, on a $m_{ik} m_{kj} m_{ji} = 1$ les arêtes (i, j), (j, k) et (k, i) sont présentes dans le graphe, et 0 sinon, c'est à dire

$$m_{ik} m_{kj} m_{ji} = \delta_{i,j,k \text{ est un triangle}}$$

et donc

$$(M^3)_{ii} = \left|\{(j, k) \in S^2 \mid i, j, k \text{ forment un triangle}\}\right|$$

On remarque que chaque tel triangle est compté **2 fois** (pour (j, k) et (k, j)), et donc on a

$$\boxed{(M^3)_{ii} = 2 \times |\{\text{ triangles contenant } i\}|}$$

1.8

Pour compter le nombre de triangles il suffit donc de calculer la trace de M^3, c'est à dire

$$\sum_{i \in S} (M^3)_{ii}$$

Puis on constate que chaque triangle est compté 2×3 fois: 2 fois pour chaque sommet, et 3 fois pour chaque permutation de ses sommets.

In [ ]:
```python
def rechercher_triangle_matrice(M):
    n, p = np.shape(M)
    num_triangles = 0
    # On effectue le calcul de M^3
    A = produit_matrice(produit_matrice(M, M), M)
    S = 0
```

```
    for i in range(n):
        S = S + A[i, i]
    # Chaque triangle est compté 2 fois
    return S // 6
```

1.9

On effectue:

- deux produits matriciels, d'où une complexité en $O(|S|^\alpha)$
- un parcourt de la diagonale de la matrice, d'où une complexité en $O(|S|)$.

Comme $\alpha > 1$ on a donc une complexité en $\boxed{O(|S|^\alpha)}$.

1.10

On utilise la formule explicite:

In [ ]:
```
def produit_matrice_naif(M, N):
    n, p1 = np.shape(M)
    p2, m = np.shape(N)
    # La vérification des tailles n'est pas demandée par le sujet
    if p1 != p2:
        raise ValueError("Tailles incompatibles")
    P = np.zeros((n, m))
    for i in range(n):
        for j in range(m):
            for k in range(p1):
                P[i, j] = P[i, j] + M[i, k] * N[k, j]
    return P
```

1.11

La fonction `produit_matrice_naif` a une complexité en $O(|S|^3)$, (trois boucles `for` avec des opérations en $O(1)$ dans le corps de la boucle).

Donc la complexité en utilisant la formule explicite est en $\boxed{O(|S|^3)}$.

On a donc la même complexité que les méthodes précédentes.

1.12

Pour calculer le produit de deux matrices de tailles $2^k \times 2^k$, on

- extrait les blocs (complexité en $O(4^k)$)
- calcule matrices $M_1, \dots, M_7$ pour chacune d'entre elle il y a des sommes en $O(4^k)$ et un produit en $C(k-1)$ opérations.
- ajoute les blocs pour obtenir le résultat (complexité en $O(4^k)$).

Donc au total

$$C(k) = O(4^k) + 7\left(C(k-1) + O(4^k)\right) + O(4^k)$$

c'est à dire qu'il existe $\gamma > 0$ tel que

$$\boxed{C(k) \leq 7C(k-1) + \gamma 4^k}.$$

1.13

Avec les notations précédentes on a pour tout $k \geq 1$,

$$\frac{C(k)}{7^k} \leq \frac{C(k-1)}{7^{k-1}} + \gamma\left(\frac{4}{7}\right)^k$$

et donc par somme télescopique, pour tout $k \geq 1$,

$$\begin{aligned}\frac{C(k)}{7^k} &\leq \frac{C(0)}{7^0} + \gamma\sum_{i=1}^{k}\left(\frac{4}{7}\right)^i \\ &= C(0) + \gamma\frac{1-\left(\frac{4}{7}\right)^{k+1}}{1-\frac{4}{7}} \\ &\leq (C(0) + \frac{7}{3}\gamma)\end{aligned}$$

Par conséquent

$$\boxed{C(k) = O(7^k)}$$

ce qui peut se réécrire comme

$$7^k = \left((2)^{\log_2(7)}\right)^k$$

$$\boxed{C(k) = O((2^k)^\alpha)}$$

où

$$\boxed{\alpha = \log_2(7)}.$$

Et comme

$$4 < 7 < 8$$

on a

$$2 < \alpha < 3$$

1.14

On ne vérifie pas dans la fonction que les matrices sont de même taille et que cette taille est une puissance de 2 (il s'agit d'une donnée de l'énoncé).

In [ ]:
```python
def produit_matrice_strassen(A, B):
    """Effectue le produit de deux matrices A et B en utilisant l'algorithme de Strassen, et en supposant que les matrices sor
    n, p = np.shape(A)
    if n == 1:
        return A * B
    m = n // 2
    # On extrait les sous-matrices de A et B
    A11 = A[:m, :m]
    A12 = A[:m, m:]
    A21 = A[m:, :m]
    A22 = A[m:, m:]
    B11 = B[:m, :m]
    B12 = B[:m, m:]
    B21 = B[m:, :m]
    B22 = B[m:, m:]
    # On calcule les matrices de Strassen
    M1 = produit_matrice_strassen(A11 + A22, B11 + B22)
    M2 = produit_matrice_strassen(A21 + A22, B11)
    M3 = produit_matrice_strassen(A11, B12 - B22)
    M4 = produit_matrice_strassen(A22, B21 - B11)
    M5 = produit_matrice_strassen(A11 + A12, B22)
    M6 = produit_matrice_strassen(A21 - A11, B11 + B12)
    M7 = produit_matrice_strassen(A12 - A22, B21 + B22)
    # On assemble les sous-matrices pour former le produit final
    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6
    C = np.zeros((n, n))
    C[:m, :m] = C11
    C[:m, m:] = C12
    C[m:, :m] = C21
    C[m:, m:] = C22
    # On retourne le produit final
    return C
```

1.15

On obtient donc une complexité de détection de triangles en $\boxed{O(|S|^{\log_2(7)})}$ et comme

$$2 < \log_2(7) < 3$$

cet algorithme est **plus efficace** que les algorithmes précédents en $O(|S|^3)$ dans le pire des cas.

Remarque culturelle: la question de la complexité optimale pour la détection de triangles est un problème ouvert en informatique théorique.
