# Structure de graphes et complexité

Youcef Taleb 191938012108
Lokmane Ammari 212131049131

June 28, 2025

## 1 Introduction

Dans un monde de plus en plus interconnecté, la compréhension des relations complexes entre les entités est devenue essentielle. La théorie des graphes, une branche des mathématiques, offre un cadre puissant pour modéliser et analyser ces structures intricantes. Des réseaux sociaux aux systèmes de transport, en passant par les réseaux biologiques et le World Wide Web, les graphes constituent un outil précieux pour représenter et raisonner sur divers phénomènes du monde réel.

Ce document vise à fournir une vue d'ensemble complète des structures de graphes et de leurs applications. Nous commencerons par introduire les concepts fondamentaux tels que les sommets, les arêtes et les différents types de graphes. Ensuite, nous explorerons les représentations courantes des graphes, notamment les matrices d'adjacence et les listes d'adjacence, et discuterons de leurs avantages et inconvénients respectifs. De plus, nous analyserons la complexité des algorithmes de graphe fondamentaux, tels que la recherche en profondeur et la recherche en largeur. Enfin, nous aborderons des sujets plus avancés.

## 2 Objectifs du travail

Ce travail vise à explorer les représentations des graphes en mémoire, à implémenter des opérations courantes sur les graphes et à analyser leurs complexités. Nous étudierons également des exemples de problèmes NP-complets sur les graphes et explorerons l'utilisation des graphes dans les applications biologiques

## 3 Partie I : définitions de base

### 3.1 Graphe

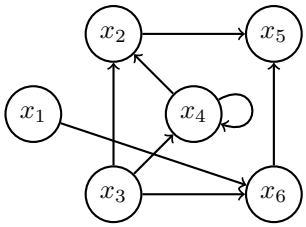
Un graphe est un ensemble X de points nommés **sommets** (parfois **nœuds**) reliés par un ensemble E de **traits** parfois non orientés nommés **arêtes** (ou **arcs** dans le cas orienté). On note $G = (X, E)$ une telle structure.

### 3.2 Graphe orienté

Un graphe orienté $G = (X, U)$ est défini par les deux ensembles:

- $X = \{x_1, x_2, ..., x_n\}$ est l'ensemble fini de **sommets**, $n \geqslant 1$.
- $U = \{u_1, u_2, ..., u_m\}$ est l'ensemble fini d'**arcs**.

Chaque élément $u_i \in U$ est une paire ordonnée de sommets, $u_i = (x, y)$. $x$ est appelé **extrémité initiale** de $u_i$ et $y$ est **extrémité terminale**. $U$ peut être vide.



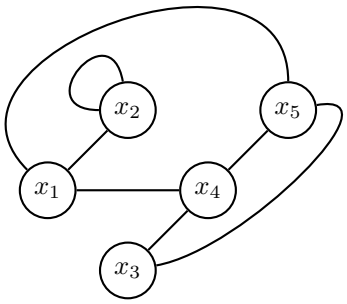

Ce graphe est défini par l'ensemble des sommets $X = \{x1, x2, x3, x4, x5, x6\}$ et l'ensemble d'arcs $U = \{(x1, x6), (x2, x5), (x3, x2), (x3, x4), (x3, x6), (x4, x2), (x4, x4), (x6, x5)\}$

## 3.3 Graphe non orienté

Un graphe orienté $G = (X, E)$ est défini par les deux ensembles:

- $X = \{x_1, x_2, ..., x_n\}$ est l'ensemble fini de **sommets**, $n \geqslant 1$.
- $E = \{e_1, e_2, ..., e_m\}$ est l'ensemble fini d'**arêtes**.

Chaque élément $e_i \in E$ est une paire non ordonnée de sommets, $e_i = (x, y)$. $x$ et $y$ sont appelés extrémités de $e_i$. $E$ peut être vide.




De même pour ce graphe, l'ensemble des sommets $X = \{x1, x2, x3, x4, x5\}$ et l'ensemble des arêtes $E = \{(x1, x2), (x1, x4), (x1, x5), (x2, x2), (x3, x4), (x3, x5), (x4, x5)\}$

## 3.4 Propriétés des graphes

- L'**ordre** d'un graphe G est le nombre de sommets dans le graphe G.
- Le **degré d'un sommet** et le nombre d'arêtes (ou d'arcs) incidentes à ce sommet. On note $\delta(G)$ le plus grand degré dans G.
- Si les deux extrémités d'un arc (resp. d'une arête) sont confondues alors cet arc (resp. cette arête) est appelé(e) **boucle**.
- Si deux arcs possèdent les mêmes extrémités initiales et mêmes extrémités terminales, on dit alors qu'ils sont **parallèles**.
- Une **chaîne** reliant un sommet u à un sommet $x$ est une succession d'arêtes qui permet de se rendre de u à v. Dans le cas orienté, on parle de **chemin** de u vers v.
- Un **cycle** dans G est une chaîne dont les deux extrémités coïncident. Dans le cas orienté, on parle de **circuit**.

Soit un arc $u = (x, y)$ (resp. un arête $e = (x, y)$) :

- x et y sont dits sommets **adjacents**.

- Pour le cas de l'arc u : x est dit **prédécesseur** de y. y est dit **successeur** de x.
- x et y sont **incidents** à l'arc u (resp. à l'arête e), l'arc u (resp. l'arête e) est aussi incident(e) aux sommets x et y.
- Deux arcs (resp. arêtes) sont dits adjacents s'ils ont une extrémité en commun.

Soit $x \in X$, un sommet du graphe orienté $G = (X, U)$. On définit :

- $\Gamma^+(x) = \{y \in X/(x, y) \in U\}$ appelé **ensemble des successeurs** de x.
- $\Gamma^-(x) = \{y \in X/(y, x) \in U\}$ appelé **ensemble des prédécesseurs** de x.

Soit $x \in X$, un sommet du graphe G, on appelle voisin de x tout sommet $y \in X$ différent de x et qui est adjacent à x. L'ensemble des voisins de x, noté V(x), est défini comme suit:

$$V(x) = \{y \in X/x \neq yet(x, y) \in Uou(y, x) \in U\}$$

## 3.5 Graphe simple

Un graphe non orienté est dit simple s'il ne contient pas de boucles, de plus entre tout couple de sommets il y a au plus une arête.

## 3.6 Graphe complet

Dans le cas orienté : G est complet ssi $\forall x, y \in X, (x, y) \notin U \Rightarrow (y, x) \in U$.
Dans le cas non orienté : G est complet ssi $\forall x \neq y \in X, (x, y) \in E$.
Un graphe simple complet d'ordre n est noté $K_n$.

## 3.7 Graphe Régulier

Un graphe G est dit k-régulier si $\forall x \in G$, on a $d_G(x) = k$.

## 3.8 Graphe Biparti

G est dit biparti ssi l'ensemble de ses sommets X admet une partition en 2 sous ensembles $X_1$ et $X_2$ avec $X_1 \cap X_2 =$ et $X_1 \cup X_2 = X$.
Dans le cas orienté : $\forall (x, y) \in U \Rightarrow x \in X_1$ et $y \in X_2$.
Dans le cas non orienté : $\forall x, y \in E(x \in X_1$ et $y \in X_2)$ou$(x \in X_2$et $y \in X_1)$.
G est dit biparti complet ssi G est dit biparti et $\forall x \in X_1$ et $\forall y \in X_2 \Rightarrow (x, y) \in U$.

## 3.9 Graphe étiqueté

Un graphe étiqueté est un graphe (orienté ou non) dont les liaisons entre les sommets (arêtes ou arcs) sont affectées d'étiquettes (mot, lettre, symbole, etc...).

## 3.10 Graphe pondéré

Un graphe pondéré est un graphe étiqueté dont toutes les étiquettes sont des nombres réels positifs ou nuls. Ces nombres sont les poids des liaisons (arêtes ou arcs) entre les sommets.

## 3.11 Graphe eulérien

Un chemin (resp. chaîne) est dit **eulérien(ne)** s'il passe par toutes les arêtes (resp. arcs) juste une seule fois. Si ce chemin (resp. chaîne) est fermé, on parlera de circuit (resp. cycle) **eulérien**.
Un graphe est dit **eulérien** s'il possède un cycle (resp. circuit) eulérien.

## 3.12 Graphe hamiltonien

Un chemin (resp. chaîne) est dit **hamiltonien(ne)** s'il passe par tous les sommets une et une seule fois. Si ce chemin (resp. chaîne) est fermé (i.e. il existe une arête (resp. arc) reliant le sommet de départ au sommet d'arrivée), on parlera de circuit (resp. cycle) **hamiltonien**.
Un graphe est dit **hamiltonien** s'il possède un cycle (resp. circuit) hamiltonien.

## 3.13 Connexité

Un graphe est dit connexe s'il existe une chaîne joignant chaque paire de sommets x et y $x \neq y$.

### 3.13.1 Composante connexe

Soit un graphe $G = (X, E)$: La relation :
**{**$xRy \Leftrightarrow$ {**soit x = y / soit** $\exists$ **une chaîne reliant x et y}**
est une relation d'équivalence, les classes d'équivalence $X_1, X_2, ...X_k$ induites par R sur l'ensemble de sommets X engendre les composantes connexes de G.
G comporte alors k composantes connexes.
Un graphe qui contient une seule composante connexe est dit graphe connexe.

### 3.13.2 Graphe k-connexe

Un graphe G est dit k-connexe si et seulement si G est connexe d'ordre $n>k + 1$ et il n'existe pas d'ensemble $S \subset X$ de cardinal k  1 tel que le sous graphe engendré par X  S n'est pas connexe. En d'autres termes, en supprimant moins de k sommets, le graphe sera toujours connexe.

## 3.14 Multi-graphe

Un graphe non orienté qui n'est pas simple est un multi-graphe

## 3.15 Densité d'un graphe

La densité d'un graphe est le rapport entre le nombre de liens observés et le nombre de liens d'un graphe complet. Ainsi, elle varie entre 0 pour un graphe vide et 1 pour un graphe complet. Si chaque point d'un graphe est atteignable depuis n'importe quel point alors le graphe est connecté ou connexe.

# 4 Partie II

## 4.1 Matrice d'adjacence :

La matrice d'adjacence est une représentation du graphe à l'aide d'un tableau à deux dimensions où les nœuds sont indexés par les lignes et les colonnes. Chaque case de la matrice représente l'existence d'une arête ou d'un arc entre deux nœuds. Si une arête existe entre deux nœuds i et j, la case **M[i][j]** contiendra un "1" (ou la pondération de l'arc si le graphe est pondéré). Sinon, elle contiendra un "0". Dans le cas d'un graphe orienté, la matrice n'est pas symétrique. En revanche, pour un graphe non orienté, elle est symétrique par rapport à la diagonale principale.

**Caractéristiques :**

La matrice est carrée de taille N×NN×N où NN est le nombre de nœuds du graphe. Les éléments de la diagonale représentent les boucles (si elles existent). Si le graphe est pondéré, les "0" ou "1" sont remplacés par les poids des arêtes. La complexité de l'accès à une arête est O(1)O(1), mais la complexité de la recherche d'arêtes sortantes ou entrantes pour un nœud est O(N)O(N), ce qui rend cette représentation inefficace pour les graphes creux (c'est-à-dire avec peu d'arêtes).

### 4.2 Liste d'adjacence :

La liste d'adjacence est une autre manière de représenter un graphe. Elle consiste à stocker, pour chaque nœud, une liste (ou un ensemble) de ses voisins directs (arêtes sortantes). Cette représentation est particulièrement efficace pour les graphes creux, car elle permet de ne stocker que les arêtes existantes, ce qui réduit la quantité de mémoire utilisée par rapport à la matrice d'adjacence.

**Caractéristiques :**

Chaque nœud est représenté par un index dans un tableau, et chaque élément du tableau contient une liste des nœuds voisins du nœud correspondant. Pour les graphes orientés, on distingue les listes des successeurs et des prédécesseurs pour chaque nœud. L'accès à une arête entre deux nœuds est effectué en parcourant la liste des voisins d'un nœud, ce qui rend l'accès à une arête plus coûteux en termes de temps (O(k)O(k), où kk est le nombre de voisins).

### 4.3 Quand choisir chaque représentation ?

| **Critère** | **Matrice d'adjacence** | **Liste d'adjacence** |
|---|---|---|
| **Graphes denses** | Préférée (accès rapide aux arêtes) | Moins efficace (utilise plus de mémoire) |
| **Graphes creux** | Moins efficace (utilise trop de mémoire) | Préférée (utilise moins de mémoire) |
| **Accès rapide aux arêtes** | Préférée (temps $O(1)$ pour vérifier une arête) — Moins rapide (temps $O(k)$, avec $k$ voisins) | |
| **Représentation des graphes pondérés** | Idéale pour stocker les poids des arêtes | Moins efficace pour les graphes pondérés |
| **Graphes orientés** | Moins adaptée (symétrie des arêtes perdue) | Plus efficace (les successeurs et prédécesseurs sont séparés) |
| **Graphes très grands** | Peu efficace (surtout pour les graphes creux) | Très efficace en termes de mémoire |
| **Graphes dynamiques (ajout/suppression d'arêtes)** | Moins flexible (modifications lourdes) | Plus flexible (ajout/suppression rapide) |

Table 1: Comparaison entre la matrice d'adjacence et la liste d'adjacence

## 5 Partie III

On a utilisé la représentation avec la liste d'adjacence

## 5.1 Construction d'un graphe orienté/non orienté

**Output :** Un graphe $G$ construit à partir des données saisies par l'utilisateur.
*Graphe* = *Enregistrement*{ T:Tableau de noeud; type: chaine de caractere; }
*noeud* = *Enregistrement*{ sommet: entier; succ: l̂iste; }
**Fonction** *ConstruireGraphe(): Graphe* `// Demander le type de graphe`
Ecrire("Choisir le type du graphe :");
Ecrire("* Orienté : 1");
Ecrire("* Non orienté : 0");
*type* ← Lire(); `// liser le type de graphe`
**si** *type* = 1 **alors**
  Ecrire("Construisez votre graphe orienté :");
  *G.type* ← orienté;
**sinon**
  Ecrire("Construisez votre graphe non orienté :");
  *G.type* ← non orienté;
**fin**
$G$ ← InitialiserGraphe(*G.type*);
`// Saisie des sommets`
Ecrire("Entrez le nombre de sommets :");
$n$ ← Lire();
**pour** $i \leftarrow 1$ ***à*** $n$ **faire**
  `// Liser le sommets un par un`
  Ecrire("Entrez le nom du sommet ", $i$, " :");
  *sommet* ← Lire();
  $G.T[i]$ ← *sommet*;
**fin**
`// Saisie des arêtes`
Ecrire("Entrez le nombre d'arêtes :");
$m$ ← Lire();
Ecrire("Entrez les arêtes sous forme de paires (ex: A B) :");
**pour** $i \leftarrow 1$ ***à*** $m$ **faire**
  `// Liser les aretes`
  Ecrire("Entrez l'arête ", $i$, " :");
  $u$ ← Lire();
  $v$ ← Lire();
  **si** *!existe*$(u, G.T)$ ***ou*** *!existe*$(v, G.T)$ **alors**
    Ecrire("Erreur : ", $u$, " ou ", $v$, " n'existe pas. Réessayez.");
  **sinon**
    AjouterLien($G$, $u$, $v$);
  **fin**
**fin**
**return** $G$;

**Algorithme 1 :** Complexite = O(n+m) — n:nombre des sommets, m:nobre des aretes

## 5.2 Affichage du graphe

**Input :** Un graphe $G$ représenté par une liste d'adjacence $G.T$, où chaque case contient les voisins (**non orienté**) ou les listes *succ* et *pred* (**orienté**). Un booléen *G.type* indique si le graphe est orienté.
**Output :** Affichage de la liste d'adjacence du graphe.
**Procedure** *AfficherGraphe(G)* `// Afficher le titre de la représentation du graphe`
Afficher("Représentation du Graphe :");
**si** *G.type = non orienté* **alors**
  **pour** $i \leftarrow 1$ ***à*** *taille(G.T)* **faire**
    *voisins* $\leftarrow$ ListeVoisins($G.T[i].succ$);
    Ecrire($G.T[i].sommet$ + " → " + Joindre(",", *voisins*));
  **fin**
**sinon** `// Le graphe est orienté`
  **pour** $i \leftarrow 1$ ***à*** *taille(G.T)* **faire**
    *successeurs* $\leftarrow$ ListeVoisins($G.T[i].succ$);
    Ecrire($G.T[i].sommet$ + " → " + Joindre(",", *successeurs*));
  **fin**
**fin**

**Algorithme 2 :** Complexite = O(n+m) — n:nombre des sommets, m:nobre des aretes

## 5.3 Calculer la densité du graphe

**Input :** Un graphe $G$ représenté par une liste d'adjacence $G.T$, où chaque case contient les voisins (**non orienté**) ou les listes *succ* et *pred* (**orienté**). Un booléen *G.type* indique si le graphe est orienté.
**Output :** La densité du graphe.
**Fonction** *DensiteGraphe(G): réel*
*ordre* $\leftarrow$ taille($G.T$) `// Calculer le nombre de sommets du graphe`
**si** *ordre* $\leq 1$ **alors**
  **return** 0 `// Si un seul sommet ou aucun, densité = 0`
**fin**
*max_aretes* $\leftarrow$ *ordre* $\times$ (*ordre* $- 1$) `// Nombre maximal d'arêtes possibles`
*num_aretes* $\leftarrow 0$ `// Initialiser le nombre d'arêtes à 0`
**pour** $i \leftarrow 1$ ***à*** *ordre* **faire**
  **si** *G.type = non orienté* **alors**
    *num_aretes* $\leftarrow$ *num_aretes*+ tailleListe($G.T[i]$);
  **sinon**
    *num_aretes* $\leftarrow$ *num_aretes*+ tailleListe($G.T[i].succ$);
  **fin**
**fin**
**si** *G.type = non orienté* **alors**
  *num_aretes* $\leftarrow$ *num_aretes* $\div 2$ `// Diviser par 2 pour graphe non orienté`
  *max_aretes* $\leftarrow$ *max_aretes* $\div 2$;
**fin**
*densite* $\leftarrow$ *num_aretes* $\div$ *max_aretes*;
**return** *densite*;

**Algorithme 3 :** Complexite = O(n+m) — n:nombre des sommets, m:nobre des aretes

## 5.4 Calculer le degré du graphe

**Input :** Un graphe $T$ représenté par un tableau, où chaque case pointe vers une liste chaînée de voisins (**non orienté**) ou deux listes chaînées (**orienté**) : *succ* et *pred*.
**Output :** Le degré du graphe (degré maximal d'un sommet).
**Fonction** *fonction(G:graphe,n:entier):reel*
*degree* $\leftarrow 0$ `// Initialiser le degré du graphe à 0`
**pour** $i \leftarrow 1$ ***à*** $n$ **faire**
  **si** *G.type != orienté* **alors**
    $num\_aretes \leftarrow$ tailleListe($G.T[i].succ$);
  **sinon** `// Le graphe est orienté`
    $num\_aretes \leftarrow$ tailleListe($G.T[i].succ$) + tailleListe($T[i].pred$);
  **fin**
  **si** $degree < num\_aretes$ **alors**
    $degree \leftarrow num\_aretes$;
  **fin**
**fin**
**return** *degree*;

**Algorithme 4 :** Complexite = O(n+m) — n:nombre des sommets, m:nobre des aretes

## 5.5 Vérifier si le graphe est complet

**Input :** Un graphe $G$ représenté par une liste d'adjacence $G.T$, où chaque case contient les voisins (**non orienté**) ou les listes *succ* et *pred* (**orienté**). Un booléen *G.type* indique si le graphe est orienté.
**Output : Vrai** si le graphe est complet, **Faux** sinon.
**Fonction** *EstComplet(G): booléen*
*ordre* $\leftarrow$ taille($G.T$) `// Calculer le nombre de sommets du graphe`
**si** $ordre \leq 1$ **alors**
  **return Vrai** `// Un graphe avec 0 ou 1 sommet est complet`
**fin**
**si** *G.type* = *orienté* **alors**
  **pour** $i \leftarrow 1$ ***à*** $n$ **faire**
    `// Vérifier que chaque sommet est connecté à tous les autres`
    **si** $tailleListe(G.T[i].succ) \neq ordre - 1$ ***ou*** $tailleListe(G.T[i].pred) \neq ordre - 1$ **alors**
      **return Faux**
    **fin**
  **fin**
**sinon**
  **pour** $i \leftarrow 1$ ***à*** $n$ **faire**
    `// Vérifier que chaque sommet est connecté à tous les autres`
    **si** $tailleListe(G.T[i].succ) \neq ordre - 1$ **alors**
      **return Faux**
    **fin**
  **fin**
**fin**
**return Vrai** `// Le graphe est complet`

**Algorithme 5 :** Complexite = O(n+m) — n:nombre des sommets, m:nobre des aretes

## 5.6 Trouver un sous-graphe complet maximal

**Input :** Un graphe $G$ représenté par une liste d'adjacence $G.T$, où chaque case contient les voisins (**non orienté**) ou les listes *succ* et *pred* (**orienté**). Un booléen $G.type$ indique si le graphe est orienté.

**Output :** Un sous-graphe complet maximal (clique) du graphe $G$.

```
// Fonction pour vérifier si un ensemble de sommets forme une clique
Fonction EstClique(G, Noeuds): booléen
pour i ← 1 à taille(G.T) faire
    pour j ← i + 1 à taille(G.T) faire
        si G.type = orienté alors
            si !existe(j, G.T[i].succ) ou !existe(i, G.T[j].pred) alors
                return Faux;
            fin
        sinon
            si !existe(j, G.T[i].succ) alors
                return Faux;
            fin
        fin
    fin
fin
return Vrai;
Fonction SousGrapheComplet(G): Graphe
n ← taille(G.T);
Sommets ← listeClés(G.T);
// initialiser un tableau vide
CliqueMaximale ← {};
// Tester toutes les combinaisons de sommets, de taille décroissante
pour k ← 1 à taille(G.T) faire
    combinason ← Combinaisons(Somments, k);
    pour i ← 1 a taille(Combinaison) faire
        si EstClique(G, Combinaison[i]) alors
            return Combinaison[i] // Retourner la première clique maximale trouvée
        fin
    fin
fin
return CliqueMaximale;
```

**Algorithme 6 :** Complexite = $O(n^2)|n : ordredugraphe$

## 5.7 Recherche de tous les chemins entre un nœud a et un nœud b

**Input :** Un graphe $G$ représenté par une liste d'adjacence $G.T$, un nœud de départ *debut*, un nœud de destination $fin$, et un chemin courant *chemin* (initialement vide). Un booléen $G.type$ indique si le graphe est orienté.
**Output :** Une liste contenant tous les chemins de *debut* à $fin$.
**Fonction** *TrouverTousChemins(G, debut, fin, chemin)*
**si** *vide(chemin)* **alors**
  $chemin[0] \leftarrow \{debut\}$ `// Initialiser le chemin avec le nœud de départ`
**fin**
**sinon**
  $chemin \leftarrow chemin + debut$ `// Ajouter le nœud actuel au chemin`
**fin**
**si** $debut = fin$ **alors**
  **return** *chemin* `// Chemin trouvé : renvoyer une liste contenant ce chemin`
**fin**
**si** *!existe(G.T, debut)* **alors**
  **return** `// Aucun chemin possible si le nœud de départ n'existe pas`
**fin**
$paths \leftarrow \{\}$ `// Initialiser la liste des chemins`
$voisins \leftarrow$ ListeVoisins($G.T[debut]$) `// Obtenir tous les voisins`
**pour** $i \leftarrow 1$ ***a*** *taille(voisins)* **faire**
  **si** *!existe(voisins[i], chemin)* **alors**
    *nouveaux_chemins* $\leftarrow$ TrouverTousChemins($G$, $voisins[i]$, $fin$, *chemin*);
    **pour** $j \leftarrow 1$ ***a*** *taille(nouveaux_chemins)* **faire**
      Ajouter(*nouveau_chemin*, *paths*);
    **fin**
  **fin**
**fin**
**return** *paths* `// Renvoyer tous les chemins trouvés`

**Algorithme 7 :** Complexite = O(n!) — n:ordre du graphe

## 5.8 Recherche du chemin le plus court entre deux nœuds a et b

**Input :** Un graphe $G$ représenté par une liste d'adjacence $G.T$. Les nœuds *debut* et $fin$ spécifient le chemin recherché.
**Output :** Le chemin le plus court entre *debut* et $fin$.
**Fonction** *CheminPlusCourt(G, debut, fin): Tableau*
*chemins* $\leftarrow$ TrouverTousChemins($G$, *debut*, $fin$);
*chemin_plus_court* $\leftarrow \{\}$;
**pour** $i \leftarrow 1$ ***a*** *taille(chemins)* **faire**
  **si** *taille(chemins[i])* ¡ *taille(chemin_plus_court)* **alors**
    *chemin_plus_court* $\leftarrow$ *chemin*;
  **fin**
**fin**
**return** *chemin_plus_court*;

**Algorithme 8 :** Complexite O(n!) — n:ordre du graphe

### 5.9 Recherche de composantes (fortement) connexes à partir d'un nœud a.

```python
def composantes_k_connexes(self, k=4):
    def dfs_k_connexes(node, visited, component):
        """
        Perform DFS to find k-connected components.
        """
        visited.add(node)
        component.add(node)
        if self.oriente:
            neighbors = self.listeAdj[node].get('succ', set())
        else:
            neighbors = self.listeAdj[node]

        for neighbor in neighbors:
            if neighbor not in visited and len(neighbors) >= k:
                dfs_k_connexes(neighbor, visited, component)

    visited = set()
    components = []
    for node in self.listeAdj:
        if node not in visited and len(self.listeAdj[node]) >= k:
            component = set()
            dfs_k_connexes(node, visited, component)
            components.append(component)

    return components
```

**Algorithme 9 :** Complexite = O(n+m) — n:nombre des sommets, m:nobre des aretes

### 5.10 Trouver tous les cycles/circuits dans le graphe

```python
def find_cycles(self):
    """
    Trouve tous les cycles dans un graphe (orienté ou non orienté).
    :return: Liste des cycles trouvés.
    """
    graph = self.listeAdj

    def dfs(node, parent, visite, path):
        """
        Recherche DFS pour détecter les cycles.
        """
        visite.add(node)
        path.append(node)

        if self.oriente == True:  # Graphe orienté
            voisins = graph[node].get('succ', set())
        else:  # Graphe non orienté
            voisins = graph[node]

        for voisin in voisins:
            if voisin not in visite:  # Continuer la recherche DFS
                dfs(voisin, node, visite, path)
            elif voisin != parent:  # Cycle détecté
                # Extraire le cycle
                cycle_start_index = path.index(voisin)
                cycle = path[cycle_start_index:]
                # Stocker le cycle sous forme canonique (trié)
                cycle_sorted = tuple(sorted(cycle))
                if cycle_sorted not in unique_cycles:
                    unique_cycles.add(cycle_sorted)

        # Retour en arrière
        path.pop()

    visite = set()
    unique_cycles = set()  # Utilisé pour éliminer les doublons

    for node in graph:
        if node not in visite:
            dfs(node, None, visite, [])

    # Retourner les cycles sous forme de liste de listes
    return [list(cycle) for cycle in unique_cycles]

```

**Algorithme 10 :** Complexite = O(n+n.log(n)) — n:nombre des sommets

## 5.11 Vérifier si le graphe contient un cycle/circuit hamiltonien

```python
def hamiltonian_cycle(self):
    def backtrack(node, visited, path):
        path.append(node)
        visited.add(node)

        if len(visited) == len(self.listeAdj):  # All nodes visited
            if path[0] in (self.listeAdj[node] if not self.oriente else self.listeAdj[node].get('succ', set())):
                return path + [path[0]]  # Return to start node
            visited.remove(node)
            path.pop()
            return None

        neighbors = self.listeAdj[node] if not self.oriente else self.listeAdj[node].get('succ', set())
        for neighbor in neighbors:
            if neighbor not in visited:
                result = backtrack(neighbor, visited, path)
                if result:
                    return result

        visited.remove(node)
        path.pop()
        return None

    for start_node in self.listeAdj:
        result = backtrack(start_node, set(), [])
        if result:
            return result
    return None
```

**Algorithme 11 :** Complexite = O(n!) — n:nombre des sommets

## 5.12 Vérifier si le graphe contient une k-clique (k donné)

```python
def has_k_clique(self, k):
    from itertools import combinations
    nodes = list(self.listeAdj.keys())

    for subset in combinations(nodes, k):
        # Check for k-clique in directed or undirected graph
        if all(
            (node2 in self.listeAdj[node1] if not self.oriente else node2 in self.listeAdj[node1].get('succ', set()))
            and (node1 in self.listeAdj[node2] if not self.oriente else node1 in self.listeAdj[node2].get('succ', set()))
            for node1 in subset for node2 in subset if node1 != node2
        ):
            return True
    return False
```

**Algorithme 12 :** Complexite = O(n!) — n:nombre des sommets

### 5.13 Trouver une clique maximale dans un graphe G

```python
def maximal_clique(self):
    def is_clique(nodes):
        return all(
            (node2 in self.listeAdj[node1] if not self.oriente else node2 in self.listeAdj[node1].get('succ', set()))
            and (node1 in self.listeAdj[node2] if not self.oriente else node1 in self.listeAdj[node2].get('succ', set()))
            for node1 in nodes for node2 in nodes if node1 != node2
        )

    def backtrack(curr_clique, candidates):
        if not candidates:
            return curr_clique

        max_clique = curr_clique[:]
        for node in candidates:
            new_clique = curr_clique + [node]
            if is_clique(new_clique):
                remaining_candidates = [n for n in candidates if n != node]
                candidate_clique = backtrack(new_clique, remaining_candidates)
                if len(candidate_clique) > len(max_clique):
                    max_clique = candidate_clique

        return max_clique

    nodes = list(self.listeAdj.keys())
    return backtrack([], nodes)
```

**Algorithme 13 :** Complexite $= O(2^n) | n : nombredessommets$

# 6 Evaluation expérimentale :

| Taille du graphe | Densité des arêtes | Temps d'exécution (Cycle Hamiltonien) | Temps d'exécution (Clique k) |
|---|---|---|---|
| 50 | 0.10 | 0.014456 | 0.002347 |
| 50 | 0.30 | 0.024235 | 0.003890 |
| 50 | 0.50 | 0.043768 | 0.005234 |
| 100 | 0.10 | 0.148904 | 0.024736 |
| 100 | 0.30 | 0.232125 | 0.045620 |
| 100 | 0.50 | 0.380129 | 0.076559 |
| 200 | 0.10 | 0.587461 | 0.107235 |
| 200 | 0.30 | 0.921384 | 0.156909 |
| 200 | 0.50 | 1.432232 | 0.276745 |

Table 2: Temps d'exécution des algorithmes pour différentes tailles de graphes et densités

En observant les résultats du tableau des temps d'exécution pour les algorithmes du cycle hamiltonien et de la clique k, quelque points importants peuvent être constatés :

## 6.1 Augmentation des temps d'exécution avec la taille du graphe :

Cycle Hamiltonien : Le temps d'exécution augmente significativement avec la taille du graphe. Par exemple, pour un graphe de taille 50, le temps d'exécution est relativement court, mais il devient beaucoup plus long pour un graphe de taille 200. Cela est attendu, car le problème du cycle hamiltonien est NP-complet, et donc la complexité croît exponentiellement avec la taille du graphe. Clique k : De même, l'algorithme de recherche de clique k montre une augmentation du temps d'exécution à mesure que le nombre de nœuds dans le graphe augmente. Cependant, la croissance est souvent moins rapide que pour le cycle hamiltonien, bien que cela dépende fortement de la densité des arêtes et de la structure du graphe.

## 6.2 Effet de la densité des arêtes :

Lorsque la densité des arêtes est faible (0.1), les algorithmes sont généralement plus rapides, car il y a moins de connexions à explorer. Cela rend les parcours et les vérifications plus rapides. À densité moyenne (0.3), les temps d'exécution augmentent, car il y a plus de possibilités de parcours ou de connexions à vérifier. À densité élevée (0.5), le graphe devient plus connecté, ce qui rend les algorithmes de recherche plus coûteux en termes de temps, car il y a plus de chemins à explorer pour vérifier les cycles ou les cliques.

## 6.3 Complexité des algorithmes :

Les algorithmes NP-complets comme le cycle hamiltonien montrent une forte dépendance à la taille du graphe et à la densité. À mesure que ces deux facteurs augmentent, le temps d'exécution augmente de façon exponentielle. La recherche de clique k est également coûteuse, mais moins exponentielle que celle du cycle hamiltonien, bien que cela varie en fonction de la taille du sous-ensemble de nœuds considéré (k).

# 7 Utilisation des graphes en bio-informatique :

Les graphes sont largement utilisés en bio-informatique pour modéliser et résoudre des problèmes complexes dans les domaines de la génomique, de la protéomique, de l'évolution et d'autres disciplines biologiques. Voici une présentation des applications les plus courantes des graphes en bio-informatique

## 7.1 Alignement et assemblage de séquences (Génomique)

### 7.1.1 Assemblage de génomes :

Graphes de De Bruijn : Utilisés pour assembler des séquences d'ADN à partir de fragments courts (reads). Chaque k-mer (sous-séquence de longueur kk) devient un sommet, et les arêtes représentent les chevauchements entre k-mers. Cette méthode permet de reconstruire des génomes complets à partir de données issues de séquençage.

Graphes de chevauchement : Dans cette méthode, chaque nœud est un fragment de séquence, et les arêtes représentent les chevauchements entre fragments. Utilisé pour assembler des génomes lorsque les séquences sont longues.

### 7.1.2 Alignement de séquences :

Graphes d'alignement multiple : Les séquences génomiques ou protéiques sont représentées dans un graphe où les chemins partagent des sous-séquences similaires. Permet d'effectuer des alignements multiples efficaces pour étudier les similarités et les divergences entre séquences.

## 7.2 Analyse des réseaux de gènes et de protéines

### 7.2.1 Réseaux d'interactions protéines-protéines (PPI) :

Représente les interactions biologiques entre protéines. Les protéines sont des nœuds, et les arêtes indiquent les interactions. Utilisé pour : Identifier les protéines essentielles dans un réseau. Étudier les interactions complexes responsables de processus biologiques (ex. signalisation cellulaire).

### 7.2.2 Réseaux de régulation génique :

Modélisent les interactions entre les gènes et les régulateurs transcriptionnels. Permettent d'étudier les relations de cause à effet dans l'expression des gènes.

#### 7.2.3 Réseaux métaboliques :

Les métabolites et les réactions enzymatiques sont représentés sous forme de graphe. Utilisés pour analyser les voies métaboliques, identifier des cibles thérapeutiques, ou modéliser les flux métaboliques.

### 7.3 Étude de l'évolution et des relations phylogénétiques

#### 7.3.1 Graphes phylogénétiques :

Modélisent les relations évolutives entre espèces ou séquences. Deux structures principales: Arbres phylogénétiques: Arbres enracinés ou non enracinés pour représenter les ancêtres communs. Réseaux phylogénétiques: Utilisés lorsque des événements complexes comme les transferts horizontaux de gènes ou les hybridations rendent les relations plus complexes qu'un arbre.

#### 7.3.2 Détection d'homologies :

Les graphes sont utilisés pour trouver des gènes orthologues (gènes provenant d'un ancêtre commun) ou paralogues (duplications au sein d'une espèce).

### 7.4 Analyse de données omiques

#### 7.4.1 Réseaux d'expression génique :

Modélisent les corrélations entre les niveaux d'expression de différents gènes à partir de données transcriptomiques. Permettent d'identifier des modules co-exprimés, souvent associés à des fonctions biologiques spécifiques.

#### 7.4.2 Réseaux épigénétiques :

Étudient les interactions entre les marqueurs épigénétiques (par exemple, méthylation de l'ADN, modifications des histones).

#### 7.4.3 Analyse de données multi-omiques :

Intègrent différents types de données biologiques (génomique, transcriptomique, protéomique, métabolomique) dans un graphe pour révéler des interactions complexes et découvrir des biomarqueurs.

### 7.5 Recherche de motifs et analyse de structures

#### 7.5.1 Motifs biologiques :

Utilisés pour identifier des structures répétées (comme des motifs de régulation dans les promoteurs de gènes). Par exemple, les graphes peuvent être utilisés pour détecter des motifs dans les interactions protéiques ou dans l'ADN.

#### 7.5.2 Structures de protéines :

Les protéines peuvent être modélisées comme des graphes: Les atomes ou acides aminés sont des nœuds. Les liaisons chimiques ou interactions (ex. ponts hydrogène) sont des arêtes. Utilisés pour étudier les propriétés structurelles et fonctionnelles des protéines.

### 7.6 Applications en médecine et pharmacologie

#### 7.6.1 Réseaux de maladies :

Permettent de modéliser les relations entre gènes, protéines, et maladies. Utilisés pour identifier les gènes responsables de pathologies ou pour étudier les mécanismes sous-jacents aux maladies complexes.

#### 7.6.2 Réseaux de médicaments :

Les molécules thérapeutiques et leurs cibles biologiques (protéines ou gènes) sont représentées sous forme de graphes. Aide à identifier des interactions médicamenteuses ou à découvrir de nouvelles cibles thérapeutiques.

#### 7.6.3 Analyse de graphes bipartis :

Représente les relations entre deux ensembles, par exemple: Gènes-maladies. Médicaments-cibles. Patients-phénotypes.

### 7.7 Chemins et flux biologiques

#### 7.7.1 Recherche de plus courts chemins :

Utilisés pour modéliser et analyser les voies biologiques. Exemple: Identifier les chemins minimaux dans un réseau métabolique pour prédire les flux biologiques.

#### 7.7.2 Flux maximaux :

Étudiés dans les réseaux métaboliques pour optimiser la production de biomolécules ou comprendre les goulots d'étranglement dans des systèmes biologiques.

### 7.8 Exemples Pratiques

**BLAST (Basic Local Alignment Search Tool):** Utilise des graphes pour aligner des séquences et identifier les similitudes.
**Cytoscape:** Logiciel d'analyse et de visualisation de réseaux biologiques.
**STRING:** Base de données qui intègre les interactions protéiques pour construire des réseaux PPI.

## 8 Conclusion

L'étude expérimentale menée sur les algorithmes du cycle hamiltonien et de la clique k a mis en évidence une corrélation directe entre la complexité temporelle de ces algorithmes et la taille ainsi que la densité des graphes. Comme attendu pour des problèmes NP-complets, le temps d'exécution augmente de manière significative avec la croissance du nombre de nœuds et le nombre d'arêtes.

Les résultats obtenus confirment l'importance de choisir des algorithmes adaptés à la taille et à la structure des graphes étudiés. Pour les graphes de grande taille et de haute densité, des algorithmes heuristiques ou approchés pourraient être envisagés pour obtenir des solutions sous-optimales mais en un temps de calcul raisonnable.

En ce qui concerne l'application des graphes en bio-informatique, nous avons pu constater la diversité des problèmes qui peuvent être modélisés et résolus à l'aide de cette structure de données. Des réseaux d'interactions protéiques aux graphes phylogénétiques, les graphes offrent un cadre puissant pour analyser les données biologiques et découvrir de nouvelles connaissances.

Les résultats de cette étude soulignent l'importance de continuer à développer des algorithmes efficaces pour l'analyse de graphes de grande taille, notamment dans le contexte de l'explosion des données biologiques. De plus, l'intégration de l'apprentissage automatique et de l'intelligence artificielle pourrait permettre de développer de nouvelles approches pour résoudre des problèmes complexes en bio-informatique.