REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENE

Faculté d'Informatique

Filière
Informatique

Spécialité
Bio-Informatique

Thème

Implémentation et analyse de complexité d'algorithmes sur les graphes

Module : ALGO Avanceé et Complexité

Présenté par :

- BEKKOUCHE Mustapha Yasser
- SMAIL Mehrez

Année Universitaire : 2022-2023

Table des matières

Index des figures

Index des tableaux

Introduction générale

Pour résoudre de nombreux problèmes, nous sommes amenés à dessiner des graphes, c'est-à-dire des points (appelés sommets) reliés deux à deux par des lignes (appelées arcs ou arêtes). Ces graphes font abstraction des détails non pertinents pour la résolution du problème et permettent de se focaliser sur les aspects importants.

Quelques exemples de modélisation par des graphes

Réseaux de transport

Un réseau de transport (routier, ferroviaire, métro, etc) peut être représenté par un graphe dont les sommets sont des lieux (intersections de rues, gares, stations de métro, etc) et les arcs indiquent la possibilité d'aller d'un lieu à un autre (par un tronçon de route, une ligne de train ou de métro, etc). Ces arcs peuvent être valués, par exemple, par leur longueur, la durée estimée pour les traverser, ou encore un coût. Etant donné un tel graphe, nous pourrons nous intéresser, par exemple, à la résolution des problèmes suivants :

— Quel est le plus court chemin (en longueur, en durée, ou encore en coût) pour aller d'un sommet à un autre ?

— Est-il possible de passer par tous les sommets sans passer deux fois par un même arc ?

— Peut-on aller de n'importe quel sommet vers n'importe quel autre sommet ?

Réseaux sociaux

Les réseaux sociaux (LinkedIn, Facebook, etc) peuvent être représentés par des graphes dont les sommets sont des personnes et les arêtes des relations entre ces personnes. Etant donné un tel graphe, nous pourrons nous intéresser, par exemple, à la résolution des problèmes suivants :

— Combien une personne a-t-elle de relations ?

— Quelles sont les communautés (sous-ensemble de personnes en relation directe les unes avec les autres) ?

— Par combien d'intermédiaires faut-il passer pour relier une personne à une autre ?

Un grand nombre de problèmes réels ont recours à la représentation de données sous forme de graphes, orientés ou non orientés. Par exemple, modéliser un réseau routier, un réseau d'ordinateurs, un automate d'états fini, un scénario de jeu à états, la planification de projets, …etc. Dans ce travail, on s'intéresse à l'implémentation des graphes en mémoire, les opérations usuelles sur les graphes et leurs complexités.

L'implémentation et l'analyse de complexité d'algorithmes sur les graphes sont des domaines importants de l'informatique qui consistent à mettre en œuvre des algorithmes sur des graphes et à évaluer leur efficacité en termes de temps de calcul et de consommation de mémoire. Les graphes sont des structures de données couramment utilisées pour représenter des relations entre différents éléments, et les algorithmes sur les graphes sont utilisés pour résoudre divers problèmes, tels que le plus court chemin entre deux points ou la détection de cycles dans un graphe.

Il est important de bien implémenter ces algorithmes afin de garantir qu'ils fonctionnent de manière efficace et de façon à minimiser le temps de calcul et la consommation de mémoire. De plus, l'analyse de complexité permet de comprendre à quelle vitesse un algorithme va fonctionner en fonction de la taille des données sur lesquelles il est appliqué. Cela est crucial pour pouvoir choisir l'algorithme le plus approprié pour résoudre un problème donné.

Objectifs du Travail

L'objectif de notre étude est Implémentation et analyse de complexité d'algorithmes sur les graphes.

Le mémoire est organisé comme suit. Dans la partie 1, Rappels sur la représentation de graphes en mémoire (différentes représentations) et établir les principales définitions de la structure de graphe. Dans la partie 2, Nous passons en les représentations d'un graphe en mémoire. Dans la partie 3, nous présentons des opérations sur les graphes avec calcul de complexité. Dans la partie 4, Evaluation et comparaison de complexités théoriques et expérimentales. Pour finir, nous tirons des conclusions (par rapport aux complexités dans les deux représentations, la taille et la densité du graphe.

Partie 1

Définitions

Graphe : Un graphe est un ensemble de points, dont certaines paires sont directement reliées par un (ou plusieurs) lien(s), Ces points sont nommés nœuds ou sommets. Ces liens sont nommés arêtes.

Graphe non-orienté : chaque arête peut-être parcourue dans les deux sens.

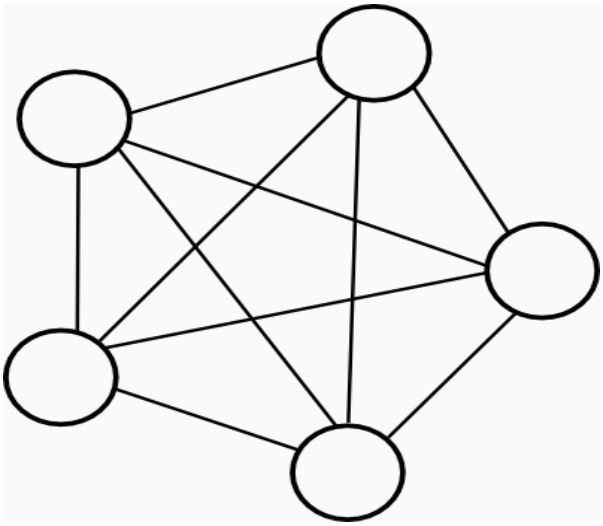


Figure 1 : Graphe non-orienté

Graphe orienté : les arêtes sont à sens unique. On les représente donc avec une flèche sur les dessins. D'ailleurs, le terme employé n'est plus arête, mais arc. Cette distinction est importante, car nombre d'algorithmes ne fonctionnent tout simplement pas sur des graphes orientés.

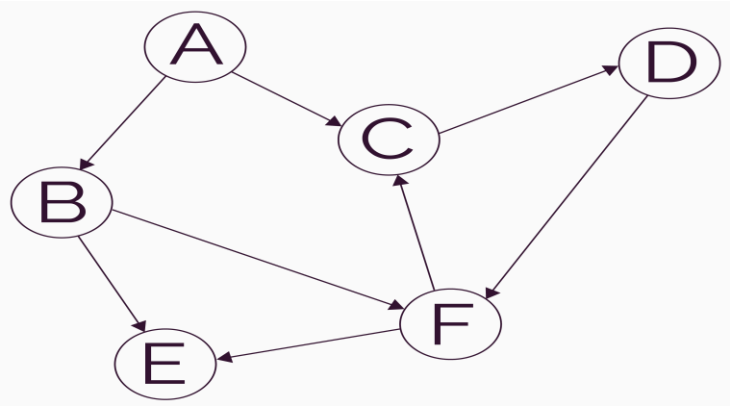


Figure 2 : Graphe orienté

Arête : Ligne qui joint deux sommets consécutifs, distincts ou non, d'un graphe non orienté.

Arc : Ligne qui joint deux sommets consécutifs, distincts ou non, d'un graphe orienté.

Cycle : Dans un graphe non orienté, un cycle est une suite d'arêtes consécutives (chaine simple) dont les deux sommets extrémités sont identiques.

Circuit : Dans les graphes orientés, un cycle est une suite d'arêtes consécutives (chaine simple) dont les deux sommets extrémités sont identiques.

Graphe pondéré : Un graphe pondéré ou un réseau est un graphe où chaque arête porte un nombre (son poids). Ces poids peuvent représenter par exemple des coûts, des longueurs ou des capacités, en fonction du problème traité. Ces graphes sont fréquents dans divers contextes, comme le problème de plus court chemin ou le problème du voyageur de commerce.

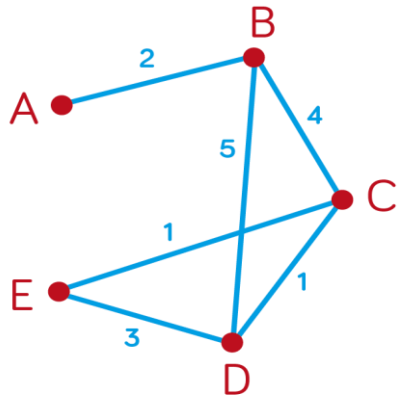


Figure 3 : Graphe pondéré

Degré d'un nœud : Le degré d'un nœud correspond au nombre d'arêtes reliées à ce nœud. Dans le cas d'un graphe orienté, le degré entrant d'un nœud est le nombre d'arcs qui aboutissent à ce nœud, et le degré sortant le nombre d'arcs qui partent de ce nœud.

Boucle : Une boucle est une arête qui relie un nœud à lui même.

Lien double : Un lien double caractérise l'existence de plusieurs arêtes entre deux nœuds donnés.

multi-graphe : Un graphe possédant l'une ou l'autre de ces caractéristiques (boucle, lien double) est dit **multi-graphe**. Un graphe ne possédant aucune des deux est dit **graphe simple**.

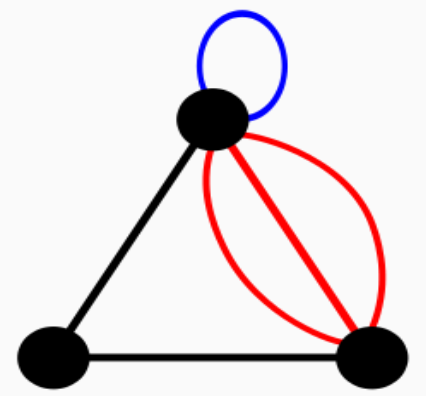


Figure 4 : Multigraphe

Graphe simple : Un graphe simple est un graphe sans boucle ni arête multiple. Il n'y a alors d'arêtes qu'entre des sommets distincts, et entre deux sommets il y a au plus une arête.

Densité d'un graphe : La densité d'un graphe correspond au rapport du nombre d'arêtes sur le nombre total d'arêtes possibles. C'est donc un réel compris entre 0 et 1. Cette caractéristique influe sur le choix de sa représentation.
Une densité de 0 correspond à un graphe sans arêtes où tout les sommets sont isolés.
Une densité de 1 correspond à un graphe complet : chaque nœud est relié à chaque autre nœud.

Graphe complet : Un graphe complet est un graphe dont chaque sommet est relié directement à tous les autres sommets.

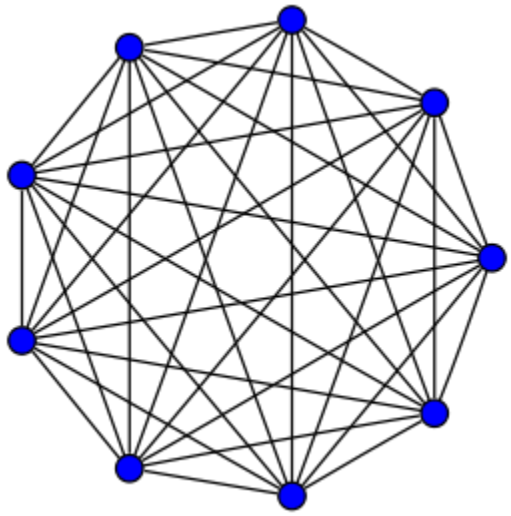


Figure 5 : Graphe complet à 9 nœuds

Graphe connexe : Un graphe est connexe quand tout sommet peut être relié à tout autre sommet par une arête ou une suite d'arêtes.

Composante connexe : Dans un graphe non orienté, une composante connexe est un sous-graphe induit maximal connexe, c'est-à-dire un ensemble de points qui sont reliés deux à deux par un chemin. On peut ainsi regrouper les sommets d'un graphe selon leur appartenance à la même composante connexe.

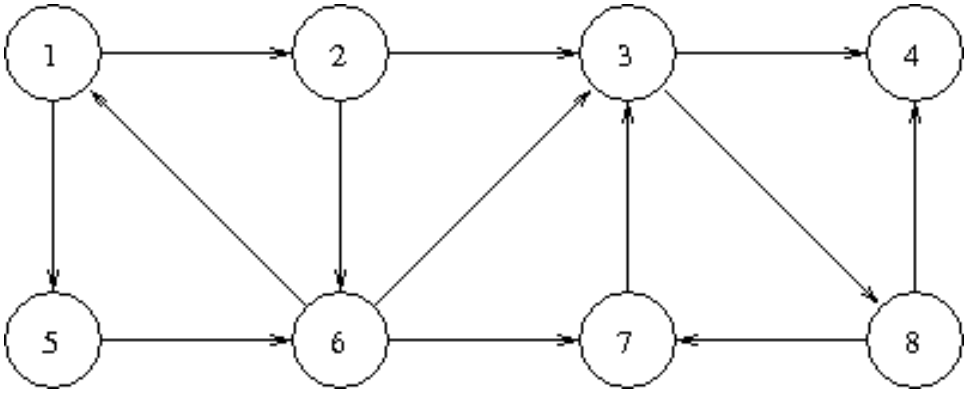


Figure 6 : Composante connexe

Parcours profondeur : Le parcours en profondeur d'un graphe à partir d'un sommet consiste à suivre les arêtes arbitrairement, en marquant les sommets déjà visités pour ne pas les visiter à nouveau**.**

Graphe biparti : Un graphe est biparti s'il est possibles de former deux partitions (comprendre, de le couper en deux morceaux distincts) de ses sommets de façon à ce que chaque arête passe d'une partition à l'autre, sans que jamais une arête ne relie deux nœuds dans la même partition.

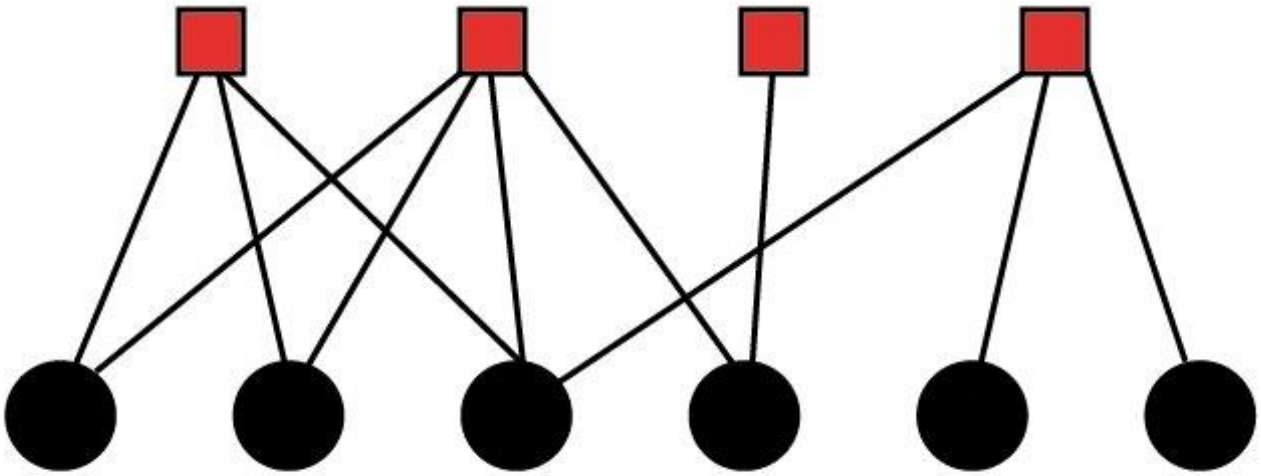


Figure 7 : Graphe biparti

Partie 2

Les représentations d'un graphe en mémoire

2.1 Introduction

Maintenant, nous avons une idée bien plus précise de ce qu'est un graphe. Mais il n'est pas destiné à rester un outil purement théorique, Il faut pouvoir résoudre des problèmes avec, et donc implémenter des algorithmes qui travaillent dessus.

Nous devons donc trouver une structure de données adaptée pour le stocker, qui soit économe en mémoire, et qui permette aux algorithmes de l'exploiter rapidement.

Chaque nœud est stocké dans un tableau, une liste, ou n'importe quelle autre structure de données plus complexe (aucune contrainte particulière à ce propos, à vous de choisir la plus adaptée). Il est composé de propriétés, comme par exemple une position, une lettre, une valeur ou n'importe quelle entité plus complexe, qui dépend du problème (au cas par cas).

La difficulté consiste donc à lister intelligemment les arêtes entre les nœuds.

On résout ce problème avec une liste d'adjacence ou une matrice d'adjacence. La quasi totalité des implémentations d'un graphe sont des variantes de ces deux structures de données.

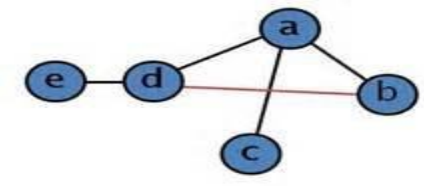
La liste et la matrice ne sont pas seulement des structures de données, elles sont aussi une représentation du graphe.

La matrice et la liste contiennent suffisamment d'informations pour définir le graphe, au même titre que la représentation géométrique à base de sommets et d'arêtes.

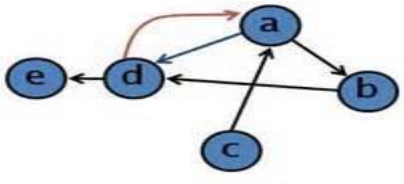
Mais contrairement à cette dernière, qui est plus intuitive et plus adaptés aux êtres humains, la matrice et la liste peuvent être aisément implémentées sous forme de structure de données et utilisées par un programme.

2.2 La matrice d'adjacence

La matrice d'adjacence est un tableau en deux dimensions. Chacune des dimensions est indexée par les nœuds du graphe (typiquement de 0 à N−1). A l'intersection de chaque ligne et colonne on trouve un nombre : il vaut 1 si une arête relie les deux nœuds indexés par les coordonnées de la case, et 0 sinon.



	a	b	c	d	e
a	0	1	1	1	0
b	1	0	0	1	0
c	1	0	0	0	0
d	1	1	0	0	1
e	0	0	0	1	0



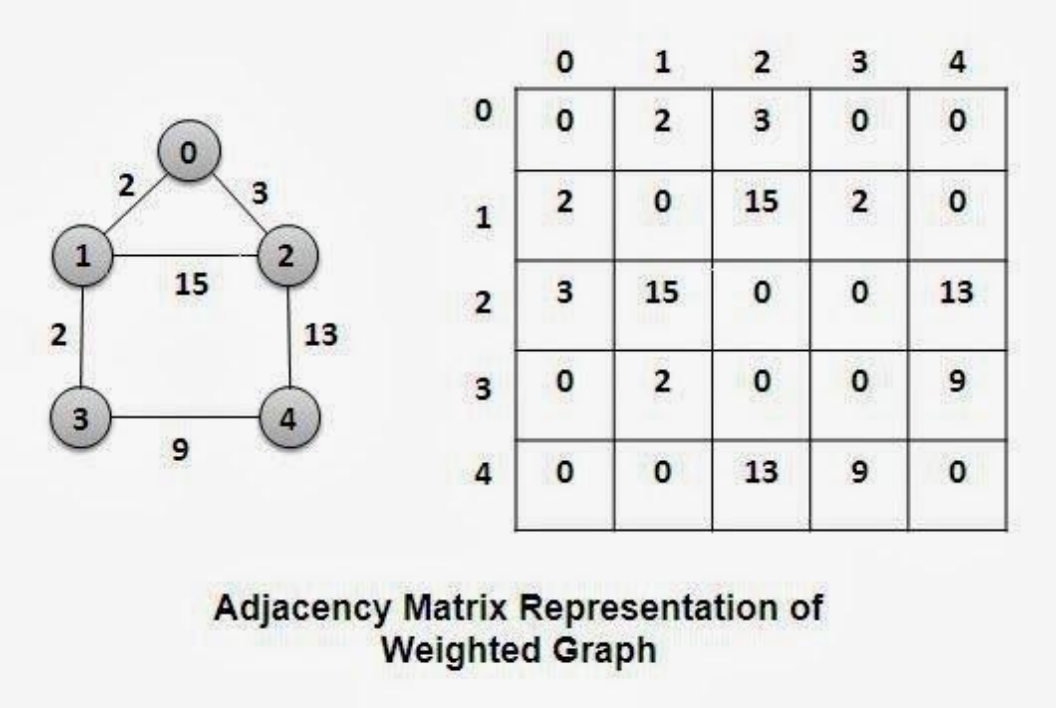
	a	b	c	d	e
a	0	1	0	1	0
b	0	0	0	1	0
c	1	0	0	0	0
d	1	0	0	0	1
e	0	0	0	0	0

Figure 8 : Matrice d'adjacense

On observe plusieurs choses intéressantes.

• Les boucles reliant un nœud à lui même sont sur la diagonale de la matrice.
• Cette matrice est symétrique par rapport à sa diagonale dans un graphe non-orienté (puisque si A est relié à B, alors B est relié à A).
• Si le graphe ne comporte aucune arête, alors c'est la matrice nulle.
• Si le graphe est creux, alors la matrice le sera aussi.
• Pour N nœuds, cette matrice est de taille N×N. Soit une complexité de O(N2) en mémoire. Si votre langage le permet, vous pouvez stocker chacun des booléens sur un bit; il en résulte une consommation en mémoire de N28 octets exactement, ce qui est très compact (incompressible dans le cas général).
• Vous pouvez itérer sur tout les voisins d'un nœud en O(N).
• En outre, à tout moment vous pouvez déterminer si deux nœuds sont voisins (c'est à dire reliés par une arête) en O(1).
• Pour finir les ajouts et retraits d'arêtes se font en O(1).

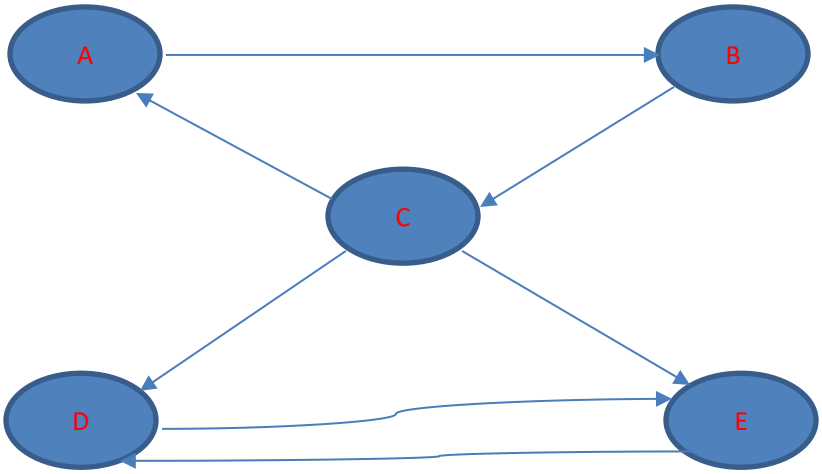
Si le graphe est pondéré, vous pouvez remplacer les booléens par des nombres correspondant à la pondération de chaque arête. Et vous fixez une valeur spéciale pour signifier l'absence d'arête (−1, 0 ou ∞ par exemple, selon les cas).



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

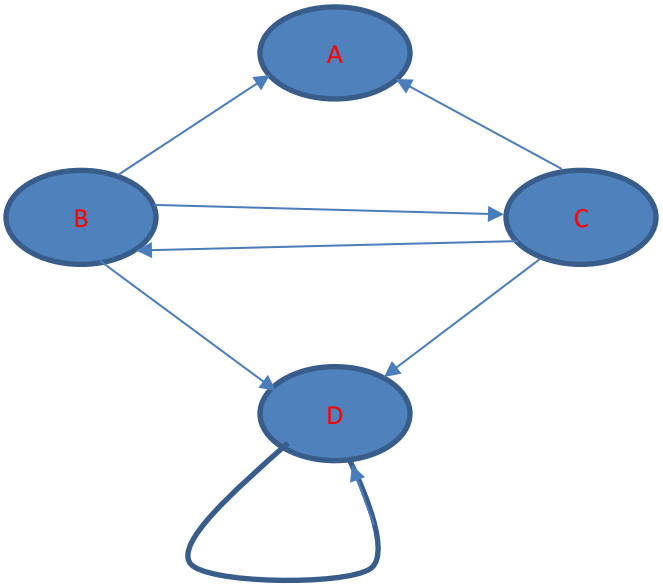
Figure 9 : Matrice d'adjacence de graphe pondéré

2.3 Les représentations d'un graphe en mémoire



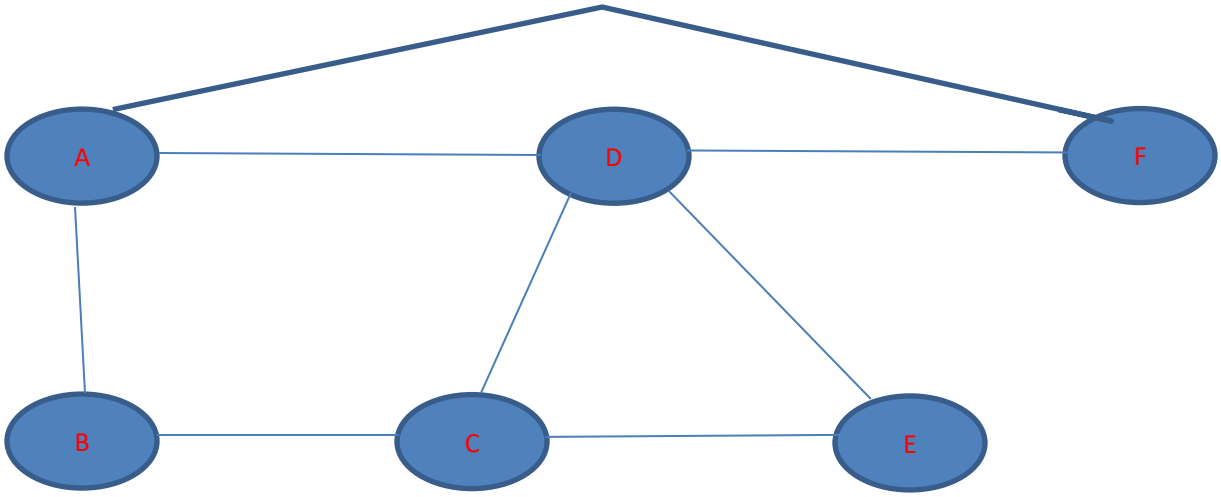
• La matrice d'adjacence :

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	0	0
C	1	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0



• La matrice d'adjacence :

	A	B	C	D
A	0	0	0	0
B	1	0	1	1
C	1	1	0	1
D	0	0	0	1



• La matrice d'adjacence :

	A	B	C	D	E	F
A	0	1	0	1	0	1
B	1	0	1	0	0	0
C	0	1	0	1	1	0
D	1	1	0	0	1	1
E	0	0	1	1	0	0
F	1	0	0	1	0	0

2.4 Conclusion

Un petit tableau récapitulatif fera l'affaire :

Opérations	Matrice d'adjacence
Retirer une arête	O(1)
Ajouter une arête	O(1)
Itérer sur les voisins d'un nœud	O(N)
Tester si deux nœuds sont voisins	O(1)
Complexité mémoire	$O(N^2)$

Tableau 1 : Complexité

Partie 3

3.1 Description des algorithmes et calcul de complexité

• Construction d'un graphe orienté/non orienté :

```
N = 10
        M= np.full(shape=(N, N), fill_value=0, dtype=int)
        for i in range(N):
            for j in range(i + 1, N):
                M[i][j] = int(random.randint(0, 1))
                M[j][i] = M[i][j]
```

- Complexité : **O(N^2)**

• Affichage du graphe :

```
graph = pydot.Dot("mehrez", graph_type='graph')
        for i in range(N):
            graph.add_node(pydot.Node(str(i + 1)))
            for j in range(i + 1, N):
                if M[i][j] == 1:
                    d = j + 1
                    s = i + 1
                    graph.add_edge(pydot.Edge(str(s), str(d), color="blue"))
```

- Complexité : **O(N^2)**

• Calculer la densité du graphe :

```
  def densite(self ):
        s = 0
        x = len(M)
        for i in range(x):
            for j in range(x):
                s = s + M[i][j]
        d = s / (x * (x - 1))
        d="{0:.5f}".format(d)
        self.textEdit.setText(str(d))
```

- Complexité : **O(N^2)**

• Recherche d'un nœud a dans le graphe (afficher le nœud et ses liens) :

```
def RN(self):
    start = time.time()
    Z=self.lineEdit.text()
    graph.get_node(Z)[0].set_color("red")
    n=len(M)
    for j in range(n):
        l = int(Z) - 1
        if M[l][j] == 1:
            d = j + 1
            graph.get_node(str(d))[0].set_color("yellow")
    graph.write_png("recher.png")
    GG = QPixmap("recher.png")
    self.label_9.setPixmap(GG)
    end = time.time()
    temp = end - start
    temp = "{0:.5f}".format(temp)
    self.textEdit_22.setText(str(temp))
```

- Complexité : **O(N)**

• Recherche d'un chemin entre un nœud a et un nœud b :

```
def ch(self):
    start = time.time()
    X = nx.drawing.nx_pydot.from_pydot(graph)
    a=self.lineEdit_2.text()
    b=self.lineEdit_3.text()
    r = nx.shortest_path(X, a, b, weight="weight")
    for i in r:
        graph.get_node(i)[0].set_color("red")
    for i in range(len(r) - 1):
        graph.get_edge(r[i], r[i + 1])[0].set_color("red")
    graph.write_png("ch.png")
    GG = QPixmap("ch.png")
    self.label_9.setPixmap(GG)
    end = time.time()
    temp = end - start
    temp = "{0:.5f}".format(temp)
    self.textEdit_22.setText(str(temp))
```

- Complexité : **$O(N^2)$**

• Recherche du chemin le plus court entre deux nœuds a et b :

```
def chc(self):
    start = time.time()
    X = nx.drawing.nx_pydot.from_pydot(graph)
    a=self.lineEdit_4.text()
    b=self.lineEdit_5.text()
    r = nx.shortest_path(X, a, b, weight="weight")
    for i in r:
        graph.get_node(i)[0].set_color("red")
    for i in range(len(r) - 1):
        graph.get_edge(r[i], r[i + 1])[0].set_color("red")
    graph.write_png("chc.png")
```

- Complexité : **O(nlogn)**

• Ajouter un nœud a avec ses liens :

```
def Ajouter_un_noeud(self):
    start = time.time()
    s = self.lineEdit_6.text()
    graph.add_node(pydot.Node(s))
    a = self.lineEdit_7.text()
    for i in range(len(a)):
        b = a[i]
        graph.add_edge(pydot.Edge(s, b, color="yellow"))
    graph.write_png("Ajouter_un_nœud.png")
    GG = QPixmap("Ajouter_un_nœud.png")
    self.label_9.setPixmap(GG)
    end = time.time()
    temp = end - start
    temp = "{0:.5f}".format(temp)
    self.textEdit_22.setText(str(temp))
```

- Complexité : **O(nb lien)**

• Supprimer un nœud a avec ses liens :

```
def supN(self):
    start = time.time()
    N=len(M)
    a=self.lineEdit_8.text()
    print(a)
    for i in range(N):
        M[int(a) - 1][i] = 0
        M[i][int(a) - 1] = 0
    supn = pydot.Dot("supn", graph_type='graph')
    for i in range(N):
        if i != int(a) - 1:
            supn.add_node(pydot.Node(str(i + 1)))
        for j in range(i + 1, N):
            if M[i][j] == 1:
                d = j + 1
                s = i + 1
                supn.add_edge(pydot.Edge(str(s), str(d), color="blue"))
    supn.write_png("supn.png")
    GG = QPixmap("supn.png")
    self.label_9.setPixmap(GG)
    end = time.time()
    temp = end - start
    temp = "{0:.5f}".format(temp)
```

- Complexité : **$O(N^2)$**

• Ajouter un lien (arc ou arête) entre deux nœuds existants :

```
def Ajouter_un_lien(self):
    start = time.time()
    a=self.lineEdit_9.text()
    b=self.lineEdit_10.text()
    M[int(a) - 1][int(b) - 1] = 1
    M[int(b) - 1][int(a) - 1] = 1
    graph.add_edge(pydot.Edge(str(a), str(b), color="yellow"))
    graph.write_png("Ajouter_un_lien.png")
    GG = QPixmap("Ajouter_un_lien.png")
    self.label_9.setPixmap(GG)
    end = time.time()
    temp = end - start
    temp = "{0:.5f}".format(temp)
```

- Complexité : **O(1)**

3.2 Evaluation expérimentale

Tableau d'évaluation expérimentale de la complexité des algorithmes en considérant des graphes de taille 10, 20, 30, 50 :

Taille n	10	20	30	50
Construction(Tps)	0.0014	0.0243	0.0342	0.0564
Affichage (Tps)	0.00398	0.04955	0.05403	0.08229
la densité (Tps)	0.00018	0.00027	0.00147	0.00327
Recherche d'un nœud (Tps)	0.04023	0.26718	1.15526	10.78798
Recherche d'un chemin (Tps)	0.04327	0.26667	1.16424	11.01702
Recherche du chemin court (Tps)	0.04385	0.29089	1.19473	10.82299
Ajouter un nœud a (Tps)	0.44444	0.29597	1.20679	8.11026
Supprimer un nœud a (Tps)	0.07620	0.28251	1.14320	9.24575
Ajouter un lien (Tps)	0.05130	0.34383	1.16186	11.42266

Tableau 2 : Evaluation expérimentale

3.3 L'interfaces d'implémentation

Dans cette section, nous présentons l' interfaces principales du code d'implémentation :

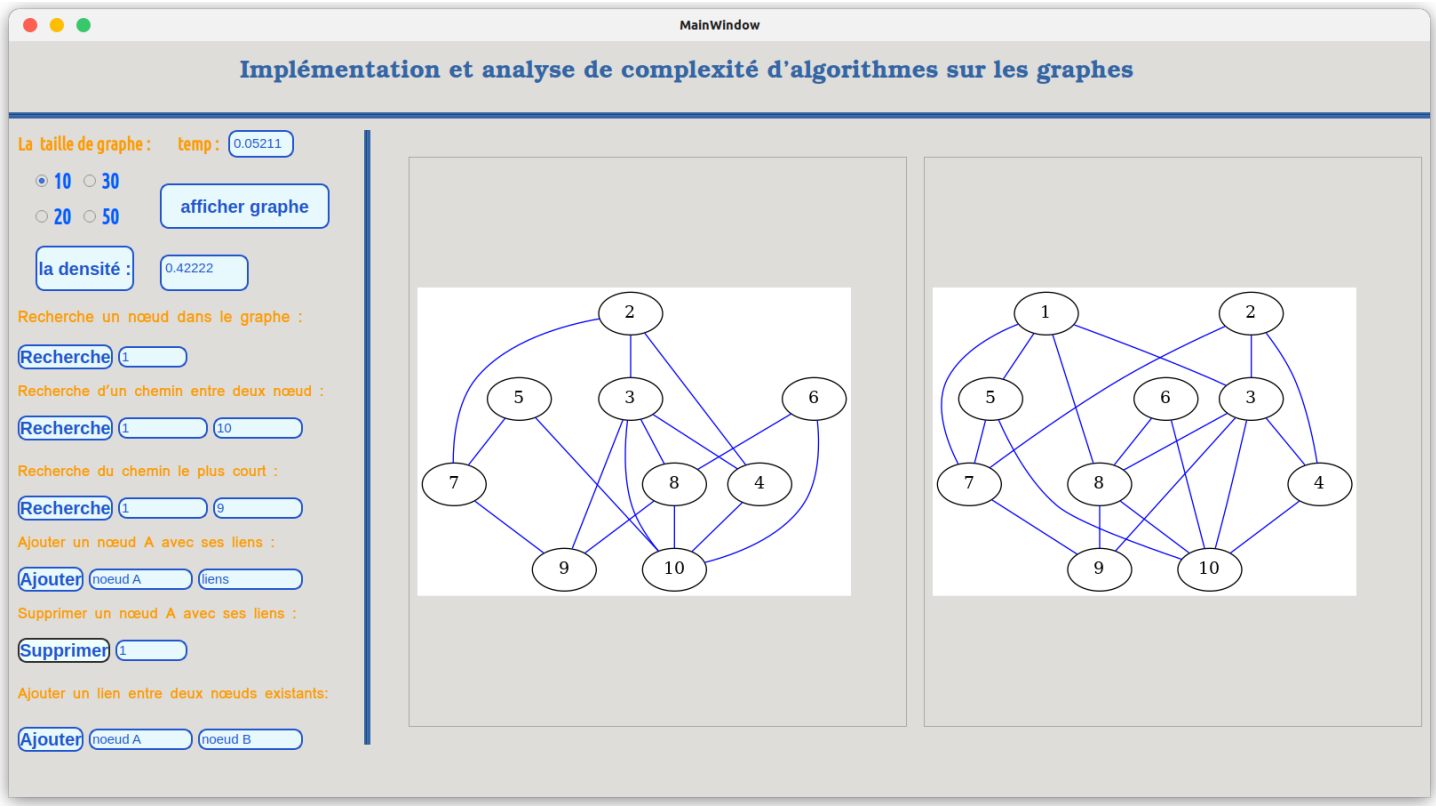


Figure 10 : L'interface d'implémentation

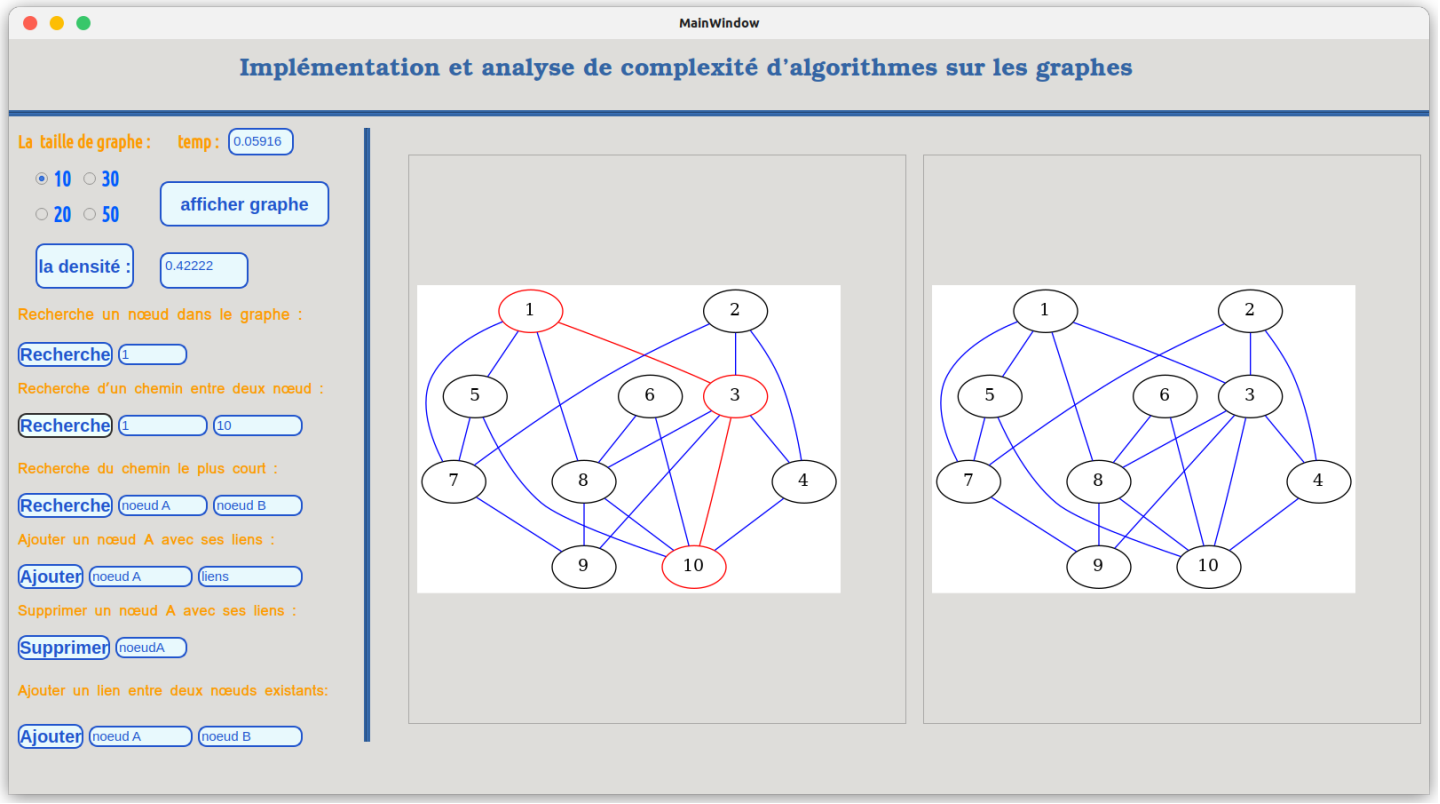


Figure 11 : L'interface d'implémentation

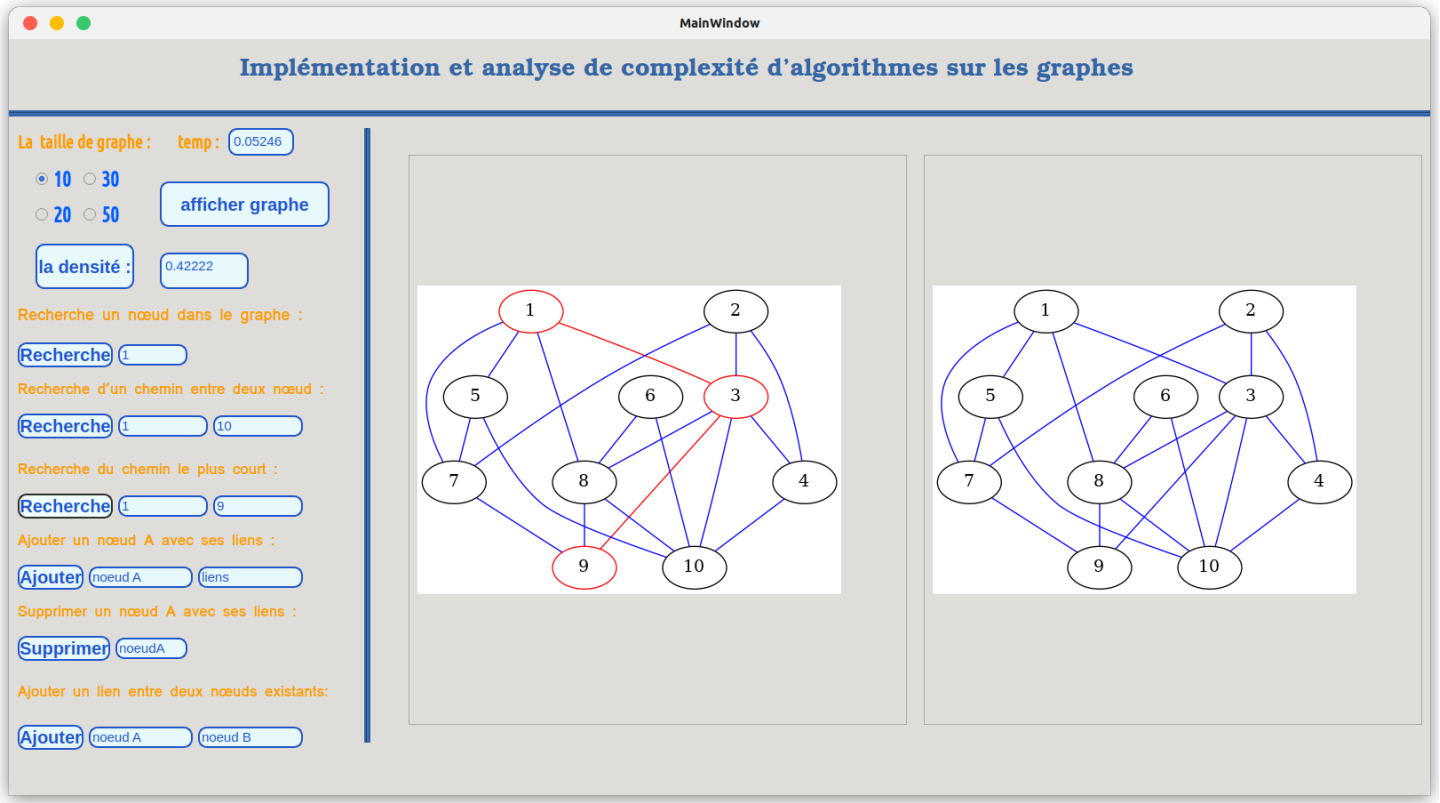


Figure 12 : L'interface d'implémentation

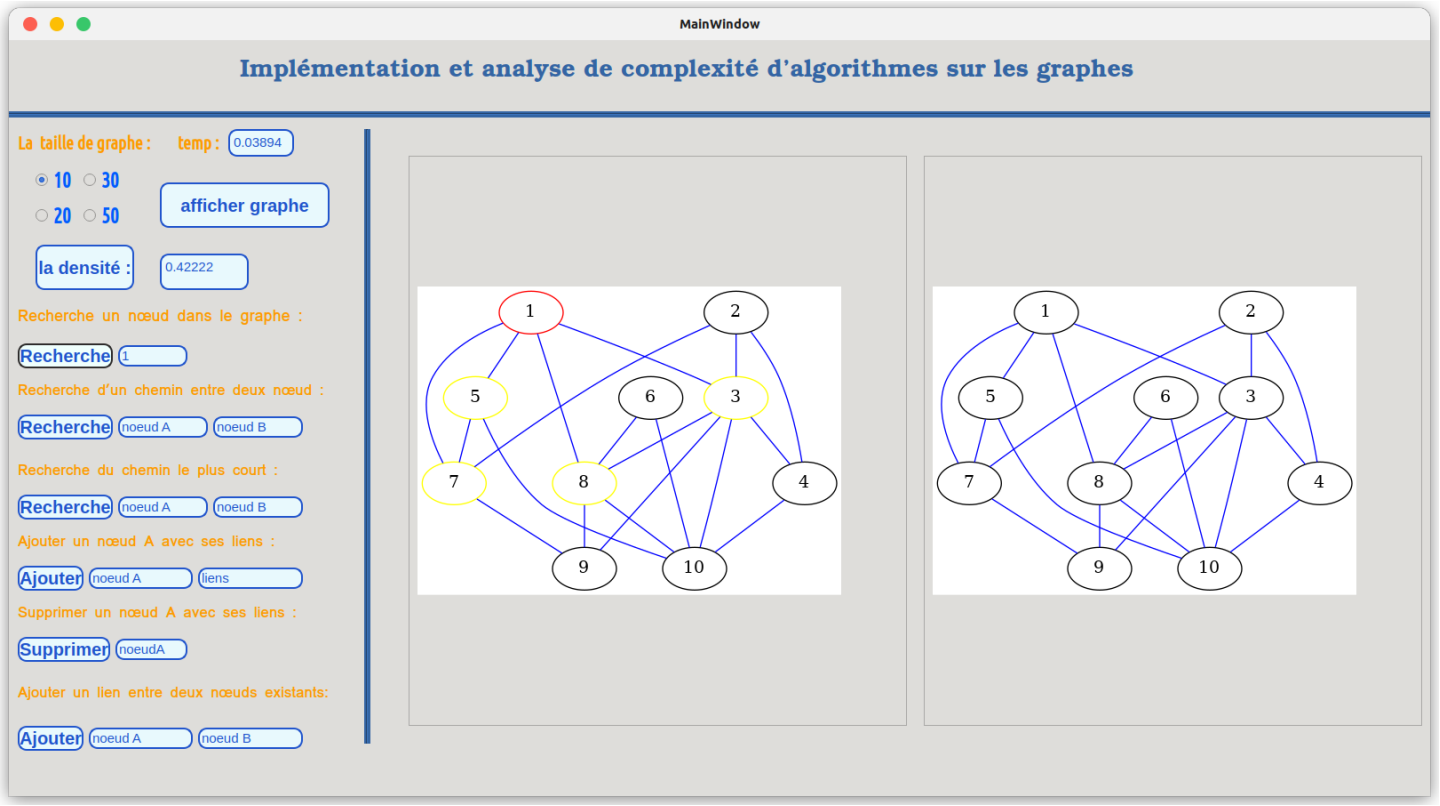


Figure 13 : L'interface d'implémentation

Conclusion

Il est généralement plus efficace d'utiliser une représentation en liste d'adjacence plutôt qu'une représentation en matrice d'adjacence pour représenter un graphe, surtout si le graphe est peu dense (c'est-à-dire s'il a peu d'arêtes par rapport à son nombre maximal possible d'arêtes). En effet, dans une représentation en liste d'adjacence, chaque noeud ne stocke que les informations sur ses voisins, ce qui peut être très efficace lorsque le graphe est peu dense et que chaque noeud a peu de voisins.

Dans une représentation en matrice d'adjacence, chaque noeud doit stocker des informations sur l'ensemble des noeuds du graphe, même s'il n'est pas connecté à eux. Cela peut être inefficace lorsque le graphe est peu dense, car beaucoup d'espace de stockage sera perdu en stockant des informations inutiles sur des noeuds non connectés.

En revanche, une représentation en matrice d'adjacence peut être plus efficace que une représentation en liste d'adjacence lorsque le graphe est densément connecté et que chaque noeud a beaucoup de voisins. Dans ce cas, la représentation en matrice d'adjacence permet d'accéder rapidement aux informations sur les voisins d'un noeud en utilisant un simple accès en temps constant à une case de la matrice.

En ce qui concerne l'influence de la taille et de la densité du graphe sur la complexité des algorithmes, plus le graphe est grand et dense, plus il sera difficile de travailler dessus et plus les algorithmes prendront du temps à s'exécuter. Cependant, cela dépend également de la nature de l'algorithme en question et de sa complexité. Certaines opérations, comme la recherche d'un chemin entre deux noeuds, peuvent être très coûteuses en temps lorsque le graphe est grand et dense, alors que d'autres opérations, comme l'ajout ou la suppression de noeuds ou d'arêtes, peuvent être plus rapides. Il est donc important de prendre en compte à la fois la taille et la densité du graphe ainsi que la nature de l'algorithme lorsque vous évaluez la complexité des algorithmes sur des graphes.