République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique

Département Informatique

Spécialité : Bio-Informatique

Projet Module ALGO Av. et Complexité

Thème

Implémentation et analyse de complexité d'algorithmes sur les graphes

---

Réalisé par :

LAIB Ayoub

TABLE DES MATIERES

TABLE DES FIGURES

1. Introduction Générale

Dans le paysage informatique contemporain, la représentation et la manipulation efficace des données sont cruciales pour résoudre une multitude de problèmes complexes. L'une des structures de données les plus polyvalentes et puissantes est le graphe, qui trouve des applications dans divers domaines, tels que la modélisation de réseaux, la planification de projets et la résolution de problèmes complexes.

Ce projet se penche sur l'essence des graphes en informatique, explorant leur représentation en mémoire et mettant en lumière les opérations fondamentales. Au-delà de la simple implémentation, l'objectif principal est de comprendre les implications algorithmiques associées à chaque opération, avec un accent particulier sur la comparaison des différentes complexités théoriques et expérimentales.

2. Objectifs du Travail

Ce projet ambitieux s'articule autour d'objectifs clairement définis visant à explorer et maîtriser les subtilités des graphes informatiques. Notre démarche se concentrera sur plusieurs axes essentiels :

1. **Compréhension Conceptuelle :** Établir une compréhension solide des concepts fondamentaux liés aux graphes, tels que les nœuds, les arêtes, les cycles, et les différentes formes de graphes.
2. **Exploration des Représentations :** Investiguer de manière approfondie les deux principales méthodes de représentation des graphes en mémoire, en examinant les caractéristiques distinctives de la matrice d'adjacence et de la liste d'adjacence.
3. **Maîtrise des Opérations Fondamentales :** Mettre en œuvre et analyser les complexités algorithmiques associées aux opérations de base sur les graphes. De la construction à la vérification de propriétés spécifiques, notre objectif est de fournir une vision holistique des différentes facettes des manipulations graphiques.
4. **Comparaison des Complexités :** Conduire une évaluation comparative des complexités théoriques et expérimentales pour chaque opération, offrant ainsi une perspective pratique sur les avantages et les inconvénients de chaque approche.

3. Partie I - Définitions

Dans cette première partie, Nous allons établir les définitions essentielles qui nous permettront de comprendre les subtilités des relations et des connexions dans le monde des graphes.

3.1. Qu'est-ce qu'un graphe ?

Un graphe est une structure de données composée de nœuds (ou sommets) reliés par des arêtes. Ces arêtes peuvent être dirigées ou non dirigées.

3.2. Définitions :

- **Graphe orienté** : Un graphe orienté est un graphe dont les arêtes sont orientées (fléchées).
- **Graphe non orienté** : Un graphe non orienté est un ensemble de points, appelés sommets, reliés par des lignes, appelées arêtes.
- Un arc reliant deux sommets est appelé une arête.
- **Cycle** : Un cycle est une chaîne dont les extrémités coïncident, et qui est composée d'arêtes toutes distinctes.
- **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.
- **Le degré d'un nœud** : Le degré d'un nœud est la quantité d'arêtes qui lui sont incidentes.
- **Arc Entrant** : Un arc qui arrive à un nœud dans un graphe orienté.
- **Arc Sortant** : Un arc qui part d'un nœud dans un graphe orienté.
- **Graphe simple** : Un graphe simple est un graphe sans boucle dont chaque couple de sommets est relié par au plus une arête.
- **Multi graphe** : Un multi graphe est un graphe qui peut avoir plusieurs arêtes entre une paire de sommets.
- **Graphe eulérien** : On appelle graphe eulérien un graphe que l'on peut dessiner sans jamais lever le crayon et sans passer deux fois par la même arête.
- **Arbre** : Un graphe non orienté, connexe et acyclique.
- **Densité d'un graphe** : La densité d'un graphe est définie par le rapport entre le nombre d'arêtes (ou d'arcs) divisé par le nombre d'arêtes (ou d'arcs) possibles.
- **Graphe connexe** : Un graphe est connexe si on peut relier deux quelconques de ses sommets par une chaîne.

- **Composante Connexe** : Un sous-graphe maximal dans un graphe non orienté, où chaque paire de nœuds est connectée par un chemin.
- **Profondeur** : Le nombre d'arêtes dans le plus court chemin entre deux nœuds.
- **Graphe complet** : Un graphe complet est un graphe simple dont tous les sommets sont adjacents.

Remarque :

- Tout graphe complet est connexe.
- Si un graphe n'est pas connexe, il ne peut pas être complet.

- **Sous graphe** : Un sous graphe d'un graphe G est un graphe constitué de certains sommets de G et de toutes les arêtes qui les relient.

PS : Toutes ces définitions proviennent du cours de mon enseignante, Mme Bouibeda, dans le module de théorie des graphes en 2ème année de licence.

4. Partie II – Les représentations d'un graphe en mémoire

4.1. Matrice d'Adjacence :

4.1.1. Définition :

La matrice d'adjacence est comme une grille où chaque ligne et chaque colonne représentent un nœud du graphe. Si deux nœuds sont reliés par une arête, alors le numéro correspondant dans la grille sera 1, sinon il sera 0.

4.1.2. Caractéristiques Observées :

Boucles : Les boucles reliant un nœud à lui-même sont situées sur la diagonale de la matrice.

Symétrie : La matrice est symétrique par rapport à sa diagonale dans un graphe non orienté (a est relié à b, alors b est relié à a).

Graphe Sans Arête : Si le graphe ne comporte aucune arête, la matrice est nulle.

Graphe Creux : Si le graphe est creux, la matrice contiendra plusieurs zéros.

4.1.3. Avantage :

Facile à implémenter pour les graphes denses.

Accès rapide pour vérifier la présence d'une arête entre deux nœuds.

4.1.4. Inconvénients :

Utilise beaucoup d'espace pour les graphes creux.

Les opérations d'ajout ou de suppression d'arêtes peuvent être coûteuses.

4.2. Liste d'Adjacence :

4.2.1. Définition :

La liste d'adjacence est comme une liste de voisins pour chaque nœud du graphe. Chaque nœud a une liste qui indique avec quels autres nœuds il est directement connecté.

4.2.2. Caractéristiques Observées :

Représentation d'un Nœud : Chaque nœud est représenté dans un tableau.

Il contient un lien vers la liste de ses voisins (ou arêtes le reliant directement à ses voisins).

Graphe Orienté : Pour un graphe orienté, différenciation entre la liste des successeurs et la liste des prédécesseurs de chaque nœud.

Densité du Graphe : La densité du graphe (nombre de liens) est souvent le critère décisif pour choisir entre la matrice d'adjacence et la liste d'adjacence.

4.2.3. Avantage :

Efficace pour les graphes creux.

Économie d'espace.

4.2.4. Inconvénients :

Un accès plus lent pour vérifier la présence d'une arête entre deux nœuds.

Peut-être plus complexe à mettre en œuvre pour certains algorithmes.

4.3.Choix entre Matrice et Liste d'Adjacence :

- **Graphe Dense :**

Matrice d'Adjacence peut être préférable en raison de l'accès rapide.

- **Graphe Creux :**

Liste d'Adjacence est souvent plus efficace en termes d'espace.

- **Opérations Fréquentes d'Ajout/Suppression d'Arêtes :**

Liste d'Adjacence est généralement plus pratique.

4.4.La représentation des graphes :

4.4.1. Graphe G1 :

Un graphe orienté avec des liens reliant les nœuds a, b, c, d, e, et f dans une séquence spécifique.

Matrice d'adjacence

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

Liste d'adjacence

[a] → [b]

[b] → [c]

[c] → [a, d, e]

[d] → [e]

[e] → [d, f]

[f] → [b]

4.4.2. Graphe G2 :

Un graphe orienté avec des connexions partant des nœuds 1, 2, 3 vers le nœud 4, établissant une structure hiérarchique.

Matrice d'adjacence

	1	2	3	4
1	0	0	0	1
2	1	1	0	0
3	1	1	1	1
4	1	1	1	0

Liste d'adjacence

[1] → [4]

[2] → [1, 2]

[3] → [1, 2, 3, 4]

[4] → [1, 2, 3]

4.4.3. Graphe G3 :

Un graphe non orienté avec des relations entre les nœuds a, b, c, d, e, et f, créant une toile de connexions sans direction spécifique.

Matrice d'adjacence

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

Liste d'adjacence

[A] → [B, F]

[B] → [A, C]

[C] → [B, D]

[D] → [C, E]

[E] → [D, F]

[F] → [A, E]

5. Partie III – Opérations sur les graphes et complexité

5.1.Les algorithmes des opérations

Construction d'un graphe orienté/non orienté

Matrice d'adjacence

```
// Fonction pour créer un graphe
FONCTION Graphe()
  DEBUT
      ÉCRIRE ("nombre de nœuds")
      LIRE (n)
      ALLOUER g [n][n]
      POUR i DE 0 À n-1 FAIRE
         ALLOUER g[i] [n]
         POUR j DE 0 À n-1 FAIRE
            g[i][j] = 0
         FIN POUR
      FIN POUR
   FIN
// Procédure pour ajouter une arête entre deux nœuds
   PROCÉDURE ajouterArete(x, y)
      g[x-1][y-1] = 1
      g[y-1][x-1] = 1
   FIN PROCÉDURE
// Algorithme principal pour créer un graphe orienté ou non orienté
ALGORITHME GrapheOrM
   Graphe G
   G.ajouterArete()
FIN
```

Méthode Graphe() : Boucle Extérieure (i) : Exécute n fois, Boucle Intérieure (j) : Exécute n fois.

La complexité totale est O(n^2)

Méthode ajouterArete(x, y) : la complexité est O(1).

Complexité Totale

La complexité totale de ce programme est dominée par la création du graphe est O(n^2).

Liste d'adjacence

```
# Définition de la structure Liste
ALGORITHME Graphe
   STRUCTURE Liste
      info : ENTIER
      suiv : ADRESSE Liste
   FIN STRUCTURE
# Fonction pour créer une liste vide
   FONCTION creerListe()
      RETOURNER NULL
   FIN FONCTION
# Fonction pour ajouter un successeur à une liste
   FONCTION ajouterSuccesseur(liste, valeur)
      NOUVEAU : ADRESSE Liste
      nouveau.info = valeur
      nouveau.suiv = NULL
      SI liste == NULL ALORS
         RETOURNER nouveau
      SINON
         p = liste
         TANT QUE p.suiv != NULL FAIRE
            p = p.suiv
         FIN TANT QUE
         p.suiv = nouveau
         RETOURNER liste
      FIN SI
   FIN FONCTION
   FONCTION creerGraphe()
      ÉCRIRE ("donne le nombre de nœuds ")
      LIRE (n)
      g = tableau de n Listes
      POUR i DE 0 À n-1 FAIRE
```

```
            g[i] = creerListe()
 ÉCRIRE ("créer liste de successeur de nœud ", (i+1))
        LIRE valeur
        TANT QUE valeur != 0 FAIRE
          g[i] = ajouterSuccesseur(g[i], valeur)
          ÉCRIRE ("Donnez un successeur")
          LIRE (valeur)
        FIN TANT QUE
      FIN POUR
  FIN ALGORITHME
# Algorithme principal pour l'application
ALGORITHME Application
  Graphe G
  G.creerGraphe()
  G.afficherGraphe()
FIN ALGORITHME
```

Méthode Graphe() : Boucle Extérieure (i) : Exécute n fois, Appel à la Fonction creerListe() : Exécute en O(1), Boucle Intérieure (TANT QUE) : Peut être négligée dans la complexité totale.

La complexité totale est O(n) pour la création des listes d'adjacence.

Méthode ajouterSuccesseur(liste, valeur)

Création de Nouveau Noeud : Exécute en O(1).

Boucle Intérieure (TANT QUE) : Peut être négligée dans la complexité totale.

La complexité totale est O(1) pour l'ajout d'un successeur à une liste.

Méthode creerGraphe()

Boucle Extérieure (i) : Exécute n fois.

La complexité totale est O(n) pour la création de toutes les listes d'adjacence.

La complexité totale de ce programme est dominée par la création du graphe est O(n^2).

Affichage du graphe

Matrice d'adjacence

```
PROCÉDURE afficherGraphe()
    POUR i DE 0 À n-1 FAIRE
        ÉCRIRE ("les Successeurs du nœud: ", (i+1))
        POUR j DE 0 À n-1 FAIRE
            SI g[i][j] ≠ 0 ALORS
                ÉCRIRE (" ", (j+1))
            FIN SI
        FIN POUR
    FIN POUR
FIN PROCÉDURE
```

La complexité de cette procédure dépend du nombre total d'arêtes dans le graphe. Supposons que le nombre total d'arêtes soit EE. Dans le pire des cas, la double boucle parcourt chaque élément de la matrice, ce qui donne une complexité de O(n^2), où n est le nombre de nœuds dans le graphe.

Liste d'adjacence

```
PROCÉDURE afficherGraphe()
    POUR i DE 0 À n-1 FAIRE
        ÉCRIRE ("les Successeurs du nœud ", (i+1))
        p = g[i]
        TANT QUE p != NULL FAIRE
            ÉCRIRE (" ", p.info)
            p = p.suiv
        FIN TANT QUE
        ÉCRIRE ("")
    FIN POUR
FIN PROCÉDURE
```

Boucle Extérieure (i) : Exécute n fois.

Boucle Intérieure (TANT QUE) : Exécute dans le pire cas m fois, où m est le nombre total de successeurs dans toutes les listes.

La complexité totale est O(n + m) pour l'affichage des successeurs.

Calculer la densité du graphe

Matrice d'adjacence

```
Procédure DensiteGrapheM(matrice)
    nbrNoeuds = LONGUEUR(matrice)
    nbrAretes = 0

    POUR i DE 0 À nbrNoeuds - 1 FAIRE
        POUR j DE i + 1 À nbrNoeuds - 1 FAIRE
            SI matrice[i][j] == 1 ALORS
                nbrAretes = nbrAretes + 1
            FIN SI
        FIN POUR
    FIN POUR
```

Liste d'adjacence

```
Procédure DensiteGrapheL(liste)
    nbrNoeuds = LONGUEUR(liste)
    nbrAretes = 0

    POUR i DE 0 À nbrNoeuds - 1 FAIRE
        nbrAretes = nbrAretes + LONGUEUR(liste[i])
    FIN POUR

    RETOURNER 2 * nbrAretes / (nbrNoeuds *
(nbrNoeuds - 1))
FIN Procédure
```

La première boucle s'exécute n fois, où n est le nombre de nœuds

RETOURNER 2 * nbrAretes / (nbrNoeuds * (nbrNoeuds - 1))

FIN Procédure

La première boucle s'exécute n fois, où n est le nombre de nœuds.

La deuxième boucle s'exécute n−i fois en moyenne.

Les opérations à l'intérieur des boucles sont en O(1).

En simplifiant, la complexité est en O(n^2).

Les opérations à l'intérieur de la boucle sont en O(1).

En simplifiant, la complexité est en O(n).

Calculer le degré du graphe

Matrice d'adjacence

```
Procedure DegreGrapheM(matrice)
    nbrNoeuds = LONGUEUR(matrice)
    degres = NOUVEAU tableau de taille nbrNoeuds
    degres[] = 0 ;
    POUR i DE 0 À nbrNoeuds - 1 FAIRE
        POUR j DE 0 À nbrNoeuds - 1 FAIRE
            degres[i] = degres[i] + matrice[i][j]
        FIN POUR
    FIN POUR
    RETOURNER degres
FIN Procedure
```

La première boucle s'exécute n fois, où n est le nombre de nœuds.

La deuxième boucle s'exécute n fois.

Les opérations à l'intérieur des boucles sont en O(1).

Complexité Totale: O(n^2)

Liste d'adjacence

```
Procedure DegreGrapheL(liste)
    nbrNoeuds = LONGUEUR(liste)
    degres = NOUVEAU tableau de taille nbrNoeuds
initialisé à zéro

    POUR i DE 0 À nombreNoeuds - 1 FAIRE
        degres[i] = LONGUEUR(liste[i])
    FIN POUR

    RETOURNER degres
FIN Procedure
```

La boucle s'exécute n fois, où n est le nombre de nœuds.

Les opérations à l'intérieur de la boucle sont en O(1).

Complexité Totale: O(n))

Vérifier si le graphe est eulérien

Matrice d'adjacence

```
Procedure GrapheEulerienM(matrice)
   nbrNoeuds = LONGUEUR(matrice)
   POUR i DE 0 À nbrNoeuds - 1 FAIRE
      sommeDegres = 0
      POUR j DE 0 À nombreNoeuds - 1 FAIRE
         sommeDegres = sommeDegres + matrice[i][j]
      FIN POUR
# Vérification de la Parité
      SI sommeDegres % 2 != 0 ALORS
         RETOURNER FAUX
      FIN SI
   FIN POUR
   RETOURNER VRAI
FIN Procedure
```

La première boucle s'exécute n fois, où n est le nombre de nœuds.

La deuxième boucle s'exécute nn fois.

Les opérations à l'intérieur des boucles sont en O(1).

Complexité Totale: O(n^2)

Liste d'adjacence

```
Procedure GrapheEulerienL (list)
   nbrNoeuds = LONGUEUR(liste)
   POUR i DE 0 À nombreNoeuds - 1 FAIRE
      sommeDegres = LONGUEUR(liste[i])
      SI sommeDegres % 2 != 0 ALORS
         RETOURNER FAUX
      FIN SI
   FIN POUR
   RETOURNER VRAI
FIN Procedure
```

La boucle s'exécute nn fois, où nn est le nombre de nœuds.

Les opérations à l'intérieur de la boucle sont en O(1).

Complexité Totale: O(n)

Vérifier si le graphe est complet

Matrice d'adjacence

```
Procedure GrapheCompletM(matrice)
   nbrNoeuds = LONGUEUR(matrice)
   POUR i DE 0 À nbrNoeuds - 1 FAIRE
      POUR j DE 0 À nbrNoeuds - 1 FAIRE
         SI i != j ET matrice[i][j] != 1 ALORS
            RETOURNER FAUX
         FIN SI
      FIN POUR
   FIN POUR
```

Liste d'adjacence

```
Procedure GrapheCompletL(liste)
   nbrNoeuds = LONGUEUR(liste)
   POUR i DE 0 À nbrNoeuds - 1 FAIRE
      POUR CHAQUE voisin DANS liste[i] FAIRE
         SI voisin == i ALORS
            CONTINUER
         FIN SI
         SI voisin PAS DANS liste[i] ALORS
            RETOURNER FAUX
```

```
RETOURNER VRAI
FIN Procedure
```

Les deux boucles s'exécutent n^2 fois, où n est le nombre de nœuds.

Les opérations à l'intérieur des boucles sont en O(1).

Complexité Totale: O(n^2).

```
FIN SI
    FIN POUR
  FIN POUR
  RETOURNER VRAI
FIN Procedure
```

La première boucle s'exécute n fois, où n est le nombre de nœuds.

La deuxième boucle s'exécute au maximum n fois dans le pire cas (si tous les nœuds sont voisins entre eux).

Les opérations à l'intérieur des boucles sont en O(1).

Complexité Totale: O(n^2)

Vérifier si le graphe est un arbre

Matrice d'adjacence

```
ALGORITHME VerifierGrapheArbreM(matrice)
  nbrNoeuds = LONGUEUR(matrice)
  SI NON EstConnexeM(matrice) ALORS
    RETOURNER FAUX
  FIN SI

  SI ContientCycleM(matrice) ALORS
    RETOURNER FAUX
  FIN SI
  RETOURNER VRAI
FIN ALGORITHME

Procedure EstConnexeM(matrice)
  nombreNoeuds = LONGUEUR(matrice)
  sommetsVisites = NOUVEAU tableau de booléens
de taille nbrNoeuds initialisé à FAUX
  // Choisir un nœud de départ (ici, le premier)
ParcoursEnProfondeurM(0, matrice, sommetsVisites)
  // Vérifier si tous les nœuds ont été visités
  POUR i DE 0 À nbrNoeuds - 1 FAIRE
```

Liste d'adjacence

```
ALGORITHME VerifierGrapheArbreL(liste)
  nbrNoeuds = LONGUEUR(liste)
  SI NOT EstConnexeL(liste) ALORS
    RETOURNER FAUX
  FIN SI
  SI ContientCycleL(liste) ALORS
    RETOURNER FAUX
  FIN SI
  RETOURNER VRAI
FIN ALGORITHME

Procedure EstConnexeL(liste)
  nbrNoeuds = LONGUEUR(liste)
  sommetsVisites = NOUVEAU tableau de booléens
de taille nbrNoeuds initialisé à FAUX
  // Choisir un nœud de départ (ici, le premier)
  ParcoursEnProfondeurL(0, listeAdjacence,
sommetsVisites)
  // Vérifier si tous les nœuds ont été visités
  POUR i DE 0 À nbrNoeuds - 1 FAIRE
    SI NOT sommetsVisites[i] ALORS
```

```
        SI NON sommetsVisites[i] ALORS
RETOURNER FAUX
        FIN SI
    FIN POUR
    RETOURNER VRAI
FIN Procedure


Procedure ParcoursEnProfondeurMa(noeudCourant, matrice, sommetsVisites)
    sommetsVisites[noeudCourant] = VRAI
    // Parcours des voisins non visités
    POUR i DE 0 À LONGUEUR(matrice[noeudCourant]) - 1 FAIRE
        SI matrice[noeudCourant][i] == 1 ET NOT sommetsVisites[i] ALORS
  ParcoursEnProfondeurM(i, matrice, sommetsVisites)
        FIN SI
    FIN POUR
FIN Procedure
Procedure ContientCycleM(matrice)
    // La fonction prend en paramètre la matrice d'adjacence du graphe
    nbrNoeuds = LONGUEUR(matrice)  // Nombre de nœuds dans le graphe
    // Tableau pour suivre les nœuds visités pendant le parcours
    DEBUT
        tableauVisites = INITIALISER_VIDE()  // Initialisation du tableau des nœuds visités
    FIN
    // Boucle pour parcourir chaque nœud du graphe
    POUR i DE 0 À nbrNoeuds - 1 FAIRE
        DEBUT
            // Si le nœud n'a pas été visité, on lance une recherche en profondeur
            SI tableauVisites[i] == NON_VISITÉ ALORS
                DEBUT
```

```
RETOURNER FAUX
        FIN SI
    FIN POUR
    RETOURNER VRAI
FIN Procedure
Procedure ParcoursEnProfondeurL(noeudCourant, liste, sommetsVisites)
    sommetsVisites[noeudCourant] = VRAI
    // Parcours des voisins non visités
    POUR CHAQUE voisin DANS liste[noeudCourant] FAIRE
        SI NOT sommetsVisites[voisin] ALORS
            ParcoursEnProfondeurL(voisin, liste, sommetsVisites)
        FIN SI
    FIN POUR
FIN Procedure


Procedure ContientCycleL(liste)
    // La fonction prend en paramètre la liste d'adjacence du graphe

    nbrNoeuds = LONGUEUR(liste)  // Nombre de nœuds dans le graphe

    // Tableau pour suivre les nœuds visités pendant le parcours
    DEBUT
        tableauVisites = INITIALISER_VIDE()  // Initialisation du tableau des nœuds visités
    FIN

    // Boucle pour parcourir chaque nœud du graphe
    POUR i DE 0 À nbrNoeuds - 1 FAIRE
        DEBUT
            // Si le nœud n'a pas été visité, on lance une recherche en profondeur
            SI tableauVisites[i] == NON_VISITÉ ALORS
                DEBUT
```

```
        // Appel à la fonction de recherche en profondeur (DFS)
                SI ContientCycleDFS(i, tableauVisites, matrice) ALORS
                    RETOURNER VRAI  // Le graphe contient un cycle
                FIN SI
            FIN
        FIN SI
    FIN
  FIN POUR
  // Aucun cycle n'a été détecté après le parcours de tous les nœuds
  RETOURNER FAUX
FIN Procedure


Procedure ContientCycleDFS(noeudCourant, tableauVisites, matrice)
  // Fonction DFS récursive pour explorer le graphe en profondeur et détecter la présence d'un cycle
  // Marquer le nœud courant comme visité
  tableauVisites[noeudCourant] = VISITÉ
  // Parcourir tous les nœuds adjacents au nœud courant
  POUR i DE 0 À LONGUEUR(matrice[noeudCourant]) - 1 FAIRE
      DEBUT
        SI matrice[noeudCourant][i] == 1 ALORS
            // Si le nœud adjacent n'a pas été visité, lancer une nouvelle recherche en profondeur
            SI tableauVisites[i] == NON_VISITÉ ALORS
                DEBUT
                    // Appel récursif à la fonction DFS
                    SI ContientCycleDFS(i, tableauVisites, matrice) ALORS
                        RETOURNER VRAI  // Un cycle a été détecté
                    FIN SI
                FIN
            FIN SI
```

```
// Appel à la fonction de recherche en profondeur (DFS)
                SI ContientCycleDFSL(i, tableauVisites, liste) ALORS
                    RETOURNER VRAI  // Le graphe contient un cycle
                FIN SI
            FIN
        FIN SI
    FIN
  FIN POUR
  // Aucun cycle n'a été détecté après le parcours de tous les nœuds
  RETOURNER FAUX
FIN Procedure


Procedure ContientCycleDFSL(noeudCourant, tableauVisites, liste)
  // Fonction DFS récursive pour explorer le graphe en profondeur et détecter la présence d'un cycle
  // Marquer le nœud courant comme visité
  tableauVisites[noeudCourant] = VISITÉ
  // Parcourir tous les nœuds adjacents au nœud courant
  POUR CHAQUE voisin DANS liste[noeudCourant] FAIRE
      DEBUT
          // Si le nœud adjacent n'a pas été visité, lancer une nouvelle recherche en profondeur
          SI tableauVisites[voisin] == NON_VISITÉ ALORS
            DEBUT
                // Appel récursif à la fonction DFS
                SI ContientCycleDFSL(voisin, tableauVisites, liste) ALORS
                    RETOURNER VRAI  // Un cycle a été détecté
                FIN SI
            FIN
  FIN POUR
```

```
        FIN SI
    FIN
FIN POUR
// Si la recherche en profondeur ne détecte pas de cycle, retourner FAUX
RETOURNER FAUX
FIN Procedure
```

EstConnexeM :

Complexité : O(n + e) où n est le nombre de nœuds et e est le nombre d'arêtes.

ContientCycleDFS :

Complexité : O(n + e) où n est le nombre de nœuds et e est le nombre d'arêtes.

ContientCycleM :

Complexité : O(n * (n + e)) où n est le nombre de nœuds et e est le nombre d'arêtes.

VerifierGrapheArbreM :

Complexité : O(n * (n + e)) où n est le nombre de nœuds et e est le nombre d'arêtes.

Cela peut être simplifié en O(n^2) dans le pire des cas.

```
// Si la recherche en profondeur ne détecte pas de cycle, retourner FAUX
RETOURNER FAUX
FIN ALGORITHME
```

EstConnexeL :

Complexité : O(n + e) où n est le nombre de nœuds et e est le nombre d'arêtes.

ContientCycleDFSL :

Complexité : O(n + e) où n est le nombre de nœuds et e est le nombre d'arêtes.

ContientCycleL :

Complexité : O(n * (n + e)) où n est le nombre de nœuds et e est le nombre d'arêtes.

VerifierGrapheArbreL :

Complexité : O(n * (n + e)) où n est le nombre de nœuds et e est le nombre d'arêtes.

Cela peut être simplifié en O(n^2) dans le pire des cas.

Recherche d’un noeud a dans le graphe

Matrice d’adjacence

```
ALGORITHME RechercheNoeudM(a, matrice)
    nbrNoeuds = LONGUEUR(matrice)
    indiceNoeud = -1
    // Trouver l'indice du nœud recherché
    POUR i DE 0 À nombreNoeuds - 1 FAIRE
        SI matrice[i][0] == noeudRecherche ALORS
            indiceNoeud = i
            ARRÊTER
        FIN SI
    FIN POUR
// Afficher le nœud et ses liens
    SI indiceNoeud != -1 ALORS
```

Liste d’adjacence

```
ALGORITHME RechercheNoeudL(noeudRecherche, liste)
    nbrNoeuds = LONGUEUR(liste)
    indiceNoeud = -1
    // Trouver l'indice du nœud recherché
    POUR i DE 0 À nbrNoeuds - 1 FAIRE
        SI liste [i][0] == noeudRecherche ALORS
            indiceNoeud = i
            ARRÊTER
        FIN SI
    FIN POUR
    // Afficher le nœud et ses liens
```

```
 ECRIRE "Noeud trouvé : ", matrice[indiceNoeud][0]
    ECRIRE "Liens : "
    POUR j DE 1 À LONGUEUR(matrice[indiceNoeud]) - 1 FAIRE
      ECRIRE matrice[indiceNoeud][j], " "
    FIN POUR
  SINON
    ECRIRE "Noeud non trouvé"
  FIN SI
FIN
```

La complexité totale de l'algorithme est O(n), où n est le nombre de nœuds.

```
SI indiceNoeud != -1 ALORS
    ECRIRE "Noeud trouvé : ", liste[indiceNoeud][0]
    ECRIRE "Liens : "
    POUR j DE 1 À LONGUEUR(liste[indiceNoeud]) - 1 FAIRE
      ECRIRE liste[indiceNoeud][j], " "
    FIN POUR
  SINON
    ECRIRE "Noeud non trouvé"
  FIN SI
FIN
```

La complexité totale de l'algorithme est O(n), où n est le nombre de nœuds.

Recherche de tous les chemins entre un noeud a et un noeud b

Matrice d'adjacence

```
ALGORITHME TousCheminsM(noeudA, noeudB, matrice)
  nombreNoeuds = LONGUEUR(matrice)
  cheminActuel = NOUVEAU tableau d'entiers
  cheminsTrouves = NOUVEAU tableau de tableaux d'entiers
 TrouverCheminsM(noeudA, noeudB, matrice, cheminActuel, cheminsTrouves)
  ECRIRE "Tous les chemins entre ", noeudA, " et ", noeudB, " :"
  POUR i DE 0 À LONGUEUR(cheminsTrouves) - 1 FAIRE
    ECRIRE "Chemin ", i + 1, ": ", cheminsTrouves[i]
  FIN POUR
FIN ALGORITHME

Procedure TrouverCheminsM(noeudCourant, noeudB, matrice, cheminActuel, cheminsTrouves)
  AJOUTER noeudCourant À cheminActuel
```

Liste d'adjacence

```
ALGORITHME TousCheminsL(noeudA, noeudB, liste)
  nombreNoeuds = LONGUEUR(liste)
  cheminActuel = NOUVEAU tableau d'entiers
  cheminsTrouves = NOUVEAU tableau de tableaux d'entiers
 TrouverCheminsL(noeudA, noeudB, listE, cheminActuel, cheminsTrouves)
  ECRIRE "Tous les chemins entre ", noeudA, " et ", noeudB, " :"
  POUR i DE 0 À LONGUEUR(cheminsTrouves) - 1 FAIRE
    ECRIRE "Chemin ", i + 1, ": ", cheminsTrouves[i]
  FIN POUR
FIN ALGORITHME

Procedure TrouverCheminsL(noeudCourant, noeudB, listeAdjacence, cheminActuel, cheminsTrouves)
  AJOUTER noeudCourant À cheminActuel
```

```
SI noeudCourant == noeudB ALORS

        // Noeud de destination atteint, ajouter le cheminActuel à cheminsTrouves

        AJOUTER COPIE(cheminActuel) À cheminsTrouves

    SINON

        // Explorer les voisins non visités

        POUR i DE 0 À LONGUEUR(matrice[noeudCourant]) - 1 FAIRE

            SI matrice[noeudCourant][i] == 1 ET i NON DANS cheminActuel ALORS

 TrouverCheminsM(i, noeudB, matrice, cheminActuel, cheminsTrouves)

            FIN SI

        FIN POUR

    FIN SI

    // Retirer le dernier nœud pour revenir en arrière lors de la récursion

    SUPPRIMER DERNIER(cheminActuel)

FIN Procedure
```

La complexité de cet algorithme dépend du nombre de chemins entre noeudA et noeudB, ce qui peut être exponentiel dans le pire des cas. En notation big-O, la complexité peut être exprimée comme O(2^E), où E est le nombre total d'arêtes dans le graphe.

```
SI noeudCourant == noeudB ALORS

        // Noeud de destination atteint, ajouter le cheminActuel à cheminsTrouves

        AJOUTER COPIE(cheminActuel) À cheminsTrouves

    SINON

        // Explorer les voisins non visités

        POUR CHAQUE voisin DANS liste[noeudCourant] FAIRE

            SI voisin NON DANS cheminActuel ALORS

TrouverCheminsL(voisin, noeudB, liste, cheminActuel, cheminsTrouves)

            FIN SI

        FIN POUR

    FIN SI

    // Retirer le dernier nœud pour revenir en arrière lors de la récursion

    SUPPRIMER DERNIER

FIN Procedure
```

La complexité de cet algorithme dépend du nombre de chemins entre noeudA et noeuDB dans la liste d'adjacence. Dans le pire des cas, si chaque nœud a nombreNoeuds−1 voisins et chaque voisin est inclus ou exclu dans le chemin, la complexité peut être exponentielle, O(2^nombreNoeuds)

Recherche du chemin le plus court entre deux noeuds a et b

Matrice d'adjacence

```
PROCEDURE CheminPlusCourtM(noeudA, noeudB, matrice)

    nbrNoeuds = LONGUEUR(matrice)

    fileNoeuds = NOUVELLE file de nœuds

    fileChemins = NOUVELLE file de chemins

    visites = NOUVEAU tableau de booléens de taille nbrNoeuds

    INITIALISER visites À FAUX
```

Liste d'adjacence

```
    PROCEDURE CheminPlusCourtL(noeudA, noeudB, listeAdjacence)

    nbrNoeuds = LONGUEUR(liste)

    fileNoeuds = NOUVELLE file de nœuds

    fileChemins = NOUVELLE file de chemins

    visites = NOUVEAU tableau de booléens de taille nbrNoeuds

    INITIALISER visites À FAUX
```

```
ENQUEUE (noeudA, [noeudA]) DANS fileNoeuds
// Ajouter le nœud initial avec son propre chemin
   TANT QUE fileNoeuds NON VIDE FAIRE
      (noeudCourant, cheminCourant) = DEQUEUE(fileNoeuds)
      SI noeudCourant == noeudB ALORS
         // Afficher le chemin le plus court trouvé
         ECRIRE "Chemin le plus court entre ", noeudA, " et ", noeudB, " : ", cheminCourant
         ARRÊTER
      FIN SI
      SI visites[noeudCourant] == FAUX ALORS
         POUR i DE 0 À nbrNoeuds - 1 FAIRE
            SI matrice[noeudCourant][i] == 1 ET visites[i] == FAUX ALORS
               ENQUEUE (i, CONCATÉNER(cheminCourant, [i])) DANS fileNoeuds
               visites[i] = VRAI
            FIN SI
         FIN POUR
      FIN SI
   FIN TANT QUE
   ECRIRE "Aucun chemin trouvé entre ", noeudA, " et ", noeudB
FIN PROCEDURE
```

Boucle principale: O(n)

Boucle interne: O(n)

Opérations dans la boucle: O(1)

Complexité totale : O(n^2)

```
ENQUEUE (noeudA, [noeudA]) DANS fileNoeuds
// Ajouter le nœud initial avec son propre chemin
   TANT QUE fileNoeuds NON VIDE FAIRE
      (noeudCourant, cheminCourant) = DEQUEUE(fileNoeuds)
      SI noeudCourant == noeudB ALORS
         // Afficher le chemin le plus court trouvé
         ECRIRE "Chemin le plus court entre ", noeudA, " et ", noeudB, " : ", cheminCourant
         ARRÊTER
      FIN SI
      SI visites[noeudCourant] == FAUX ALORS
         POUR CHAQUE voisin DANS liste[noeudCourant] FAIRE
            ENQUEUE (voisin, CONCATÉNER(cheminCourant, [voisin])) DANS fileNoeuds
            visites[voisin] = VRAI
         FIN POUR
      FIN SI
   FIN TANT QUE
   ECRIRE "Aucun chemin trouvé entre ", noeudA, " et ", noeudB
FIN PROCEDURE
```

Boucle principale: O(n)

Boucle interne: O(n)

Opérations dans la boucle: O(1)

Complexité totale : O(n^2)

Recherche d'une composante (fortement) connexe à partir d'un noeud a.

Matrice d'adjacence

```
PROCEDURE ComposanteConnexeM(noeudA, matrice)
    nbrNoeuds = LONGUEUR(matrice)
    pileNoeuds = NOUVELLE pile de nœuds
    composanteConnexe = NOUVEAU tableau de booléens de taille nbrNoeuds
    visites = NOUVEAU tableau de booléens de taille nbrNoeuds

    INITIALISER composanteConnexe À FAUX
    INITIALISER visites À FAUX

    EMPILER noeudA DANS pileNoeuds
    TANT QUE pileNoeuds NON VIDE FAIRE
        noeudCourant = DEPILER(pileNoeuds)
        composanteConnexe[noeudCourant] = VRAI
        visites[noeudCourant] = VRAI
        POUR i DE 0 À nbrNoeuds - 1 FAIRE
            SI matrice[noeudCourant][i] == 1 ET visites[i] == FAUX ALORS
                EMPILER i DANS pileNoeuds
            FIN SI
        FIN POUR
    FIN TANT QUE
    // Afficher les nœuds de la composante connexe
    AFFICHER "Composante connexe à partir de ", noeudA, " :"
    POUR i DE 0 À nbrNoeuds - 1 FAIRE
        SI composanteConnexe[i] == VRAI ALORS
            AFFICHER i
        FIN SI
    FIN POUR
FIN PROCEDURE
```

Liste d'adjacence

```
PROCEDURE ComposanteConnexeL(noeudA, listeAdjacence)
    nbrNoeuds = LONGUEUR(liste)
    pileNoeuds = NOUVELLE pile de nœuds
    composanteConnexe = NOUVEAU tableau de booléens de taille nbrNoeuds
    visites = NOUVEAU tableau de booléens de taille nbrNoeuds

    INITIALISER composanteConnexe À FAUX
    INITIALISER visites À FAUX

    EMPILER noeudA DANS pileNoeuds
    TANT QUE pileNoeuds NON VIDE FAIRE
        noeudCourant = DEPILER(pileNoeuds)
        composanteConnexe[noeudCourant] = VRAI
        visites[noeudCourant] = VRAI
POUR CHAQUE voisin DANS liste[noeudCourant] FAIRE
            SI visites[voisin] == FAUX ALORS
                EMPILER voisin DANS pileNoeuds
            FIN SI
        FIN POUR
    FIN TANT QUE
    // Afficher les nœuds de la composante connexe
    AFFICHER "Composante connexe à partir de ", noeudA, " :"
    POUR i DE 0 À nbrNoeuds - 1 FAIRE
        SI composanteConnexe[i] == VRAI ALORS
            AFFICHER i
        FIN SI
    FIN POUR
FIN PROCEDURE
```

Initialisation : O(n)

Boucle principale : O(n^2)

Opérations dans la boucle : O(1)

Complexité totale : O(n^2)

Initialisation : O(n)

Boucle principale : O(n·d), où d est le degré maximal du graphe

Opérations dans la boucle : O(1)

Complexité totale : O(n*d)

Trouver tous les cycles/circuits dans le graphe

Nous avons utilisé cette procédure pour vérifier si le graphe est un arbre, donc elle est déjà réalisée.
complexité :

Initialisation : O(n)

Boucle principale : O(n^2)

Opérations dans la boucle : O(1)

Complexité totale : O(n^2)

Initialisation : O(n)

Boucle principale : O(n·d), où d est le degré maximal du graphe

Opérations dans la boucle : O(1)

Complexité totale : O(n*d)

Ajouter un noeud a avec ses liens

Matrice d'adjacence

```
PROCEDURE AjouterNoeudM(noeud, matrice)
  nbrNoeuds = LONGUEUR(matrice)
  // Ajouter une nouvelle ligne et colonne à la matrice
  POUR i DE 0 À nbrNoeuds FAIRE
    AJOUTER 0 À matrice[i]
  FIN POUR
  AJOUTER NOUVELLE LIGNE DE
(nombreNoeuds + 1) ZÉROS À matrice
  // Mettre à jour les liens existants
  POUR i DE 0 À nombreNoeuds FAIRE
    SI matrice[i][noeud] == 1 ALORS
      matrice[i][noeud] = 0  // Supprimer le lien
existant
      matrice[noeud][i] = 1  // Ajouter un nouveau
lien
    FIN SI
  FIN POUR
FIN PROCEDURE
```

Liste d'adjacence

```
PROCEDURE AjouterNoeudL(noeud, listeAdjacence)
  nbrNoeuds = LONGUEUR(liste)
  // Ajouter une nouvelle liste vide pour le nouveau
nœud
  listeAdjacence[AJOUTER LISTE VIDE]
  // Mettre à jour les liens existants
  POUR i DE 0 À nbrNoeuds - 1 FAIRE
    SI i ≠ noeud ALORS
      AJOUTER 0 À listeAdjacence[i]  // Ajouter un
lien vide pour le nouveau nœud
    FIN SI
  FIN POUR
FIN PROCEDURE
```

La création d'une nouvelle liste pour le nouveau nœud prend O(1).

La suppression des liens existants dans les listes voisines prend O(d), où d est le degré moyen du nœud.

La complexité totale est O(n*d).

La création d'une nouvelle ligne et colonne prend O(n) (nombre de nœuds).

La mise à jour des liens existants prend O(n) dans le pire cas.

La complexité totale est O(n).

Supprimer un noeud a avec ses liens :

Matrice d'adjacence

```
PROCEDURE SupprimerNoeudM(noeud, matrice)
    nbrNoeuds = LONGUEUR(matrice)
    // Supprimer la ligne du nœud
    POUR i DE 0 À nbrNoeuds - 1 FAIRE
        SUPPRIMER matrice[i][noeud]
    FIN POUR
    // Supprimer la colonne du nœud
    POUR i DE 0 À nbrNoeuds - 1 FAIRE
        SUPPRIMER matrice[noeud][i]
    FIN POUR
    // Supprimer la ligne et la colonne du nœud
    SUPPRIMER matrice[noeud]
FIN PROCEDURE
```

La boucle principale parcourt tous les nœuds existants pour mettre à jour les liens existants. Elle s'exécute en O(n), où n est le nombre total de nœuds dans le graphe.

Les opérations à l'intérieur des boucles sont en O(1),

Complexité Totale : O(n)

Liste d'adjacence

```
PROCEDURE SupprimerNoeudL (noeud, liste)
    SUPPRIMER listeAdjacence[noeud]
    // Parcourir les listes voisines
    POUR CHAQUE liste VOISINE DANS listeAdjacence FAIRE
        // Supprimer le nœud des listes voisines
        SI noeud DANS liste ALORS
            liste SUPPRIMER noeud
        FIN SI
    FIN POUR
FIN PROCEDURE
```

La boucle principale parcourt tous les nœuds existants pour mettre à jour les liens existants. Elle s'exécute en O(n), où n est le nombre total de nœuds dans le graphe.

Boucle Interne : À l'intérieur de la boucle principale, une autre boucle parcourt les voisins du nœud actuel. Cette boucle interne s'exécute en O(d), où d est le degré maximal d'un nœud (nombre de voisins).

Les opérations à l'intérieur des boucles sont en O(1),

Complexité Totale : O(n * d)

Ajouter un lien (arc ou arête) entre deux noeuds existants

Matrice d'adjacence

```
PROCEDURE AjouterLienM(noeudA, noeudB, matrice)
    matrice[noeudA][noeudB] = 1
    matrice[noeudB][noeudA] = 1
FIN PROCEDURE
```

L'ajout d'un lien entre deux nœuds dans la matrice prend O(1).

Liste d'adjacence

```
PROCEDURE AjouterLienL(noeudA, noeudB, liste)
    SI noeudB NON DANS liste[noeudA] ALORS
        AJOUTER noeudB À liste[noeudA]
    FIN SI
    SI noeudA NON DANS liste[noeudB] ALORS
        AJOUTER noeudA À liste[noeudB]
    FIN SI
FIN PROCEDURE
```

La vérification de l'existence du lien dans la liste d'adjacence prend O(d), où d est le degré moyen du nœud.

L'ajout d'un lien entre deux nœuds dans la liste d'adjacence prend O(1).

La complexité totale est O(d).

5.2.Tableau comparatif des différentes complexités

Algorithme	matrice d'adjacence	liste d'adjacence
Construction d'un graphe orienté/non orienté	$O(n^2)$	$O(n)$
Affichage du graphe	$O(n^2)$	$O(n + m)$
Calculer la densité du graphe	$O(n^2)$	$O(m)$
Calculer le degré du graphe	$O(n)$	$O(n)$
Vérifier si le graphe est eulérien	$O(n^2)$	$O(n)$
Vérifier si le graphe est complet	$O(n^2)$	$O(n^2)$
Vérifier si le graphe est un arbre	$O(n^2)$	$O(n^2)$
Recherche d'un noeud a dans le graphe	$O(n)$	$O(n)$
Recherche de tous les chemins entre un noeud a et un noeud b	O(2^m)	O(2^n)
Recherche du chemin le plus court entre deux noeuds a et b	$O(n^2)$	$O(n^2)$
Recherche d'une composante connexe à partir d'un noeud a.	$O(n^2)$	$O(n * m)$
Trouver tous les cycles/circuits dans le graphe	$O(n^2)$	$O(n * d)$
Ajouter un noeud a avec ses liens	$O(n^2)$	$O(n*d)$
Supprimer un noeud a avec ses liens	$O(n^2)$	$O(n * d)$
Ajouter un lien (arc ou arête) entre deux noeuds existants	$O(1)$	$O(d)$

Tels que **m** représente le nombre d'arêtes et **d** est le degré moyen du nœud.

6. Evaluation expérimentale

L'évaluation expérimentale de la complexité des algorithmes en temps d'exécution permet d'analyser comment les performances des algorithmes varient en fonction de la taille des données d'entrée. En mesurant le temps nécessaire pour différentes opérations sur des graphes de tailles croissantes, on peut obtenir des informations précieuses sur l'efficacité des algorithmes dans des contextes pratiques.

Nous avons utilisé des le module time pour mesurer le temps d'exécution de nos programmes.

```
import time

start_time = time.time()

# Votre code ici

end_time = time.time()
execution_time = end_time - start_time
print(f"Temps d'exécution : {execution_time} secondes")
```

Figure 1: Mesure du Temps d'Exécution d'une Fonction en Python

Ces fonctions créent des graphes avec une matrice d'adjacence pour effectuer des tests avec différentes tailles allant de 10 à 50, sans nécessiter une saisie manuelle des données.

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

def constGraph10():
    G = nx.Graph(np.zeros((10, 10), dtype=int))
    return G

def constGraph20():
    G = nx.Graph(np.zeros((20, 20), dtype=int))
    return G

def constGraph30():
    G = nx.Graph(np.zeros((30, 30), dtype=int))
    return G

def constGraph40():
    G = nx.Graph(np.zeros((40, 40), dtype=int))
    return G

def constGraph50():
    G = nx.Graph(np.zeros((50, 50), dtype=int))
    return G

# Exemple d'utilisation
graph_10 = constGraph10()
graph_20 = constGraph20()
graph_30 = constGraph30()
graph_40 = constGraph40()
graph_50 = constGraph50()
```

Figure 2: Génération Automatique de Graphes avec Matrices d'Adjacence

De même, ces fonctions sont utilisées pour la représentation avec des listes d'adjacence.

```python
import networkx as nx
import matplotlib.pyplot as plt

def constGraphList10():
    G = nx.Graph()
    for i in range(1, 11):
        G.add_node(i)
    return G

def constGraphList20():
    G = nx.Graph()
    for i in range(1, 21):
        G.add_node(i)
    return G

def constGraphList30():
    G = nx.Graph()
    for i in range(1, 31):
        G.add_node(i)
    return G

def constGraphList40():
    G = nx.Graph()
    for i in range(1, 41):
        G.add_node(i)
    return G

def constGraphList50():
    G = nx.Graph()
    for i in range(1, 51):
        G.add_node(i)
    return G

# Exemple d'utilisation
graph_list_10 = constGraphList10()
graph_list_20 = constGraphList20()
graph_list_30 = constGraphList30()
graph_list_40 = constGraphList40()
graph_list_50 = constGraphList50()
```

Figure 3: Génération Automatique de Graphes avec Listes d'Adjacence

Tableau d'évaluation expérimentale

Taille	10	20	30	40	50
Construction	4.3153762817 38281e-05	0.0001249313 3544921875	0.0001492500 3051757812	0.0007071495 056152344	0.0032372474 670410156
Affichage	0.7945282459 259033	0.8669712543 487549	0.9296352863 311768	0.9594712257 385254	1.8152244091 033936
Densité	8.5830688476 5625e-06	6.1988830566 40625e-06	6.9141387939 453125e-06	1.0013580322 265625e-05	1.1682510375 976562e-05
Degré	5.4836273193 359375e-06	5.7683715820 3125e-06	6.1988830566 40625e-06	7.6293945312 5e-06	9.2983245849 60938e-06
graphe eulérien	0.0005829334 259033203	0.0006146430 969238281	0.0006608963 012695312	0.0008170604 705810547	0.0008358955 383300781
graphe complet	4.2315964785 4559886e-06	3.1231545646 878977e-04	0.0002364236 59725586	0.0041236578 93355899	0.0126549325 566544225
graphe arbre	5.3214669756 556998-06	0.0003214698 57566985668	0.0012364896 55698745	0.0096532156 98566688	0.0536521448 9623655
Recherche noeud	0.0010306835 174560547	0.0021233558 654785156	0.0051333904 26635742	0.0083899497 98583984	0.0118792057 0373535
Recherche chemins	0.0010306835 174560547	0.0021233558 654785156	0.0051333904 26635742	0.0083899497 98583984	0.0118792057 03735352
Recherche chemin court	0.0072548389 43481445	0.2580022811 8896484	0.8782501220 703125	1.3054955005 645752	2.5751290321 350098
composante connexe	0.0902409553 527832	0.1191301345 8251953	0.2611770629 8828125	0.1950285434 7229004	0.2086873054 5043945
cycles/circuits	0.0001494884 490966797	0.0006628036 499023438	0.0006628036 499023438	0.0018334388 73291015	0.0040404796 6003418
Ajouter/ Supprimer noeud	0.0002915859 2224121094	0.0002210140 2282714844	0.0001795291 9006347656	0.0001614093 780517578	0.0002050399 7802734375
Ajouter un lien	0.0017588138 580322266	0.0015206336 975097656	0.0013477802 276611328	0.0014715194 702148438	0.0638349056 2438965

PS : Le temps d'exécution exprimé en secondes

7. Conclusion

Ce projet offre une plongée approfondie dans le monde des graphes informatiques, en mettant en lumière les subtilités de leur représentation en mémoire et les nuances des opérations fondamentales. À travers l'analyse de complexités théoriques et expérimentales, il vise à éclairer sur les compromis et les avantages de chaque approche.

Ce travail ne se limite pas à l'implémentation brute, mais cherche à fournir une compréhension holistique des structures de graphes, équipant ainsi le lecteur avec des connaissances cruciales pour aborder des problèmes complexes dans divers domaines de l'informatique.

La sélection entre la matrice d'adjacence et la liste d'adjacence pour représenter un graphe dépend de divers compromis entre la complexité spatiale et temporelle.

La matrice d'adjacence offre une représentation simple et directe du graphe, où chaque entrée indique la présence ou l'absence d'arêtes entre les nœuds. Cependant, elle consomme beaucoup de mémoire, ce qui peut devenir un inconvénient pour les graphes de grande taille ou peu denses. Les opérations, telles que la construction du graphe ou le calcul de densité, peuvent être coûteuses en termes de temps, notamment pour des graphes denses.

D'un autre côté, la liste d'adjacence utilise moins de mémoire, surtout pour des graphes peu denses, car elle ne stocke que les arêtes réellement présentes. Cette représentation permet des opérations plus rapides pour les graphes clairsemés, mais elle peut être moins efficace pour certaines opérations sur des graphes denses.

En termes de construction et d'affichage du graphe, la liste d'adjacence tend à être plus performante, tandis que la matrice d'adjacence peut être préférable pour des opérations comme le calcul de densité ou de degré dans certains cas.

La taille et la densité du graphe jouent un rôle crucial dans le choix de la représentation. Pour des graphes de petite taille et peu denses, la liste d'adjacence se distingue souvent par sa meilleure efficacité. Cependant, pour des graphes plus denses ou de grande taille, la matrice d'adjacence peut présenter des avantages, bien que cela puisse entraîner une utilisation plus intensive de la mémoire.

Annexe

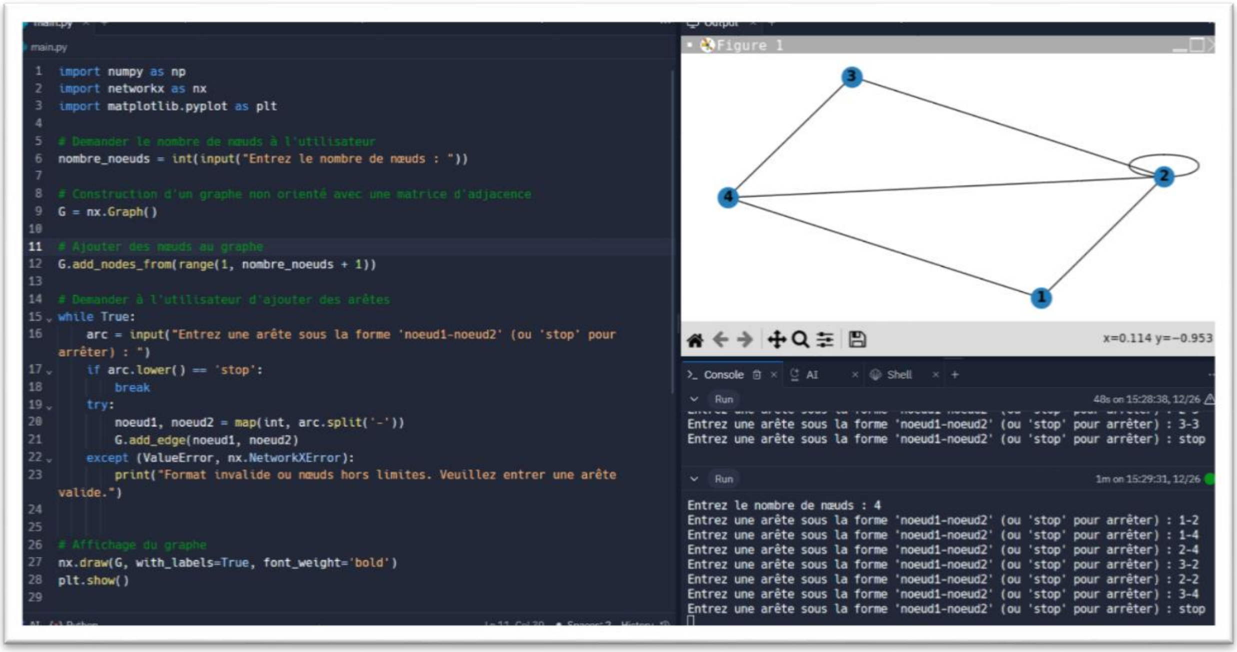

Figure 4: Construction et affichage un graphe non orienté avec matrice d'adjacence

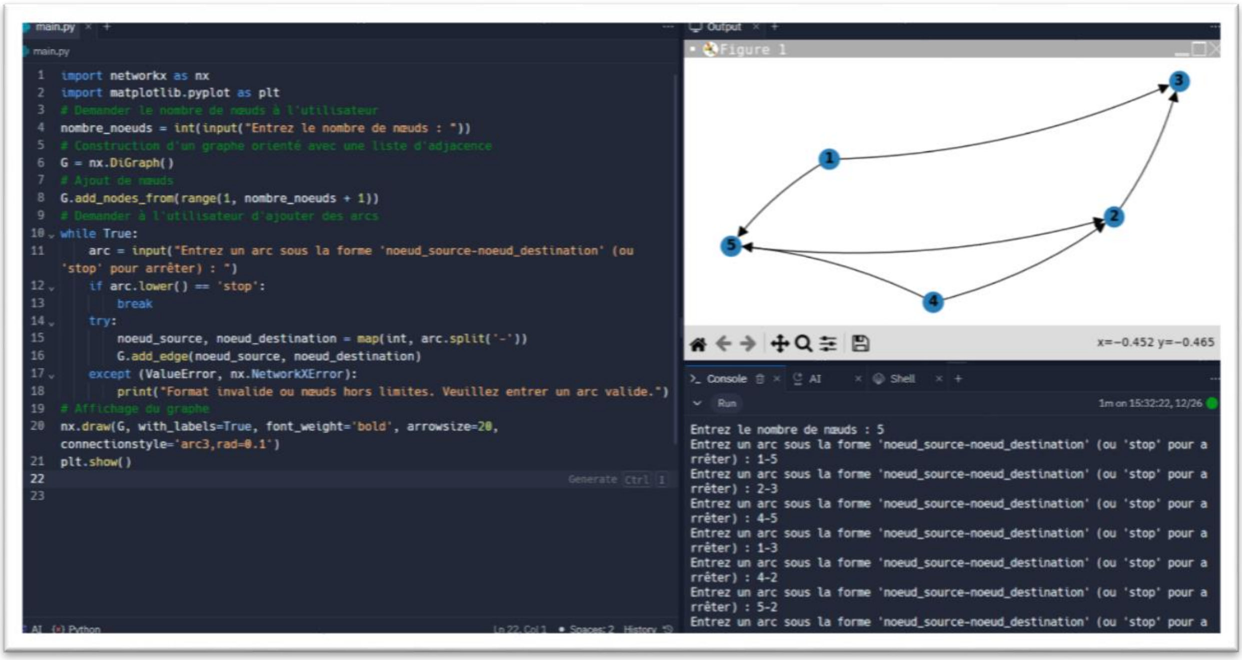

Figure 5: Construction et affichage un graphe orienté avec liste d'adjacence

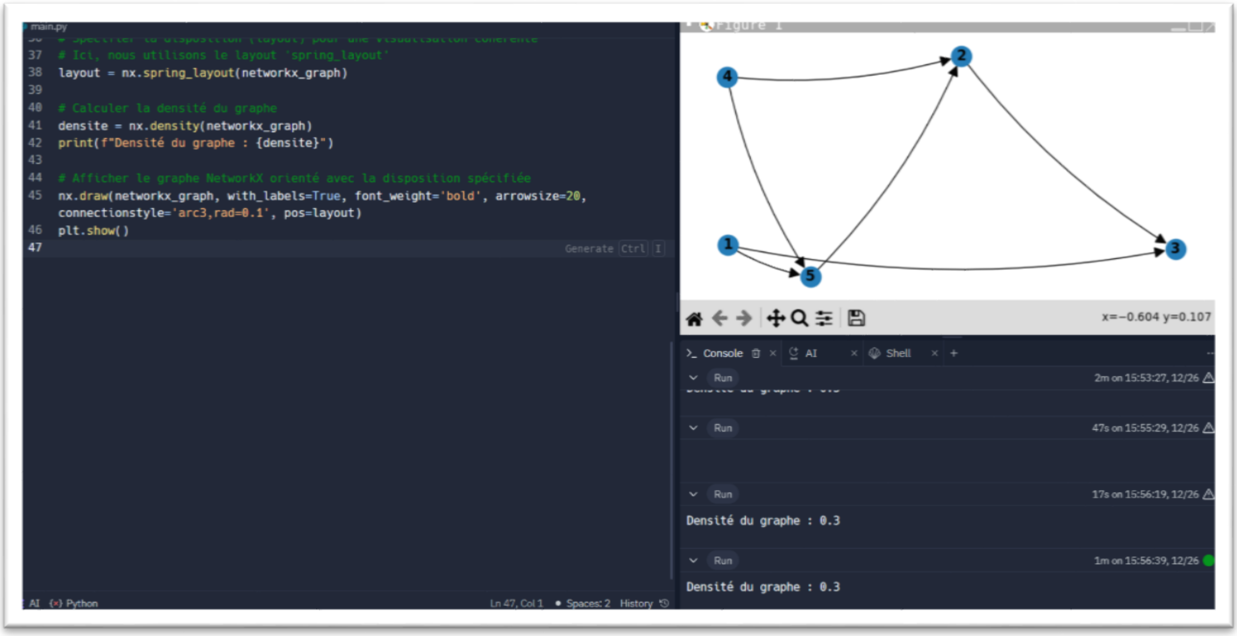

Figure 6: Calcul densité de graphe

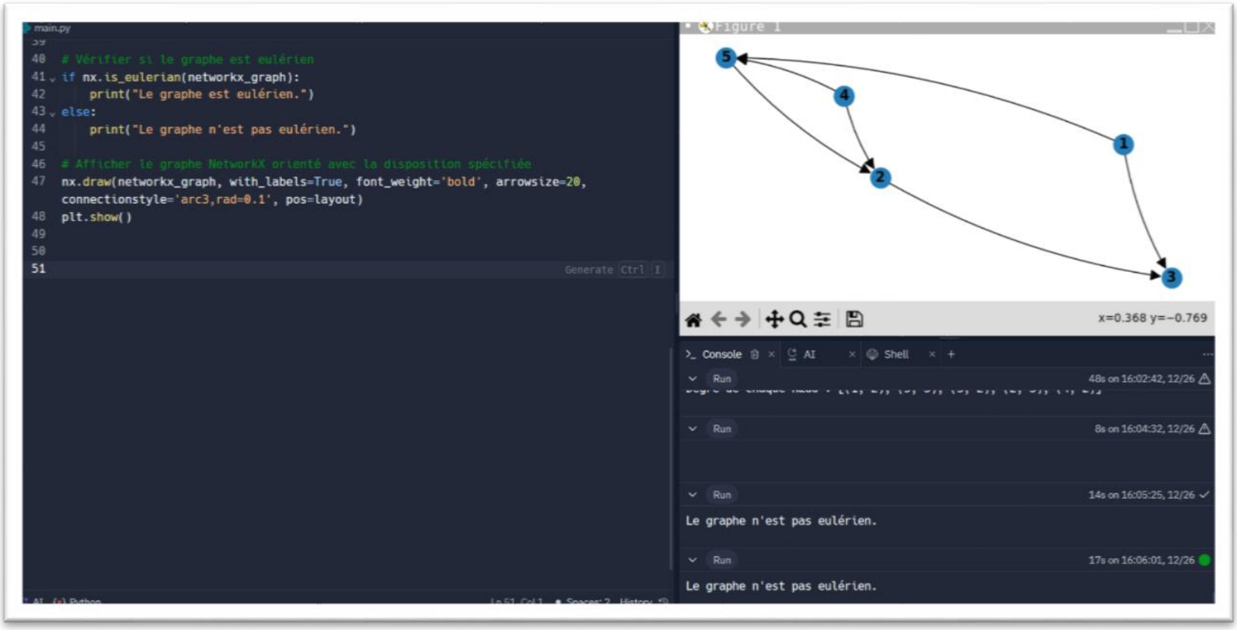

Figure 7: Vérification de la Propriété Eulérienne d'un Graphe

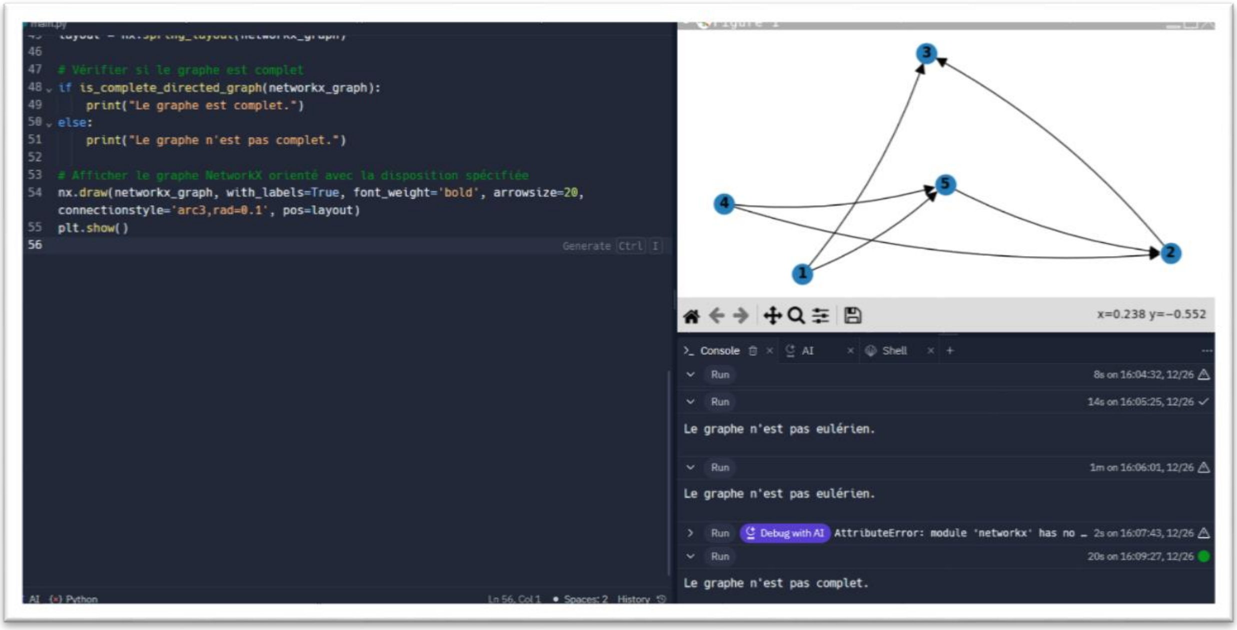

Figure 8: Vérification si le graphe est complet

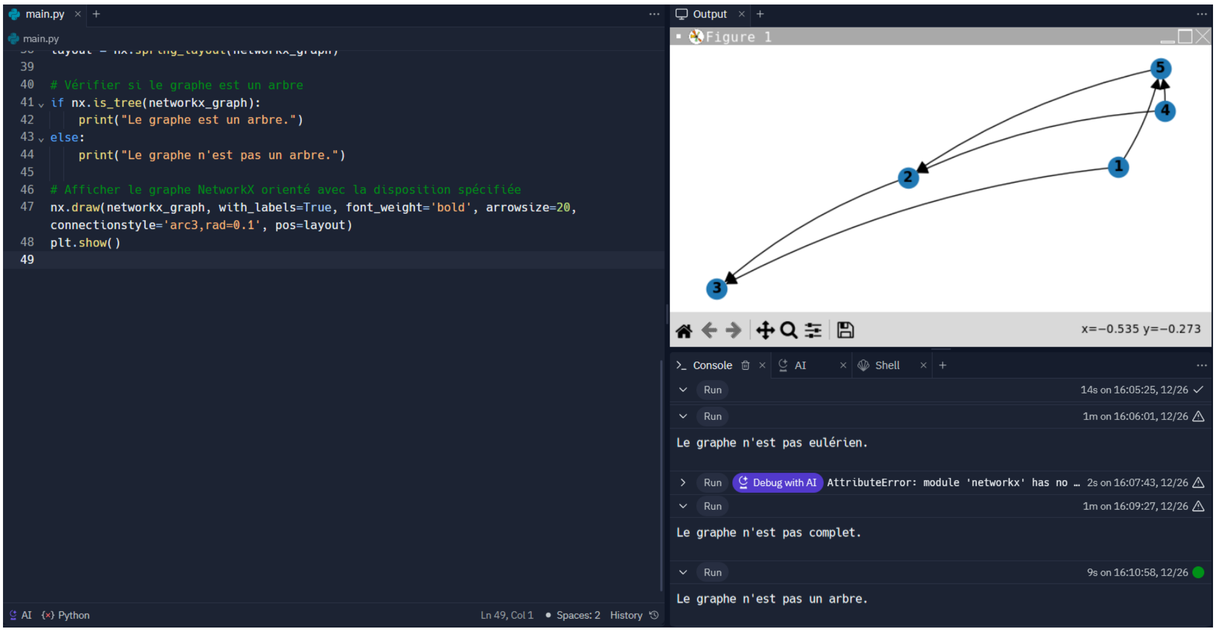

Figure 9: Vérification de la Nature Arborescente d'un Graphe

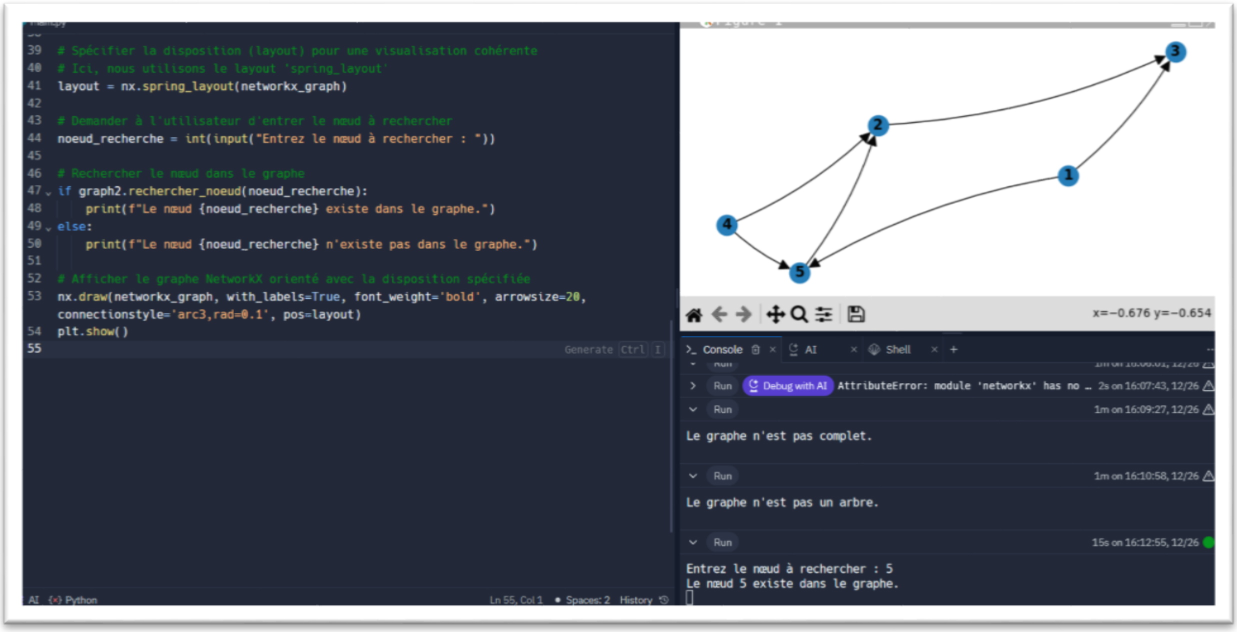

Figure 10: Recherche du Noeud dans le Graphe

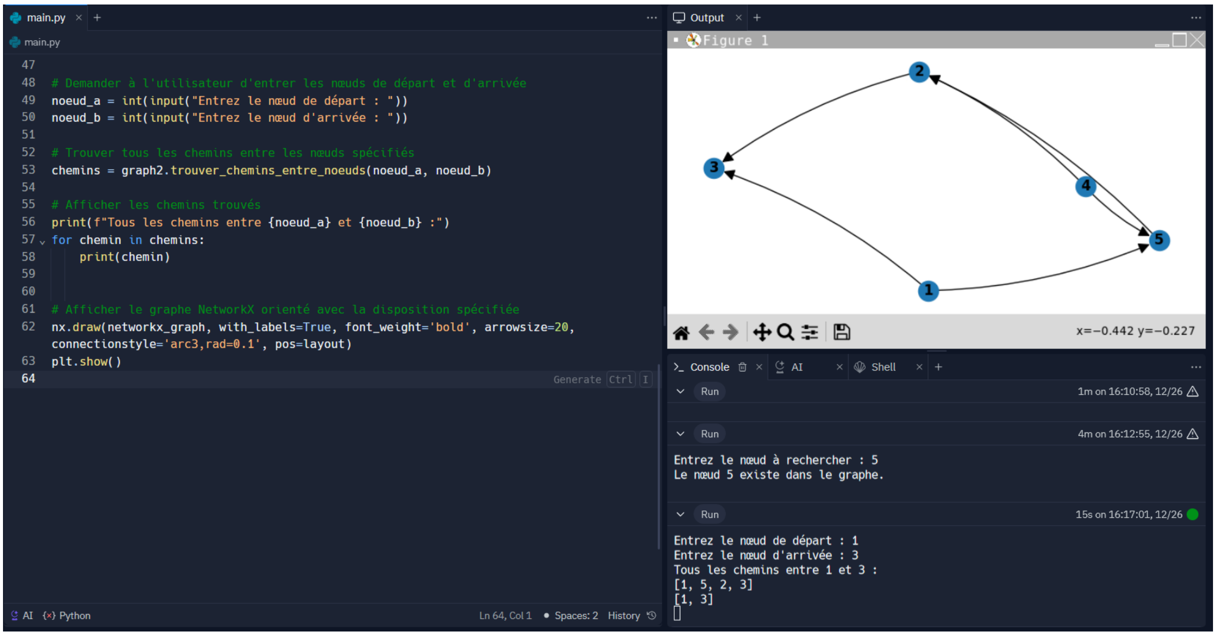

Figure 11: Recherche de Tous les Chemins entre Deux Noeuds dans le Graphe

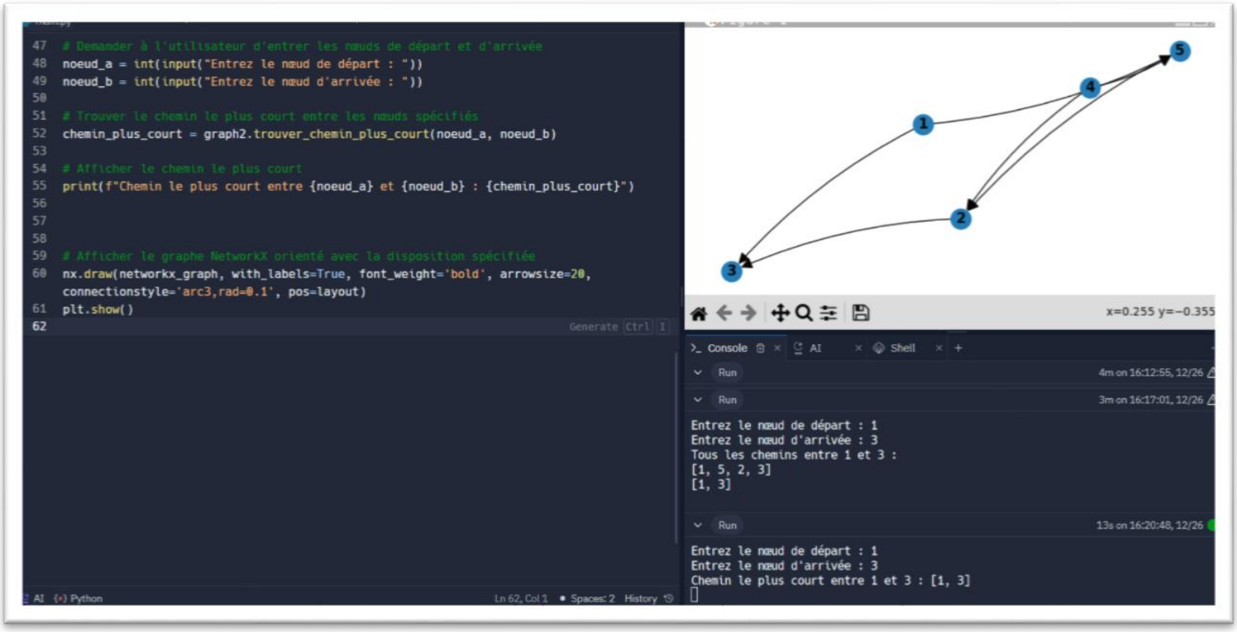

Figure 12: Recherche du Chemin le Plus Court entre les Noeuds dans le Graphe

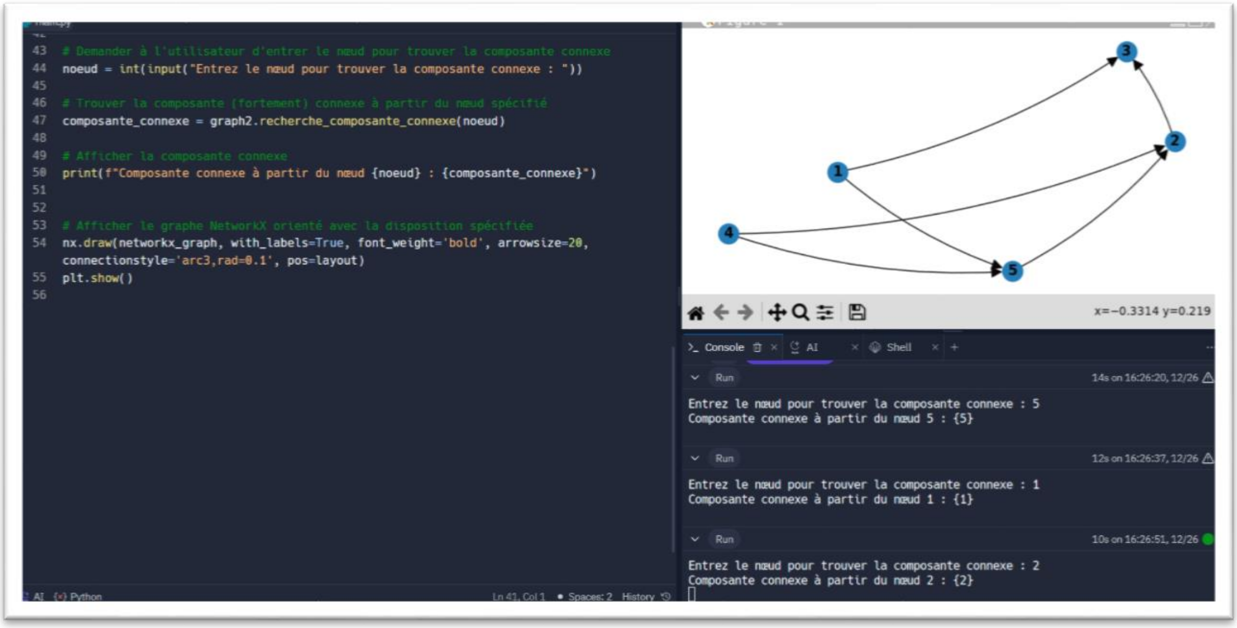

Figure 13: Recherche d'une Composante Connexe à Partir du Noeud

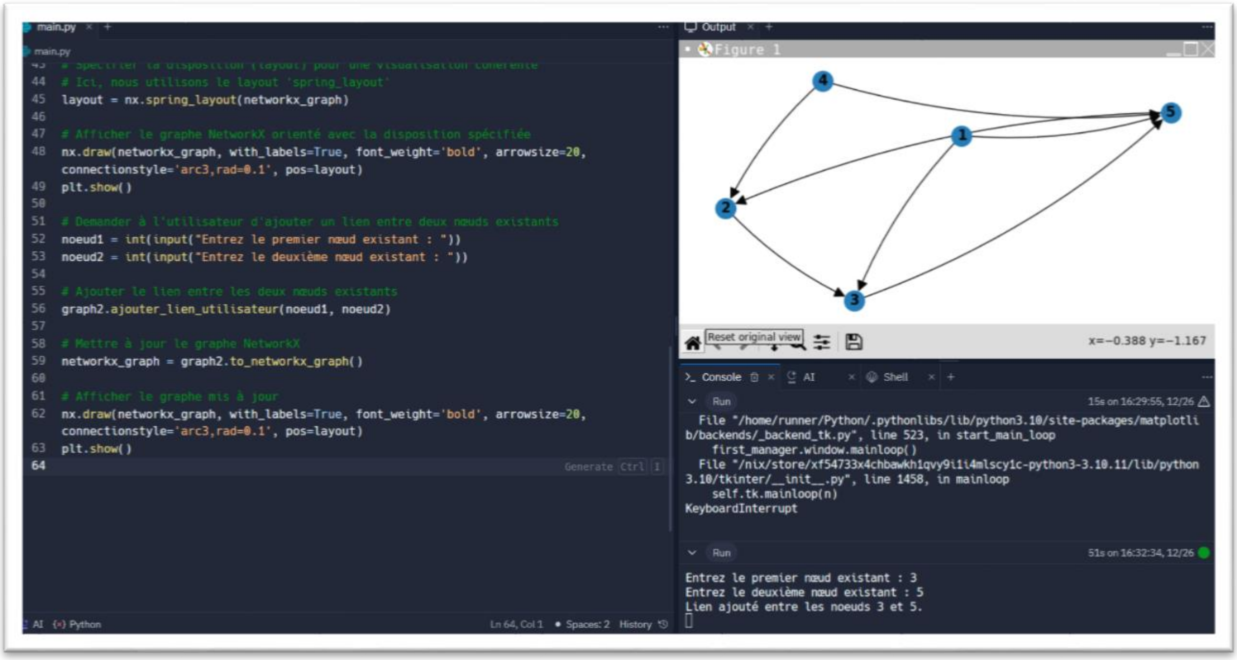

Figure 14: Addition d'un Lien entre Deux Noeuds Existants

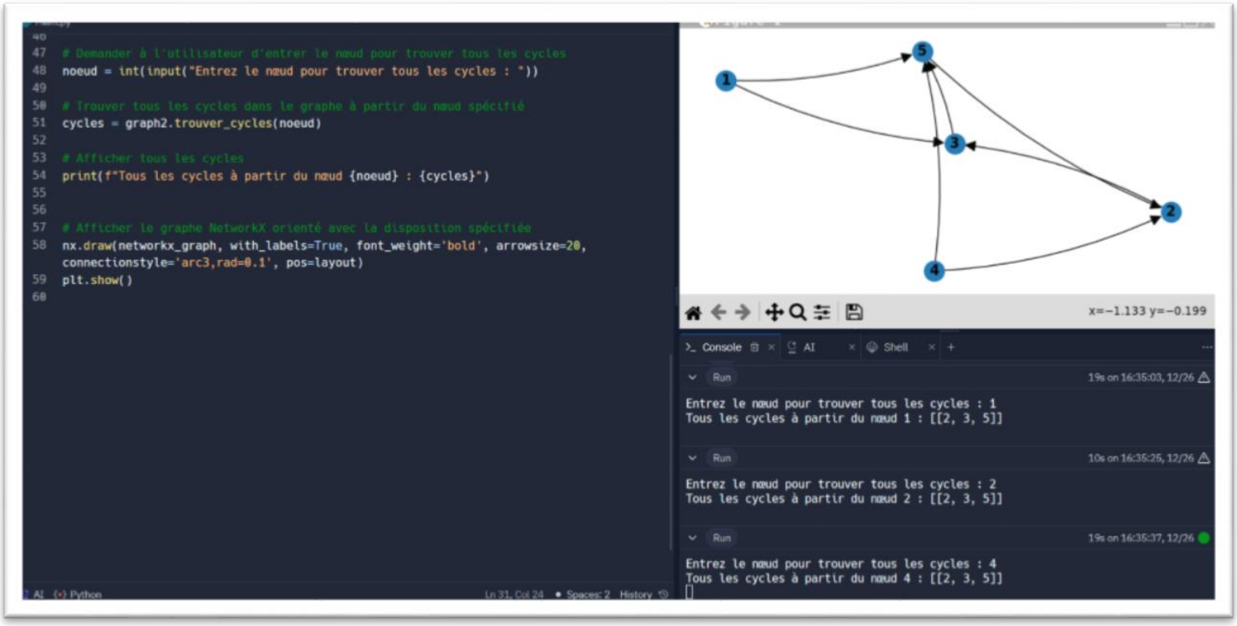

Figure 15: Identification de Tous les Cycles/Circuits dans le Graphe

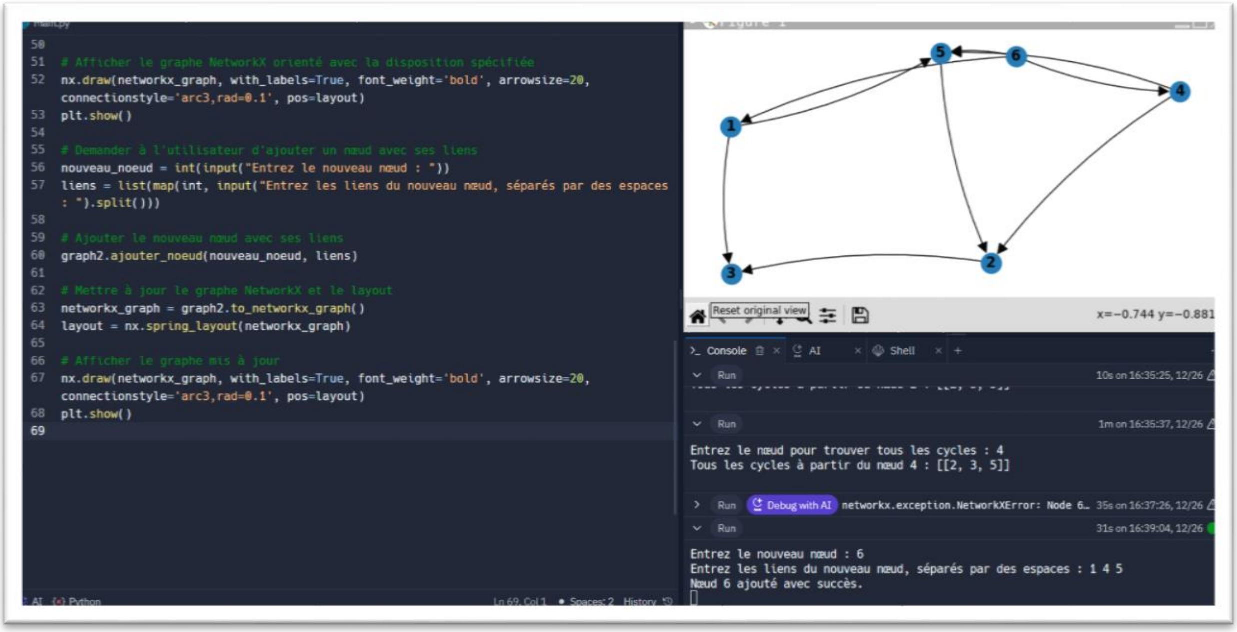

Figure 16: Ajout du Noeud avec ses Liens dans le Graphe

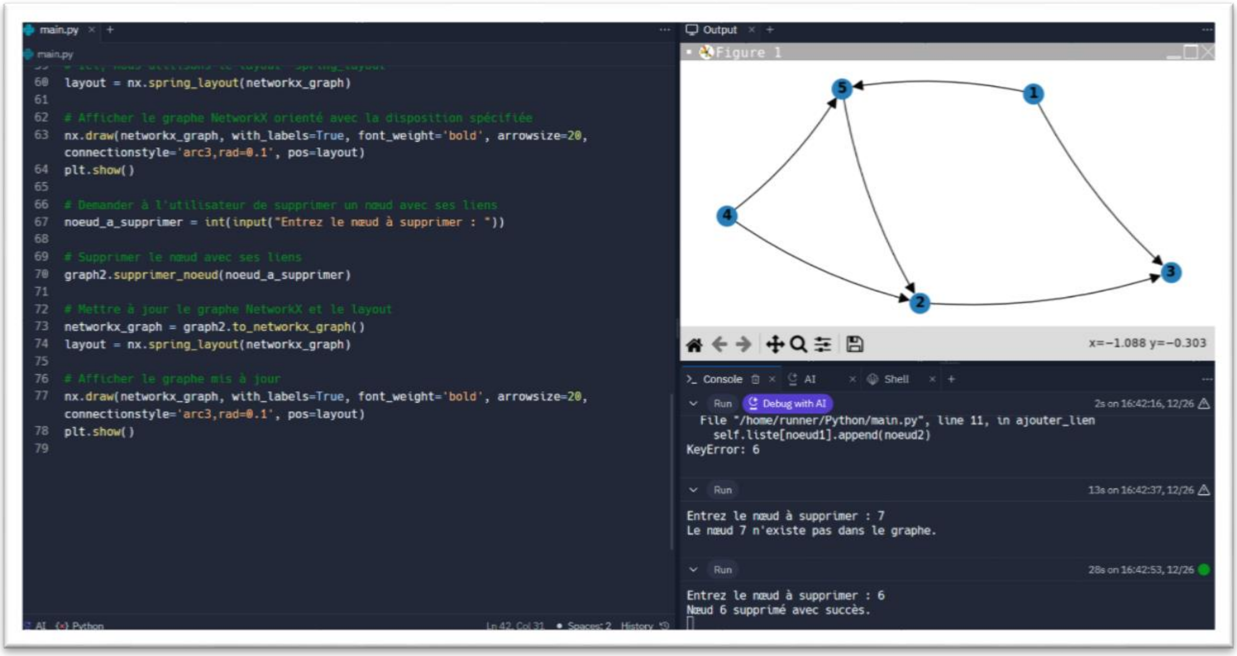

Figure 17: Suppression du Noeud avec ses Liens dans le Graphe

PS : Le graphe reste le même dans toutes les captures d'écran. Après l'exécution de chaque fonction, la fonction d'affichage est exécutée à nouveau pour présenter les différences avant et après la modification.

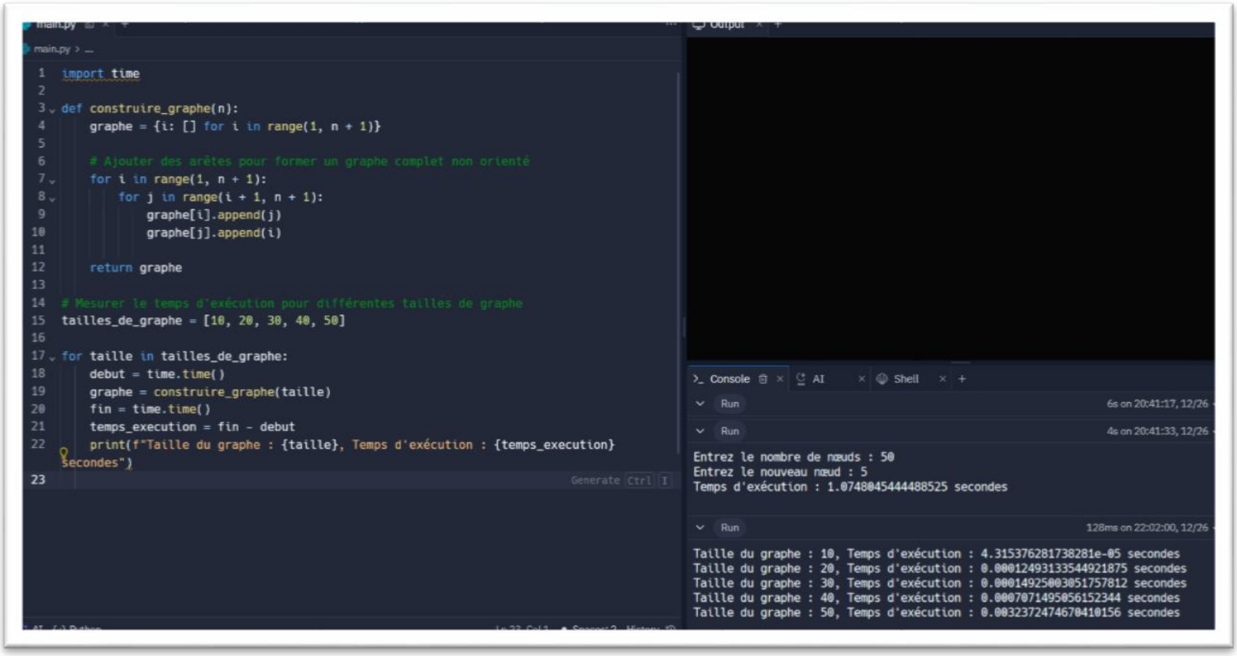

Figure 18: Le temps d'exécution de construction d'un graphe

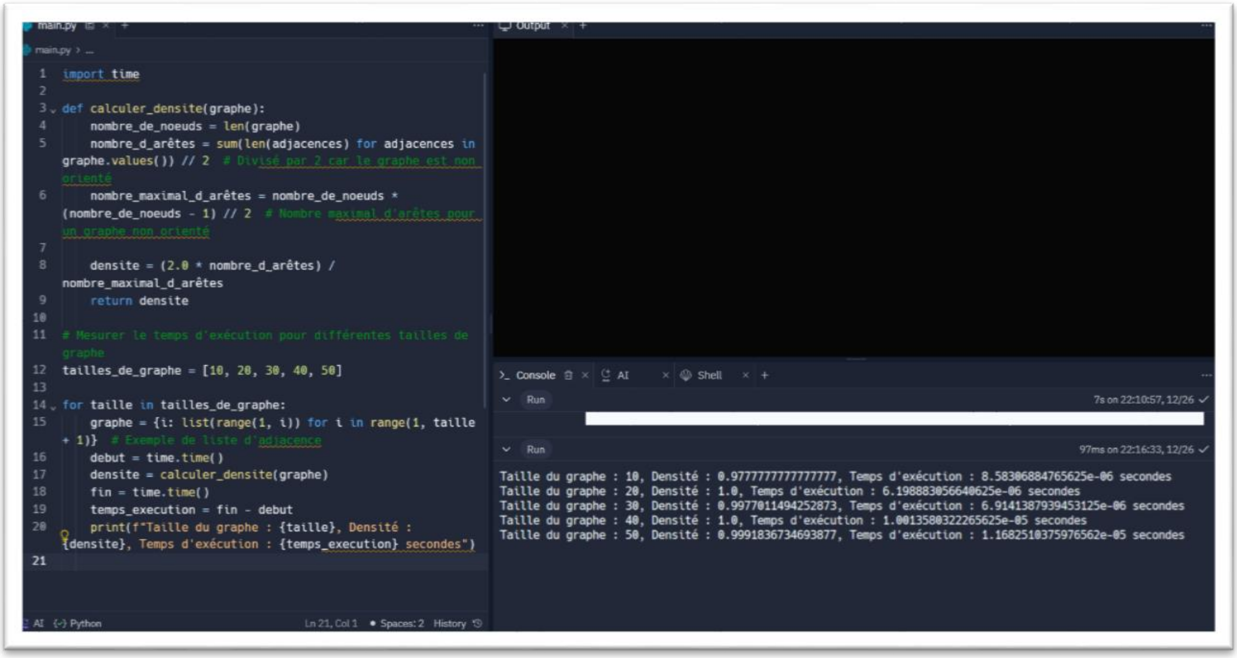

Figure 19: Le temps d'exécution - Calcul densité

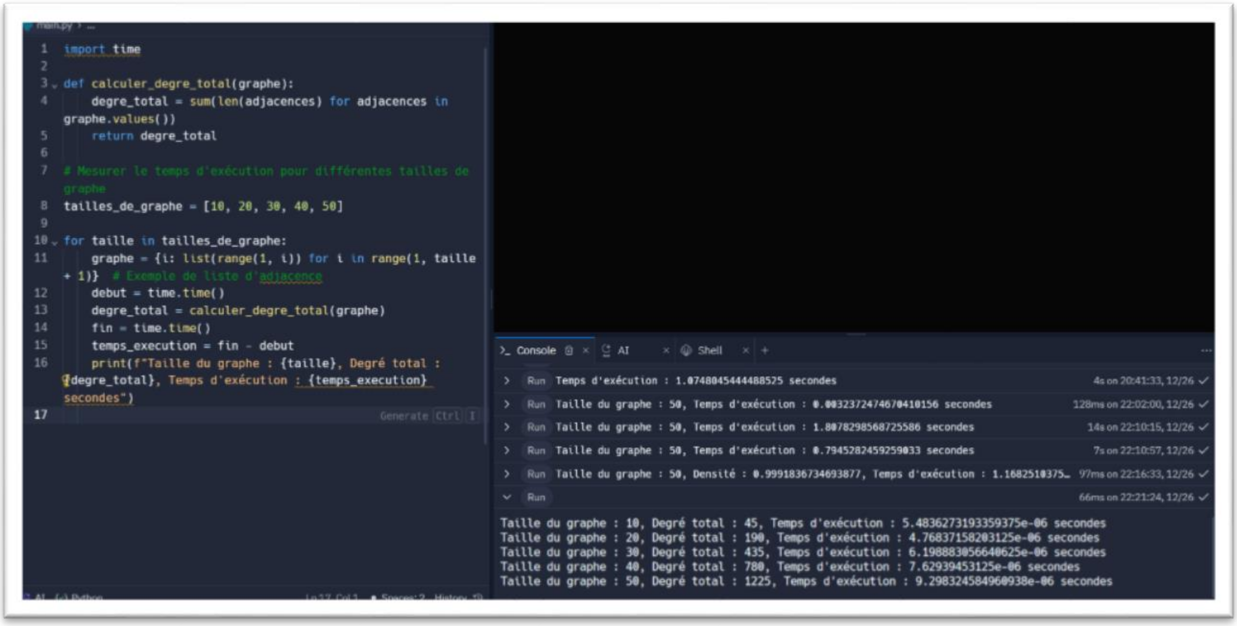

Figure 20: Le temps d'exécution - Calcul degré

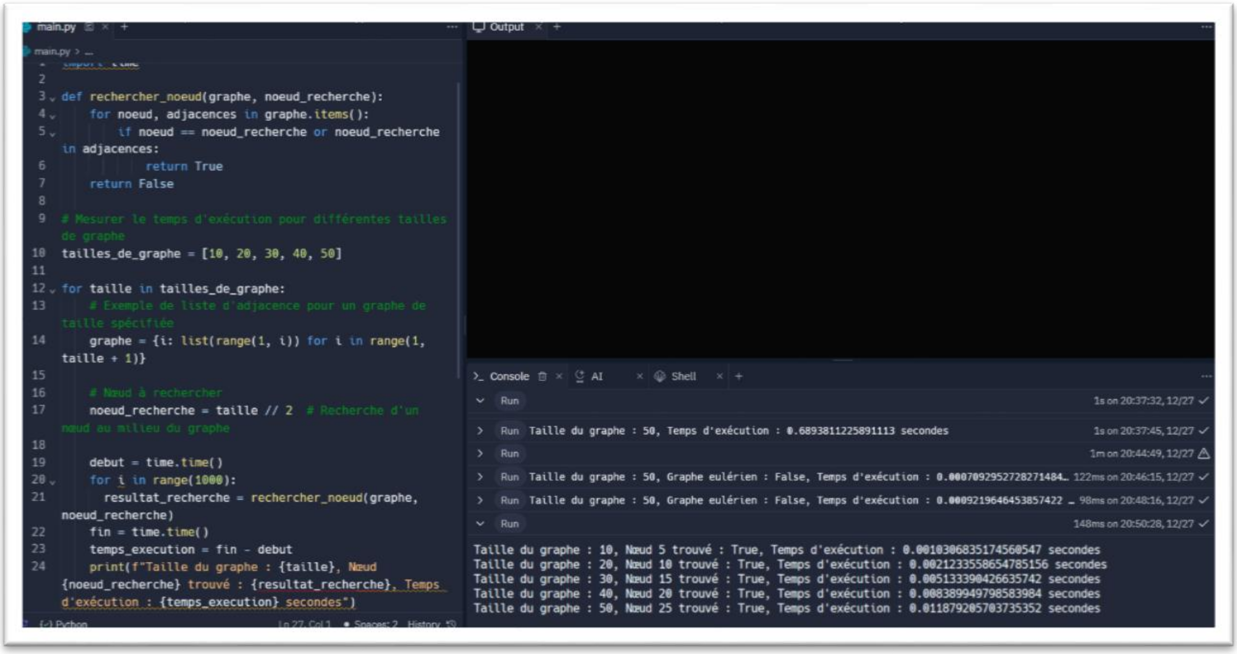

Figure 21: Le temps d'exécution - Recherche un nœud

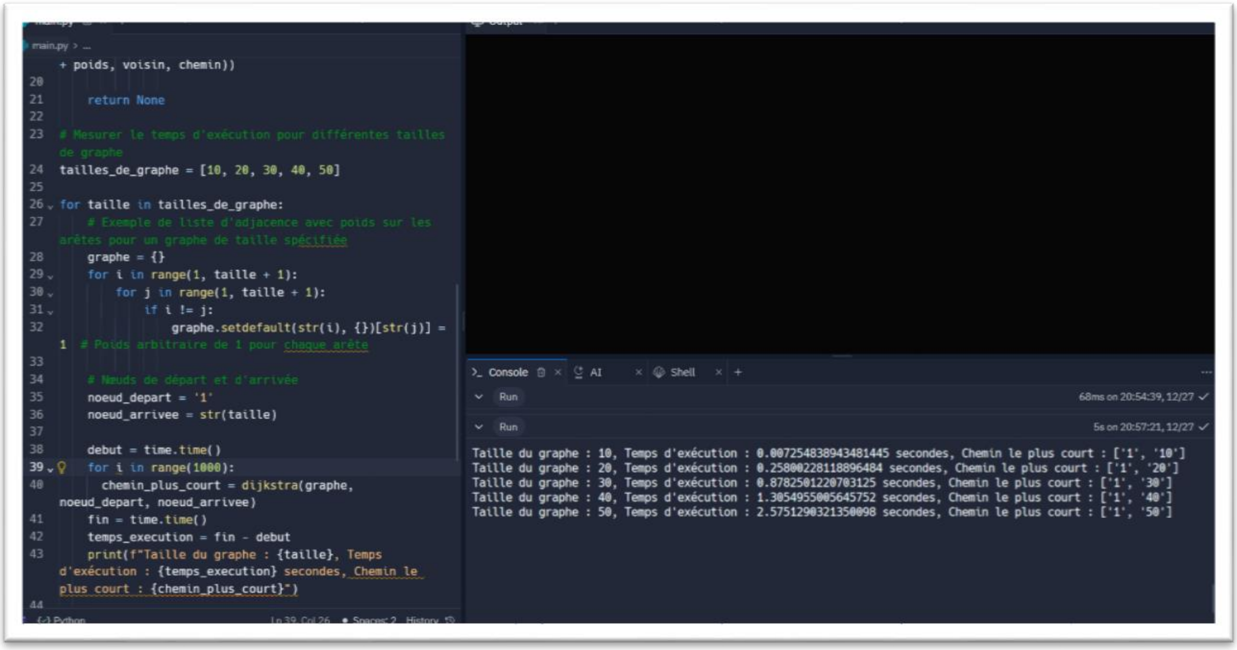

Figure 22: Le temps d'exécution - Recherche chemin court

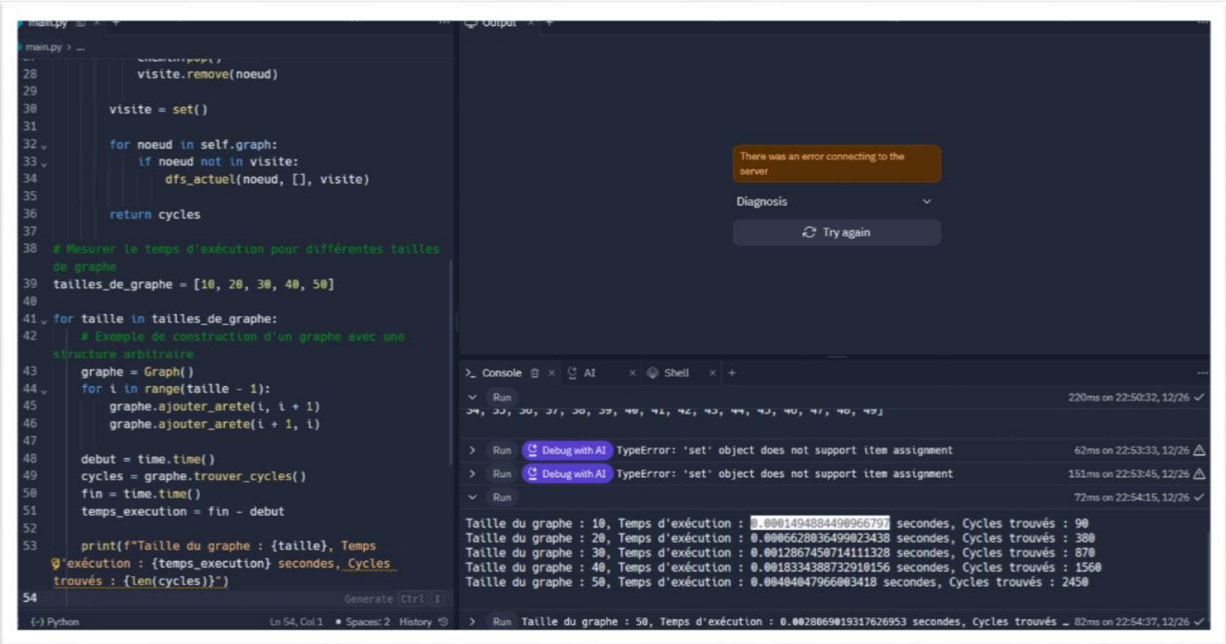

Figure 23: Le temps d'exécution - Identification les cycles