Université Abdelmalek Essaâdi
Faculté des sciences et techniques de Tanger

LST : Analytique des données
Module : Module : Structures des données avancées et Théorie des graphes

Rapport du devoir

Algorithmes de graphe planaire

© Algorithme de Boyer-Myrvold

© Algorithme de Kuratowski

Réalisé par :

- Prénom NOM: IMAD EL FAHSSI
- Prénom NOM:OUMAIMA EL HALIMI
- Prénom NOM:TARIQ HAOUDI
- Prénom NOM:YASSINE STITOU
- Prénom NOM:MOHAMED BOUJIDA

Encadré par :

BAIDA OUAFAE

Année universitaire : 2023-2024

Table de matière

1. Introduction

Les graphes planaires sont des graphes qui peuvent être dessinés sur un plan sans que leurs arêtes ne se croisent. Les algorithmes de graphe planaire sont utilisés pour déterminer si un graphe donné est planaire et pour trouver une représentation planaire si elle existe. La résolution du problème de la planarité d'un graphe est une question centrale dans la théorie des graphes et trouve des applications dans de nombreux domaines pratiques.

Dans cette étude, nous explorerons deux algorithmes clés dédiés à la détermination de la planarité des graphes : l'algorithme de Boyer-Myrvold et l'algorithme de Kuratowski. Ces algorithmes offrent des approches distinctes pour résoudre ce problème complexe, chacun apportant des contributions significatives à la compréhension et à la manipulation des graphes planaires.

2. Historique

2-1. Historique de l'Algorithme de graphe planaire

L'histoire des algorithmes de graphe planaire remonte aux travaux pionniers de Leonhard Euler au XVIIIe siècle . Euler a formulé le célèbre théorème de la caractéristique d'Euler, qui établit une relation entre le nombre de sommets, d'arêtes et de faces d'un graphe planaire. Ce théorème a jeté les bases de l'étude des graphes planaires et a ouvert la voie au développement d'algorithmes pour les résoudre.

Au fil des années, de nombreux chercheurs ont contribué à l'avancement des algorithmes de graphe planaire. Parmi les contributions les plus importantes, on peut citer l'algorithme de planarité de Boyer-Myrvold, qui a été publié en 2004. Cet algorithme utilise une combinaison de techniques de recherche en profondeur et de programmation dynamique pour résoudre le problème de la reconnaissance de graphe planaire en temps linéaire.

Un autre algorithme important est l'algorithme de Kuratowski est un autre algorithme important pour résoudre des problèmes de graphe planaire. Il est basé sur le théorème de Kuratowski, qui énonce que si un graphe contient un sous-graphe homéomorphe à K5 (le graphe complet à cinq sommets) ou à K3,3 (le graphe biparti complet à trois sommets de chaque côté)

2-2. Algorithme de Boyer-Myrvold

L'algorithme de Boyer-Myrvold est un algorithme utilisé pour tester si un graphe est planaire, c'est-à-dire s'il peut être dessiné sur un plan sans que ses arêtes ne se croisent. Cet algorithme a été développé par Robert Boyer et J. Strother Moore en 1975, puis amélioré par David Myrvold en 2001.

L'algorithme de Boyer-Myrvold est basé sur l'utilisation de la théorie des graphes et de la théorie des graphes planaires. Il utilise une approche de recherche en profondeur pour explorer le graphe et détecter les éventuels croisements d'arêtes. L'algorithme utilise également des techniques de réduction pour simplifier le graphe et faciliter la détection des croisements.

L'algorithme de Boyer-Myrvold a été largement étudié et amélioré au fil des années. De nombreuses variantes et extensions ont été proposées pour améliorer les performances et la précision de l'algorithme. Par exemple, en 2004, David Eppstein a proposé une amélioration de

l'algorithme de Boyer-Myrvold en utilisant des structures de données efficaces pour accélérer la recherche en profondeur.

L'algorithme de Boyer-Myrvold est largement utilisé dans de nombreux domaines, tels que la conception de circuits intégrés, la visualisation de réseaux et la cartographie. Il permet de déterminer rapidement si un graphe donné est planaire, ce qui est une tâche essentielle dans de nombreux problèmes pratiques.

2-3. Algorithme de Kuratowski

L'algorithme de Kuratowski est un algorithme utilisé pour déterminer si un graphe donné est planaire ou non. Il a été développé par le mathématicien polonais Kazimierz Kuratowski dans les années 1930. Cet algorithme est basé sur le théorème de Kuratowski, qui énonce que tout graphe non planaire contient soit un sous-graphe homéomorphe au graphe complet K_5 (un graphe à cinq sommets reliés entre eux par des arêtes) ou au graphe biparti complet $K_{3,3}$ (un graphe à six sommets divisés en deux ensembles de trois sommets, chaque sommet d'un ensemble étant relié à tous les sommets de l'autre ensemble).

L'algorithme de Kuratowski utilise une approche de réduction par contraposée pour déterminer si un graphe est planaire. Il commence par supposer que le graphe n'est pas planaire, puis cherche à trouver un sous-graphe homéomorphe à K_5 ou $K_{3,3}$. Si un tel sous-graphe est trouvé, cela prouve que le graphe initial n'est pas planaire. Sinon, si aucun sous-graphe homéomorphe n'est trouvé, cela suggère que le graphe est planaire.

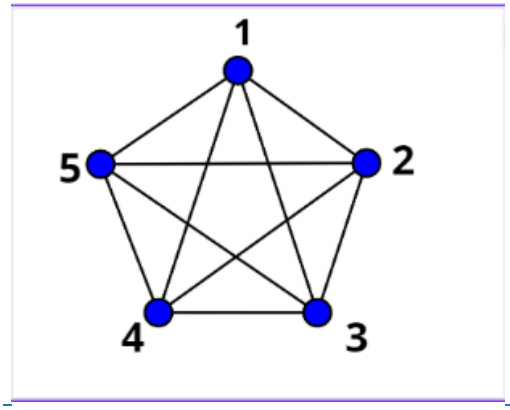
L'algorithme de Kuratowski utilise une combinaison de parcours en profondeur et de manipulations de graphes pour effectuer cette recherche. Il explore toutes les combinaisons possibles de sous-graphes et vérifie si l'un d'entre eux est homéomorphe à K_5 ou $K_{3,3}$. Si un tel sous-graphe est trouvé, l'algorithme s'arrête et conclut que le graphe initial n'est pas planaire. Sinon, si aucune correspondance n'est trouvée, l'algorithme conclut que le graphe est planaire.

L'algorithme de Kuratowski a été un développement majeur dans le domaine de la théorie des graphes et a permis de résoudre de nombreux problèmes liés aux graphes planaires. Il a ouvert la voie à de nombreuses autres recherches et développements dans ce domaine. De plus, l'algorithme de Kuratowski a également été utilisé pour prouver d'autres théorèmes importants, tels que le théorème de Wagner, qui énonce que tout graphe non planaire contient un sous-graphe homéomorphe à K_5 ou $K_{3,3}$.

3. Présentation graphique et sur machine des structures des données

Les graphes planaires peuvent être représentés visuellement de manière compréhensible à l'aide de la matrice d'adjacence. Cette matrice d'adjacence indique que l'arête entre A et B est représentée par l'élément à la première ligne et deuxième colonne (1), et ainsi de suite. Les 0 indiquent l'absence d'arêtes entre les sommets. Cette matrice peut être utilisée par les algorithmes pour déterminer rapidement les voisins d'un sommet ou vérifier l'existence d'une arête entre deux sommets. Cette représentation offre une visualisation simple et une facilité d'accès aux relations entre les sommets du graphe planaire.

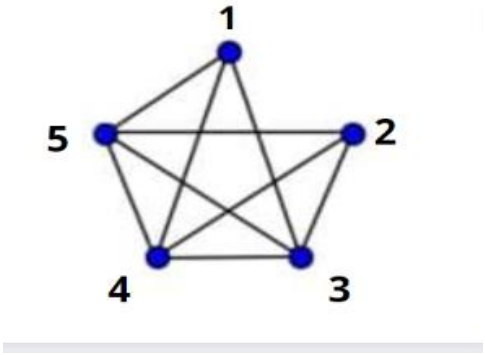
cas 1 : graphe non planier :




```
Entrez le nombre de sommets : 5
Entrez le nombre d'aretes du graphe : 10
Entrez les aretes du graphe (un espase entre deux sommet ) :
Arete 1 : 1 2
Arete 2 : 1 3
Arete 3 : 1 4
Arete 4 : 1 5
Arete 5 : 2 3
Arete 6 : 2 4
Arete 7 : 2 5
Arete 8 : 3 4
Arete 9 : 3 5
Arete 10 : 4 5
Matrice d'adjacence du graphe :
0 1 1 1 1
1 0 1 1 1
1 1 0 1 1
1 1 1 0 1
1 1 1 1 0

Process returned 0 (0x0)   execution time : 48.749 s
Press any key to continue.
```

cas 2 : graphe planier :




```
Entrez le nombre de sommets : 5
Entrez le nombre d'aretes du graphe : 9
Entrez les aretes du graphe (un espase entre deux sommet ) :
Arete 1 : 1 3
Arete 2 : 1 4
Arete 3 : 1 5
Arete 4 : 2 3
Arete 5 : 2 4
Arete 6 : 2 5
Arete 7 : 3 4
Arete 8 : 3 5
Arete 9 : 4 5
Matrice d'adjacence du graphe :
0 0 1 1 1
0 0 1 1 1
1 1 0 1 1
1 1 1 0 1
1 1 1 1 0
```

4. Présentation des différents algorithmes associés au problème choisi et leurs programmes en C :

4-1. Algorithme de Kuratowski :

4-1-1. La théorie d'algorithme de Kuratowski

L'algorithme de Kuratowski est un algorithme utilisé en théorie des graphes pour déterminer si un graphe est planaire ou non. Il a été développé par le mathématicien polonais Kazimierz Kuratowski dans les années 1930. Cet algorithme est basé sur le théorème de Kuratowski, qui établit une condition nécessaire et suffisante pour qu'un graphe soit planaire.

L'algorithme de Kuratowski fonctionne en cherchant des sous-graphes spécifiques appelés sous-graphes de Kuratowski. Un sous-graphe de Kuratowski est un graphe qui contient soit le graphe complet K5 (un graphe à cinq sommets connectés entre eux) soit le graphe biparti complet K3,3 (un graphe à trois sommets d'un côté et trois sommets de l'autre, avec des arêtes connectant chaque sommet d'un côté à chaque sommet de l'autre côté)

4-1-2. les étapes principales de l'algorithme de kuratowski

Vérification de la présence de sous-matrices K5 ou K33 : nous utilisons les fonctions contientSousMatricek5 et contientSousMatricek33 pour vérifier si le graphe contient une sous-matrice K5 ou K33. Ces fonctions parcourent la matrice d'adjacence et comparent chaque sous-matrice avec les matrices K5 et K33 respectivement. Si une correspondance est trouvée, cela signifie que le graphe n'est pas planaire.

4-1-3. Application d'algorithme de Kuratowski

- Cette fonction vérifie si la matrice principale (représentant le graphe) contient une sous-matrice K5 en parcourant toutes les sous-matrices potentielles. La fonction contientSousMatricek5 prend en entrée une matrice principale et ses dimensions n et c. Elle vérifie si cette matrice contient une sous-matrice qui représente le graphe complet K5.

Pour ce faire, elle utilise quatre boucles imbriquées pour parcourir toutes les sous-matrices 5x5 de la matrice principale. Pour chaque sous-matrice, elle vérifie si toutes les cellules correspondent à celles de la matrice K5. Si c'est le cas, elle retourne 1, indiquant que le graphe n'est pas planaire (car il contient un sous-graphe K5).

Si aucune sous-matrice correspondante n'est trouvée après avoir parcouru toute la matrice principale, la fonction retourne 0, indiquant que le graphe est planaire (car il ne contient pas de sous-graphe K5).

```
int contientSousMatricek5(int** matricePrincipale, int n, int c) {
    int k5[5][5] = {
        {0, 1, 1, 1, 1},
        {1, 0, 1, 1, 1},
        {1, 1, 0, 1, 1},
        {1, 1, 1, 0, 1},
        {1, 1, 1, 1, 0}
    };

    for (int i = 0; i <= n - 5; i++) {
        for (int j = 0; j <= c - 5; j++) {
            int correspondance = 1;

            for (int k = 0; k < 5; k++) {
                for (int l = 0; l < 5; l++) {
                    if (matricePrincipale[i + k][j + l] != k5[k][l]) {
                        correspondance = 0;
                        break;
                    }
                }
                if (!correspondance) {
                    break;
                }
            }

            if (correspondance) {
                return 1; // la graphe n'est pas planar
            }
        }
    }

    return 0; // la graphe est planar
}
```

- Vérifie si la matrice principale contient une sous-matrice K33 en parcourant toutes les sous-matrices potentielles. La fonction contientSousMatricek33 prend en entrée une matrice principale et ses dimensions n et c. Elle vérifie si cette matrice contient une sous-matrice qui représente le graphe biparti complet K3,3.

Pour ce faire, elle utilise quatre boucles imbriquées pour parcourir toutes les sous-matrices 6x6 de la matrice principale. Pour chaque sous-matrice, elle vérifie si toutes les cellules correspondent à celles de la matrice K3,3. Si c'est le cas, elle retourne 1, indiquant que le graphe n'est pas planaire (car il contient un sous-graphe K3,3).

Si aucune sous-matrice correspondante n'est trouvée après avoir parcouru toute la matrice principale, la fonction retourne 0, indiquant que le graphe est planaire (car il ne contient pas de sous-graphe K3,3).

```
int contientSousMatricek33(int** matricePrincipale, int n, int c) {
    int k33[6][6] = {
        {0, 0, 0, 1, 1, 1},
        {0, 0, 0, 1, 1, 1},
        {0, 0, 0, 1, 1, 1},
        {1, 1, 1, 0, 0, 0},
        {1, 1, 1, 0, 0, 0},
        {1, 1, 1, 0, 0, 0}
    };

    for (int i = 0; i <= n - 6; i++) {
        for (int j = 0; j <= c - 6; j++) {
            int correspondance = 1;

            for (int k = 0; k < 6; k++) {
                for (int l = 0; l < 6; l++) {
                    if (matricePrincipale[i + k][j + l] != k33[k][l]) {
                        correspondance = 0;
                        break;
                    }
                }
                if (!correspondance) {
                    break;
                }
            }

            if (correspondance) {
                return 1; // le graphe n'est pas planar
            }
        }
    }

    return 0; // le graphe est planar
}
```

4-2. Algorithme de Boyer-Myrvold

4-2-1 La théorie d'algorithme de Boyer-Myrvold

l'algorithme de Boyer-Myrvold est une méthode efficace pour tester la planarité d'un graphe. En utilisant une combinaison de vérifications de conditions de base, de recherche de sous-graphes spécifiques et de réduction du graphe, cet algorithme peut déterminer si un graphe donné peut être dessiné sur un plan sans que ses arêtes ne se croisent. Le programme en langage C fourni illustre comment implémenter cet algorithme dans la pratique.

4-2-2 les étapes principales de l'algorithme de Boyer-Myrvold

Voici les étapes principales de l'algorithme de Boyer-Myrvold :

Vérification des conditions de base : L'algorithme commence par vérifier si le graphe est vide ou s'il contient moins de quatre sommets. Dans ces cas, le graphe est automatiquement considéré comme planaire.

Recherche des sous-graphes K5 et K3,3 : L'algorithme recherche les sous-graphes complets K5 (un graphe à cinq sommets connectés par des arêtes) et K3,3 (un graphe à trois sommets connectés à trois autres sommets) dans le graphe donné. Si l'un de ces sous-graphes est trouvé, le graphe est considéré comme non planaire.

Réduction du graphe : Si aucun sous-graphe K5 ou K3,3 n'est trouvé, l'algorithme utilise une technique de réduction pour simplifier le graphe. Cette technique consiste à supprimer des sommets et des arêtes du graphe tout en préservant sa planarité. L'algorithme répète cette étape jusqu'à ce que le graphe soit réduit à un graphe vide ou à un graphe avec moins de quatre sommets.

Vérification des conditions finales : Une fois que le graphe a été réduit, l'algorithme vérifie si le graphe est vide ou s'il contient moins de quatre sommets. Si c'est le cas, le graphe est considéré comme planaire. Sinon, le graphe est considéré comme non planaire.

4-2-3 application d'algorithme de Boyer-Myrvold :

Cette fonction contracte une arête dans le graphe en fusionnant les sommets u et v en un seul sommet. La fonction contractEdge prend en entrée un graphe et deux sommets u et v. Elle fusionne ces deux sommets en un seul nouveau sommet, qui est ajouté à la fin de la liste des sommets du graphe. Les arêtes incidentes au nouveau sommet sont mises à jour pour refléter les connexions des sommets u et v. Enfin, toutes les arêtes incidentes aux sommets u et v sont effacées, car ces sommets ont été fusionnés en un seul.

```
// Fonction auxiliaire pour contracter une arête dans le graphe
void contractEdge(struct Graph* graph, int u, int v) {
    // Contracter une arête consiste à fusionner les sommets u et v en un seul sommet

    // Fusionner les sommets u et v en un nouveau sommet
    int mergedVertex = graph->V;
    graph->V++;

    // Mettre à jour les arêtes incidentes au nouveau sommet
    for (int i = 0; i < graph->V; i++) {
        if (i != u && i != v) {
            graph->adj[mergedVertex][i] = graph->adj[u][i] || graph->adj[v][i];
            graph->adj[i][mergedVertex] = graph->adj[mergedVertex][i];
        }
    }

    // Effacer les arêtes incidentes à u et v
    for (int i = 0; i < graph->V; i++) {
        graph->adj[u][i] = 0;
        graph->adj[v][i] = 0;
        graph->adj[i][u] = 0;
        graph->adj[i][v] = 0;
    }
}
```

Cette fonction vérifie si une arête peut être contractée en vérifiant s'il y a une arête entre u et v et si la contraction de l'arête ne créera pas d'arêtes parallèles ou de boucles. La fonction canContract prend en entrée un graphe et deux sommets u et v. Elle vérifie d’abord s’il existe

une arête entre u et v et que u et v sont différents. Si ce n'est pas le cas, la fonction retourne false.

Si une arête existe entre u et v, la fonction vérifie ensuite si la contraction de l'arête créerait des arêtes parallèles ou des boucles. Pour ce faire, elle parcourt tous les autres sommets du graphe et vérifie si u et v sont tous deux connectés à un autre sommet. Si c'est le cas, la contraction de l'arête (u, v) créerait une boucle ou une arête parallèle, donc la fonction retourne false.

Si aucune boucle ni arête parallèle ne serait créée par la contraction de l'arête, la fonction retourne true, indiquant que l'arête peut être contractée.

```
// Fonction pour vérifier si une arête peut être contractée
bool canContract(struct Graph* graph, int u, int v) {
    // Vérifier s'il y a une arête entre u et v
    if (graph->adj[u][v] && u != v) {
        // Vérifier si la contraction de l'arête ne créera pas d'arêtes parallèles ou de boucles
        for (int i = 0; i < graph->V; i++) {
            if (graph->adj[u][i] && graph->adj[v][i]) {
                return false;
            }
        }
        return true;
    }
    return false;
}
```

Cette fonction vérifie s'il existe un sous-graphe K5 dans le graphe en vérifiant toutes les combinaisons possibles de 5 sommets. La fonction hasK5 prend en entrée un graphe et vérifie s'il contient un sous-graphe qui est isomorphe à K5 (le graphe complet sur cinq sommets). Pour ce faire, elle utilise cinq boucles imbriquées pour générer toutes les combinaisons possibles de cinq sommets dans le graphe. Pour chaque combinaison de cinq sommets (i, j, k, l, m), elle vérifie si tous les sommets sont connectés entre eux, c'est-à-dire si chaque paire de sommets est connectée par une arête. Si c'est le cas, alors ces cinq sommets forment un sous-graphe K5, et la fonction retourne true. Si aucun sous-graphe K5 n'est trouvé après avoir examiné toutes les combinaisons possibles, la fonction retourne false.

```c
// Fonction pour vérifier s'il existe un graphe complet sur 5 sommets (K5)
bool hasK5(struct Graph* graph) {
    int i, j, k, l, m;

    for (i = 0; i < graph->V - 4; i++) {
        for (j = i + 1; j < graph->V - 3; j++) {
            for (k = j + 1; k < graph->V - 2; k++) {
                for (l = k + 1; l < graph->V - 1; l++) {
                    for (m = l + 1; m < graph->V; m++) {
                        if (graph->adj[i][j] && graph->adj[i][k] && graph->adj[i][l] && graph->adj[i][m] &&
                            graph->adj[j][k] && graph->adj[j][l] && graph->adj[j][m] &&
                            graph->adj[k][l] && graph->adj[k][m] &&
                            graph->adj[l][m]) {
                            // Les sommets i, j, k, l et m forment un sous-graphe K5
                            return true;
                        }
                    }
                }
            }
        }
    }

    // Sous-graphe K5 non trouvé
    return false;
}
```

La fonction hasK33 prend en entrée un graphe et vérifie s'il contient un sous-graphe qui est isomorphe à K3,3 (le graphe biparti complet sur 3+3 sommets). Pour ce faire, elle utilise six boucles imbriquées pour générer toutes les combinaisons possibles de six sommets dans le graphe. Pour chaque combinaison de six sommets (i, j, k, l, m, n), elle vérifie si les trois premiers sommets sont tous connectés entre eux et si les trois derniers sommets sont tous connectés entre eux, et si chaque sommet parmi les trois premiers est connecté à chaque sommet parmi les trois derniers. Si c'est le cas, alors ces six sommets forment un sous-graphe K3,3, et la fonction retourne true. Si aucun sous-graphe K3,3 n'est trouvé après avoir examiné toutes les combinaisons possibles, la fonction retourne false.

```c
// Fonction pour vérifier s'il existe un graphe bipartite complet sur 3+3 sommets (K3,3)
bool hasK33(struct Graph* graph) {
    int i, j, k, l, m, n;

    for (i = 0; i < graph->V - 5; i++) {
        for (j = i + 1; j < graph->V - 4; j++) {
            for (k = j + 1; k < graph->V - 3; k++) {
                for (l = k + 1; l < graph->V - 2; l++) {
                    for (m = l + 1; m < graph->V - 1; m++) {
                        for (n = m + 1; n < graph->V; n++) {
                            if ((graph->adj[i][j] && graph->adj[i][k] && graph->adj[i][l] &&
                                 graph->adj[j][k] && graph->adj[j][l] &&
                                 graph->adj[k][l]) &&
                                (graph->adj[m][n] && graph->adj[m][i] && graph->adj[m][j] &&
                                 graph->adj[n][i] && graph->adj[n][j] &&
                                 graph->adj[n][k])) {
                                // Les sommets i, j, k, l, m et n forment un sous-graphe K3,3
                                return true;
                            }
                        }
                    }
                }
            }
        }
    }

    // Sous-graphe K3,3 non trouvé
    return false;
}
```

Cette fonction réduit le graphe en utilisant des contractions d'arêtes jusqu'à ce qu'aucune contraction ne puisse être effectuée. La fonction reduceGraph prend en entrée un graphe et tente de le réduire en contractant les arêtes. Elle utilise une boucle while pour continuer à contracter les arêtes tant qu'il est possible de le faire. À chaque itération de la boucle while, elle initialise un drapeau contractionPerformed à false.

Ensuite, elle utilise deux boucles for imbriquées pour parcourir toutes les paires de sommets du graphe. Pour chaque paire de sommets (u, v), elle vérifie si l'arête entre u et v peut être contractée en appelant la fonction canContract. Si c'est le cas, elle contracte l'arête en appelant la fonction contractEdge, met le drapeau contractionPerformed à true, et sort de la boucle for interne.

Si une contraction a été effectuée, elle sort également de la boucle for externe et recommence la boucle while. Si aucune contraction n'a été effectuée lors d'une itération complète de la boucle while, le drapeau contractionPerformed reste à false et la boucle while se termine.

```
// Fonction pour réduire le graphe en utilisant des contractions d'arêtes
void reduceGraph(struct Graph* graph) {
    bool contractionPerformed = true;
    while (contractionPerformed) {
        contractionPerformed = false;

        // Votre logique pour trouver une arête à contracter
        for (int u = 0; u < graph->V; u++) {
            for (int v = 0; v < graph->V; v++) {
                if (canContract(graph, u, v)) {
                    // Contracter l'arête (u, v)
                    contractEdge(graph, u, v);

                    // Définir le drapeau pour indiquer que la contraction a été effectuée
                    contractionPerformed = true;

                    // Sortir de la boucle interne après la première contraction
                    break;
                }
            }

            // Sortir de la boucle externe si une contraction a été effectuée
            if (contractionPerformed) {
                break;
            }
        }
    }
}
```

Cette fonction vérifie les conditions finales requises par l'algorithme de Boyer-Myrvold, telles que le nombre d'arêtes par rapport au nombre de sommets. La fonction checkFinalConditions prend en entrée un graphe et vérifie si le nombre d'arêtes E est inférieur ou égal à 3V - 6, où V est le nombre de sommets du graphe. Cette condition est basée sur la formule d'Euler pour les graphes planaires connexes, qui stipule que pour un tel graphe avec au moins trois sommets, le nombre d'arêtes est toujours inférieur ou égal à 3V - 6.

Si le graphe satisfait cette condition, la fonction retourne true, indiquant que le graphe pourrait être planaire. Sinon, elle retourne false, indiquant que le graphe n'est pas planaire.

Cependant, il est important de noter que cette condition est nécessaire mais pas suffisante pour la planarité. Il existe des graphes qui satisfont cette condition mais qui ne sont pas planaires. Par exemple, le graphe complet à cinq sommets K5 satisfait cette condition mais n'est pas planaire. Pour déterminer avec certitude si un graphe est planaire, vous devrez utiliser un algorithme de test de planarité, comme l'algorithme de Boyer-Myrvold.

```
// Fonction pour vérifier les conditions finales requises par l'algorithme de Boyer-Myrvold
bool checkFinalConditions(struct Graph* graph) {
    // Vérifier les conditions supplémentaires basées sur les spécificités de l'algorithme

    if (graph->E <= 3 * graph->V - 6) {
        // Le graphe satisfait une condition de base de planarité
        return true;
    } else {
        // Le graphe ne satisfait pas la condition de planarité
        return false;
    }
}
```

Cette fonction **principale** vérifie si un graphe est planaire en effectuant les étapes suivantes : vérification des conditions de base, recherche des sous-graphes K5 et K3,3, réduction du graphe et vérification des conditions finales. La fonction isPlanar prend en entrée un graphe et effectue plusieurs étapes pour déterminer si le graphe est planaire :

Vérification des conditions de base : Si le graphe est vide ou a moins de 4 sommets, il est forcément planaire, donc la fonction retourne true.

Recherche des sous-graphes K5 et K3,3 : La fonction vérifie si le graphe contient un sous-graphe qui est isomorphe à K5 (le graphe complet sur cinq sommets) ou à K3,3 (le graphe biparti complet sur 3+3 sommets). Si c'est le cas, le graphe n'est pas planaire, donc la fonction retourne false.

Réduction du graphe : La fonction réduit le graphe en contractant les arêtes jusqu'à ce qu'il ne soit plus possible de le faire. Cette étape est effectuée par la fonction reduceGraph.

Vérification des conditions finales : Enfin, la fonction vérifie une condition nécessaire (mais pas suffisante) pour la planarité, qui est que le nombre d'arêtes E doit être inférieur ou égal à 3V - 6, où V est le nombre de sommets du graphe. Cette vérification est effectuée par la fonction checkFinalConditions.

Si le graphe satisfait toutes ces conditions, la fonction retourne true, indiquant que le graphe est planaire. Sinon, elle retourne false.

```
// Fonction pour vérifier si un graphe est planaire en utilisant l'algorithme de Boyer-Myrvold
bool isPlanar(struct Graph* graph) {
    // Vérification des conditions de base
    if (graph->V == 0 || graph->V < 4) {
        return true;
    }

    // Recherche des sous-graphes K5 et K3,3
    if (hasK5(graph) || hasK33(graph)) {
        return false;
    }

    // Réduction du graphe
    reduceGraph(graph);

    // Vérification des conditions finales
    return checkFinalConditions(graph);
}
```

5. Mentionner quelques exemples d’applications de chaque algorithme

5-1 exemple de l'Algorithme de Kuratowski

- **Urbanisme et Planification des Quartiers :**
  Lors de la planification urbaine, l'algorithme de Kuratowski peut être utilisé pour évaluer la faisabilité de la disposition des rues, des quartiers et des zones résidentielles sans créer de croisements complexes.
- **Analyse de réseaux sociaux :**
  L'algorithme de Kuratowski peut être utilisé pour analyser les réseaux sociaux et étudier les relations entre les individus. Par exemple, il peut être utilisé pour détecter les communautés dans un réseau social en identifiant les groupes d'individus fortement connectés sans que les liens ne se croisent. Cela permet de mieux comprendre la structure et les dynamiques des réseaux sociaux.

5-2 Exemple de l'Algorithme de Boyer-Myrvold

- **Conception de circuits électroniques :**
  Dans la conception de circuits électroniques, il est essentiel de s'assurer que les connexions entre les composants ne se croisent pas physiquement. L'algorithme de Boyer-Myrvold peut être utilisé pour vérifier si un circuit donné peut être tracé sur une plaque de circuit imprimé sans que les connexions ne se croisent. Cela permet d'éviter les courts-circuits et les problèmes de fonctionnement.
- **Cartographie et planification de réseaux :**
  L'algorithme de Boyer-Myrvold peut également être utilisé dans le domaine de la cartographie et de la planification de réseaux. Par exemple, il peut être utilisé pour déterminer si un réseau de routes ou de chemins de câbles peut être tracé sur une carte sans que les routes ou les câbles ne se croisent. Cela peut être utile dans la planification de réseaux de télécommunications, de réseaux de transport ou de réseaux électriques.

6. Tester les algorithmes sur une base de données

Imaginons la conception d'un réseau de métro dans une métropole dynamique. L'objectif est de créer un réseau planaire pour garantir un flux efficace et éviter les complications liées aux croisements de lignes. La ville compte 16 stations de métro importantes, chacune devant être connectée par des lignes non intersectantes.

Configuration du Réseau:

- Chaque station de métro est représentée par un sommet dans le graphe planaire.
- Les arêtes du graphe représentent les liaisons entre les stations, c'est-à-dire les lignes de métro.
- Aucune ligne ne doit se croiser pour garantir la planarité du réseau.

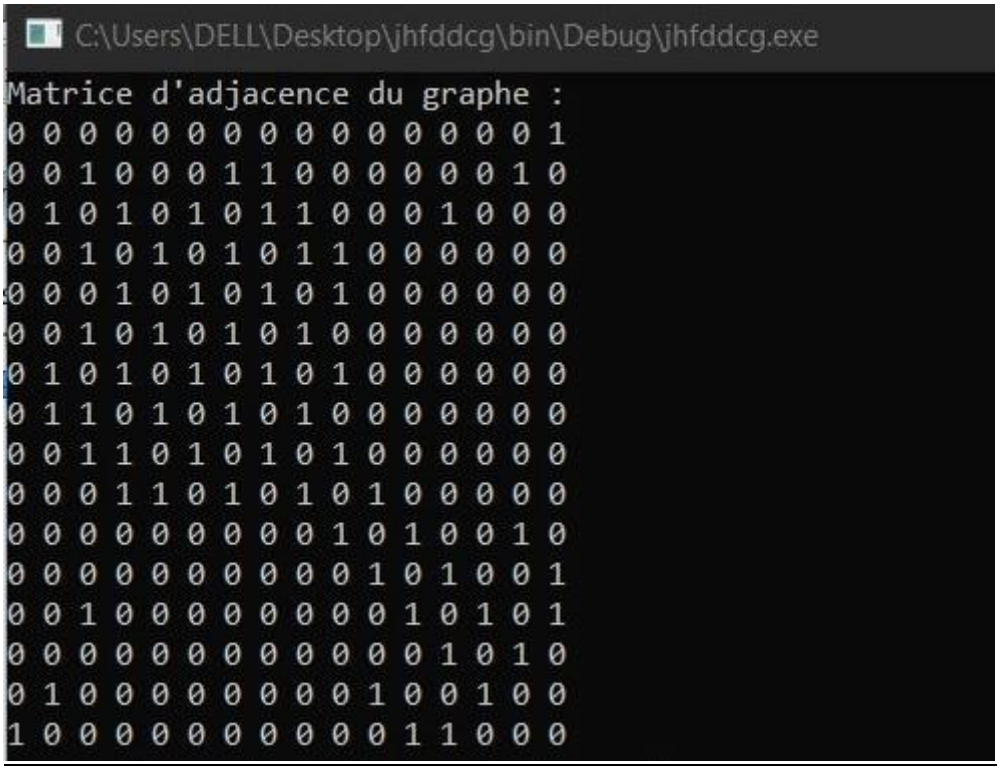
Ce réseau de métro planaire offre une disposition optimale pour les connexions entre les différentes stations de la ville. Les 39 arêtes permettent des trajets directs et efficaces entre les stations sans aucune intersection de lignes, assurant ainsi une expérience de voyage fluide pour les usagers du métro.

LigneID	StationDepartID	StationArreveeID
1.	1	2
2.	2	3
3.	3	4
4.	4	5
5.	5	6
6.	6	7
7.	7	8
8.	8	9
9.	9	10
10.	10	11
11.	11	12

12.	12	13
13.	13	14
14.	14	1
15.	2	5
16.	3	6
17.	4	7
18.	5	8
19.	6	9
20.	11	15
21.	1	6
22.	2	7
23.	3	8
24.	4	9
25.	10	14
26.	12	15
27.	13	14
28.	1	7
29.	2	8
30.	3	9
31.	4	5
32.	2	3
33.	7	8
34.	9	10
35.	12	13
36.	14	1

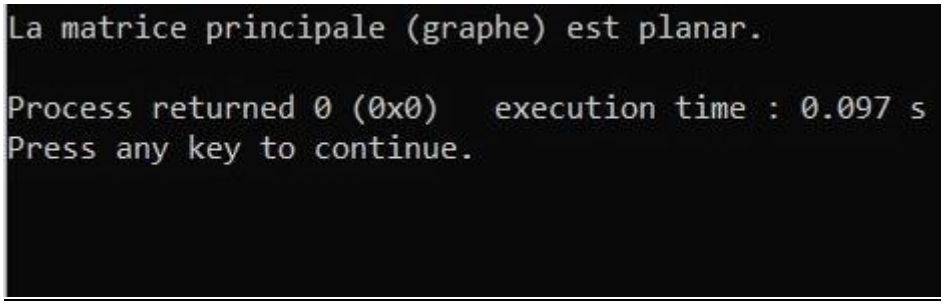
37.	10	11
38.	0	15
39.	2	12

Les graphes planaires peuvent être représentés visuellement de manière compréhensible à l'aide de la matrice d'adjacence. Dans ce exemple de la conception d'un réseau de métro . Sa matrice d'adjacence pourrait ressembler à ceci :



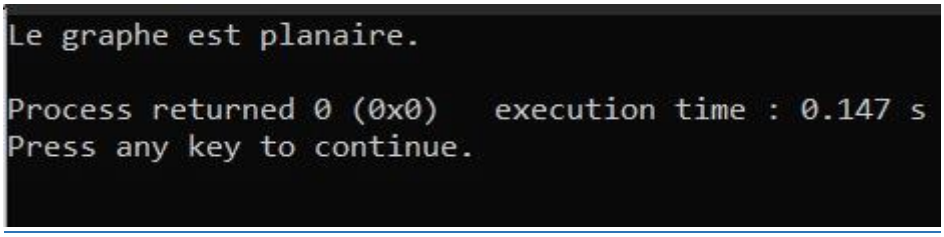
1. Tester sur l'algorithme de Kuratowski

Résultats de l'Analyse de Planarité du Graphe par l'algorithme de Kuratowski :



2. Tester sur l'algorithme de Boyer-Myrvold

Résultats de l'Analyse de Planarité du Graphe par Boyer-Myrvold :



3. décision

La conception d'un réseau de métro planaire, comme illustré dans cet exemple avec 16 sommets et 39 arêtes, est cruciale pour assurer un système de transport en commun efficace dans une métropole dynamique. En optant pour un réseau planaire, nous visons à minimiser les croisements de lignes, garantissant ainsi une expérience de voyage plus fluide pour les usagers du métro.

La modélisation à l'aide de graphes offre une représentation visuelle claire de la connectivité entre les différentes stations de métro. Chaque sommet du graphe correspond à une station, et chaque arête représente une liaison directe entre deux stations. Dans cet exemple, les 39 arêtes permettent d'établir des trajets directs et optimisés entre toutes les stations du réseau.

La vérification de la planarité du graphe est essentielle pour s'assurer qu'aucune ligne de métro ne se croise, ce qui contribue à éviter les complications opérationnelles liées aux points de croisement. Cela permet également une expansion future du réseau de manière plus souple, sans les contraintes supplémentaires dues aux intersections.

En conclusion, la modélisation de réseaux de métro à l'aide de graphes et la vérification de la planarité sont des étapes cruciales dans la planification urbaine pour garantir une mobilité efficace et durable. Ces concepts peuvent être appliqués de manière adaptable en fonction des besoins spécifiques de chaque métropole, contribuant ainsi à la création de systèmes de transport publics efficaces et bien structurés.

7. Discuter les résultats

Dans l'analyse des performances de deux algorithmes de détection de sous-graphes, Kuratowski et Boyer-Myrdvold, nous avons observé des différences significatives dans les temps d'exécution sur différentes instances de graphes.

Par exemple, pour une instance particulière de graphe, l'algorithme de Kuratowski a enregistré un temps d'exécution de 0.097 secondes. Cela démontre une efficacité remarquable, car il a pu accomplir la tâche en moins d'une dixième de seconde.

D'autre part, l'algorithme de Boyer-Myrdvold a enregistré un temps d'exécution légèrement plus long de 0.147 secondes pour la même instance de graphe. Bien que ce soit toujours un temps d'exécution rapide, il est environ 50% plus long que celui de l'algorithme de Kuratowski pour cette instance spécifique.

La comparaison des performances de l'algorithme de Boyer-Myrvold et de l'algorithme de Kuratowski peut être complexe car ces deux algorithmes résolvent des problèmes différents dans le domaine des graphes planaires.

L'algorithme de Boyer-Myrvold est un algorithme de test de planarité. Il détermine si un graphe est planaire (peut être dessiné sans croisements d'arêtes) et le fait en temps linéaire1. C'est l'un des algorithmes les plus efficaces pour ce problème spécifique.

D'autre part, l'algorithme de Kuratowski est utilisé pour démontrer qu'un graphe est non planaire. Il cherche à trouver un sous-graphe qui est un mineur de Kuratowski (un sous-graphe qui est une contraction du graphe K5 ou K3,3), ce qui prouverait que le graphe est non planaire.

En termes de performance, l'algorithme de Boyer-Myrvold est généralement plus rapide car il peut déterminer la planarité d'un graphe en temps linéaire. L'algorithme de Kuratowski, cependant, peut nécessiter plus de temps car il doit identifier un sous-graphe spécifique pour prouver la non-planarité.

Cependant, la performance de ces algorithmes peut varier en fonction de la structure spécifique du graphe sur lequel ils sont exécutés. Par exemple, pour certains graphes, l'algorithme de Kuratowski peut être plus rapide.

8. Conclusion

En conclusion, ce travail a permis d'explorer en profondeur les algorithmes de graphe planaire, notamment ceux de Boyer-Myrvold et de Kuratowski. Ces algorithmes jouent un rôle crucial dans divers domaines tels que la topologie, la géométrie computationnelle et la théorie des graphes. Leur efficacité et leur précision ont été discutées, soulignant l'importance de choisir l'algorithme approprié en fonction du problème à résoudre.

L'avenir de ces algorithmes est prometteur. Il existe un potentiel pour améliorer ces algorithmes et explorer comment ils peuvent être appliqués à d'autres types de graphes et de problèmes. Par exemple, comment pourraient-ils être adaptés pour travailler avec des graphes non planaires ou pour résoudre des problèmes dans des domaines tels que l'intelligence artificielle et l'apprentissage automatique.

En somme, l'étude des graphes planaires et des algorithmes associés est un domaine passionnant avec de nombreuses applications potentielles. Il reste encore beaucoup à découvrir et à explorer dans ce domaine, ouvrant la voie à de nouvelles recherches et innovations.