République Algérienne Démocratique et Populaire
Ministère de l'enseignement supérieur et de la recherche scientifique

UNIVERSITÉ Dr MOULAY TAHAR SAIDA
FACULTÉ: TECHNOLOGIE
DÉPARTEMENT: INFORMATIQUE

MÉMOIRE DE MASTER

OPTION : Modélisation Informatique des Connaissances et du Raisonnement (MICR)

Graph Matching: État de l'art et implémentation de VF2

Présenté par :
Lakhache Mohamed Abdeldjallil
Gacem Fatima Zohra

Encadré par :
Mr. **Rahmani Mohamed**

Soutenu le 21/09/2021

Année universitaire : 2020 — 2021

Table des matières

Remerciements

Nous tenons à remercier très sincèrement Messieurs les professeurs du département de l'informatique de la faculté des sciences et de technologie de l'Université Moulay Tahar de Saida. Nos plus vifs remerciements s'adressent à Monsieur Mohamed Rahmani, notre encadreur de ce travail pour son suivi, ses remarques et suggestions.

Dédicaces

Je dédie ce mémoire spécialement à mon père, ma mère qui m'a très encouragé, ma sœur Douae Nour El Houda et toute ma famille et mes amis . ***Lakhache Mohamed Abdeldjallil***

Je dédie spécialement ce travail:
A mon père, ma mère qui m'a très encouragé
Mes sœurs Soumia, Lamia, Ikram et Razan
Mes frères Kader, Anes et Abdelbassit
Gacem Fatima Zohra

Abstract

The goal of our work is to establish a state of the art on the graph matching algorithm. Two major categories of approaches exist for solving the graph matching problem: exact matching and approximatif matching. In the first category, we look for the isomorphism that requires the preservation of graph structures while the second category its goal is to compute a measure of similarity to determine how similar are two graphs.
Our work consists of studying and then implementing the $VF2$ algorithm in Python.

Keywords:
Matrching; graph; isomorphism; mapping,algorithm VF2.

Résumé

L'objectif de notre travail est d'établir un état de l'art sur les algorithmes d'appariement de graphes. Deux grandes catégories d'approches existent pour la résolution du problème de mise en correspondances des graphes: l'appariement exact et l'appariement inexact. Dans la première catégorie, on cherche l'isomorphisme qui nécessite la préservation des structures des graphes alors que la deuxième catégorie son but est de de calculer une mesure de similarité pour déterminer à quel point deux graphes se ressemblent.

Notre travail consiste à étudier puis implémenter l'algorithme $VF2$ en Python.

Mot clés :

Appariement ; graphe ; isomorphisme ; mise en correspondance, algorithme VF2.

Introduction générale

0.1 Introduction et problématique

La nature n'utilise que les fils les plus longs pour tisser ses motifs, de sorte que chaque petit morceau de son tissu révèle l'organisation de toute la tapisserie."Richard P. Feynman"
Les graphes, sont des outils universels utilisés très largement dans le domaine de la vision par ordinateur et de la reconnaissance de formes. Le problème de l'appariement de graphes est de trouver une correspondance entre les sommets d'un graphe et les sommets d'un autre graphe qui satisfasse à certaines contraintes ou critères d'optimalité. "Bunke, 1999".

0.2 L'objectif de l'étude

En raison de leur capacité à représenter une grande variété de problèmes, de nombreuses recherches se sont concentrées sur le développement de techniques de stockage, d'analyse et d'appariement de graphes. Et avec l'importance croissante des données structurées, le traitement des requêtes sur les graphes est devenu un sujet très important.[1]

0.3 Organisation de mémoire

Ce manuscrit suit l'organisation suivante :
Le Chapitre 1 : on a trouvé utile de fournir quelques notions de base sur la théorie de graphes où nous citons quelques définissions liées aux graphes .
Le Chapitre 2 : un état de l'art des méthodes d'appariement de graphes c'est à dire les catégories d'appariement de graphes.
Le Chapitre 3 :Nous discuterons d'abord de la structure de VF2 qui a été héritée par VF3, puis nous discuterons de ce qui a été amélioré par VF3.
Le Chapitre 4 : Nous présentons les résultats que nous avons obtenus .

Chapitre 1

Notions de base sur les graphes

1.1 Introduction

La théorie des graphes constitue un outil puissant pour schématiser les modèles des liens et relations entre les objets. L'étude des graphes a commencé depuis le 18éme siècle par un problème de curiosité mathématique lorsqu'Euler a posé le célèbre problème du pont de Königsberg (Kaliningrad). Ces deux dernières décennies la théorie des graphes a suscité un intérêt exponentiel essentiellement grâce a son rôle comme des modèles d'optimisation et de Calculs explicites nécessitant la conception et l'analyse de plusieurs algorithmes. En outre de son rôle l'éminent dans l'informatique, les mathématiques appliquées (analyse numérique matricielle), la biologie, la physique (circuits électriques), la chimie..., la théorie des graphes est devenue l'un des instruments les plus efficaces pour résoudre de nombreux problèmes discrets que pose de nombreux théories très utiles telles que la recherche opérationnelle et l'économie. Autrement dit elle contribue à résoudre de nombreux problèmes concrets de la vie courante.

1.2 Théorie de graphe

Définition 1.2.1. *Un graphe est une paire (S,A), où :*
**S est un ensemble de nœuds appelés sommets,*
**A est un multi-ensemble de paires de sommets appelées arêtes,*
**On peut voir les sommets et les arêtes comme des positions gardant des objets en mémoire.*

1.2.1 Graphe orienté

Un graphe peut être orienté, une arête est alors appelée un arc. Un arc est défini par un couple ordonné (x_i, x_j) de sommets.

Remarque 1.2.1. *À tout graphe orienté, on peut associer un graphe simple.*

1.2.2 Degré d'un sommet

Le degré d'un sommet S est égal au nombre d'arcs entrants ou sortants de S.

Représentation : $d(S)$ Le demi- degré extérieur (ou degré d'émission) de S est égal au nombre d'arcs sortants de S.

Représentation : $d+(S)$ Le demi-degré intérieur (ou degré de réception) de S égal au nombre d'arcs entrants à S.

Représentation : d-(S). Le degré du sommet s est :

$$d(S) = (d+(S)) + (d-(S))$$

1.2.3 Graphe non orienté

Un graphe non orienté $G = (S, A)$ est la donnée :

- d'un ensemble S dont les éléments sont les sommets du graphe,
- d'un ensemble A dont les éléments, les arêtes du graphe, sont des parties à un ou deux éléments de S.

Le ou les sommets d'une arête sont appelés extrémités de l'arête. Les arêtes n'ayant qu'une seule extrémité sont des boucles.

On peut de la même façon un graphe non-orienté multi-arêtes. Formellement, $G = (S, A, \alpha)$ est la donnée :

- d'un ensemble S dont les éléments sont des sommets ;
- d'un ensemble A dont les éléments sont les arêtes ;
- d'une fonction α de A dans les parties à un ou deux éléments de S.

1.2.4 Chaines, cycles et boucles

Chaines

Définition 1.2.2. *Une suite de sommets reliés par des arêtes ne pouvant passer qu'une seule fois au maximum par arc et par conséquent, ayant une longueur maximale égale au nombre d'arcs d'un graphe.*

Cycles

Définition 1.2.3. *C'est la chaine fermée, le sommet initial est le même sommet final.*

Boucles

Définition 1.2.4. *Une boucle est une arête ayant deux fois le même sommet comme extrémité.*

Exemple 1.2.1. *Dans le graphe de la figure 1.1 : j est une boucle.*

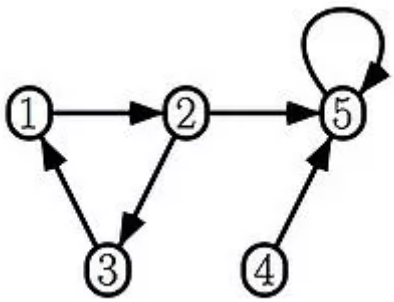


Figure 1.1: Exemple de boucle

1.2.5 Chemins et circuits

Chemins

Définition 1.2.5. *Un chemin est une chaîne dirigée. En-ce qui concerne les chaînes, nous les appelons chemin simple, c'est-à-dire le chemin qui au plus, passe par un sommet.*

Circuits

Définition 1.2.6. *Le cercle est un chemin statique simple et le sommet initial est le même que le sommet final.*

1.2.6 Structures de données pour les graphes

Matrice d'incidence

Soit G un graphe orienté qui possède n sommets numérotés de 1 à n et m arcs numérotés de 1 à m. On appelle matrice d'incidence du graphe la matrice $A = (a_{ij})$ comportant n lignes et m colonnes telle que :

- a_{ij} vaut $+1$, si l'arc numéroté j admet le sommet i comme origine ;
- a_{ij} vaut -1, si l'arc numéroté j admet le sommet i comme arrivée ;
- a_{ij} vaut 0 dans les autres cas.

Matrice d'adjacence

Définition 1.2.7. *La matrice d'adjacence généralise la structure liste d'arête dans le sens suivant :*

- *Chaque nœud S garde en mémoire un entier désignant la position du sommet dans la séquence.*
- *Matrice d'adjacence. En (i, j) pointeur vers l'arête reliant les sommets i et j ou 0 si les sommets ne sont pas adjacents.*

Exemple 1.2.2. *Le graphe de la figure 1.2 :*

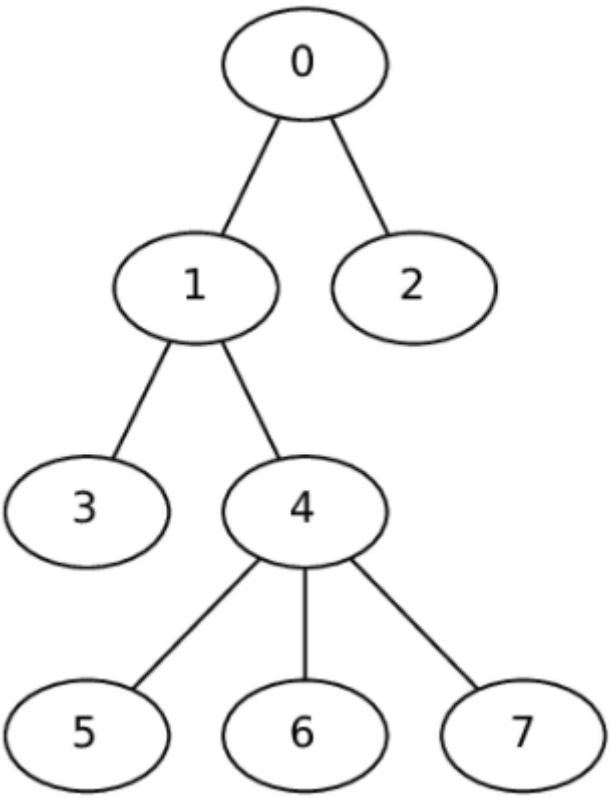


Figure 1.2: Le graphe

La représentation par matrice d'adjacence de ce graphe:

	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	1	0	0	0
2	1	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	1	0	0	0	1	1	1
5	0	0	0	0	1	0	0	0
6	0	0	0	0	1	0	0	0
7	0	0	0	0	1	0	0	0

Figure 1.3: La matrice d'adjacence

Liste d'adjacence

Définition 1.2.8. *La liste d'adjacence généralise la structure liste d'arête dans le sens suivant:*

- *On a une nouvelle séquence, la séquence d'incidence et chaque nœud S a maintenant un pointeur vers les arêtes incidentes.*
- *Chaque nœud A a maintenant des pointeurs vers les positions de ses extrémités dans la séquence d'adjacence.*

Exemple 1.2.3.

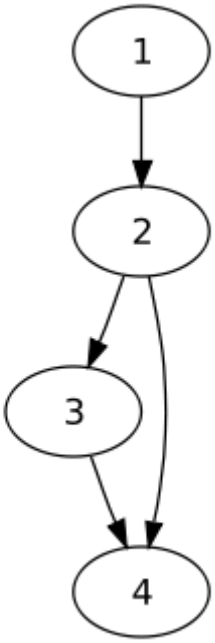


Figure 1.4: Le graphe

la représentation par listes d'adjacence du cet graphe :

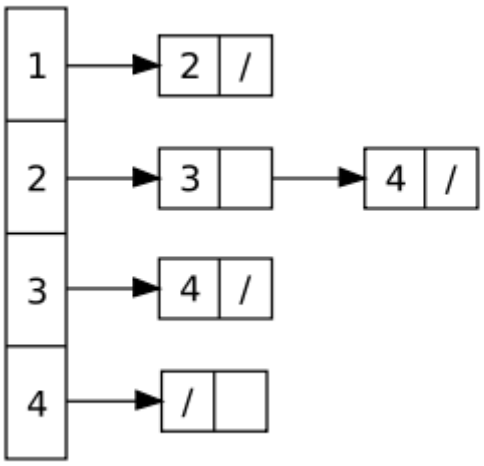


Figure 1.5: Les listes d'adjacence

1.2.7 Les types de graphe

Graphe simple

Définition 1.2.9. *Un graphe est dit simple si deux sommets distincts sont joints par au plus une arête et s'il est sans boucle.*

Sous-graphe et graphe partiel

Définition 1.2.10. *Pour un sous-ensemble de sommets A inclus dans V, le sous-graphe de G induit par A est le graphe $G' = (A, E(A))$ dont l'ensemble des sommets est A et l'ensemble des arcs $E(A)$ est formé de tous les arcs de G ayant leurs deux extrémités dans A. Autrement dit, on obtient G' en enlevant un ou plusieurs sommets du graphe G, ainsi que tous les arcs incidents à ces sommets.*

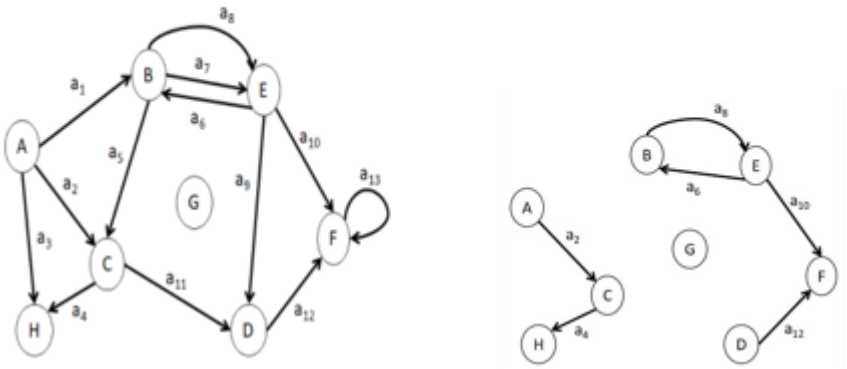


Figure 1.6: Un exemple de graphe (à gauche) et de sous-graphe (à droite)

Définition 1.2.11. *Soit $G = (V, E)$ un graphe, le graphe $G' = (V, E')$ est un graphe partiel de G, si E' est inclus dans E. Autrement dit, on obtient G' en enlevant un ou plusieurs arcs au graphe G.*

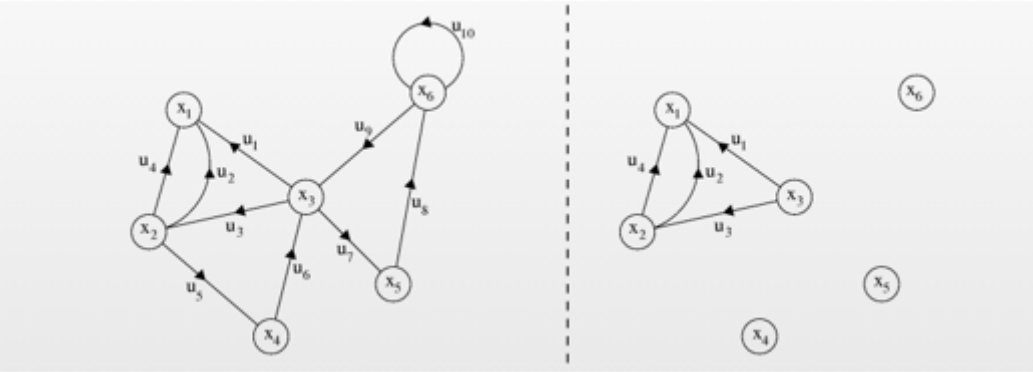


Figure 1.7: Le graphe G (à gauche) et un graphe partiel de G (à droite)

Graphe complet et clique

Définition 1.2.12. *Un graphe complet est un graphe dont les sommets sont tous reliés deux à deux par un arc.Dans un graphe G, on nomme clique un sous-graphe complet de G, c'est-à-dire une partie de l'ensemble des sommets de G telle que le graphe induit par G sur cette partie soit un graphe complet.*
Un des problèmes centraux de la théorie des graphes consiste à chercher la clique de taille maximale dans un graphe.
Un graphe complet à n sommets contient $\frac{n(n-1)}{2}$ *arcs. On note* K_n *le graphe complet d'ordre n, c'est-à-dire contenant n sommets.*

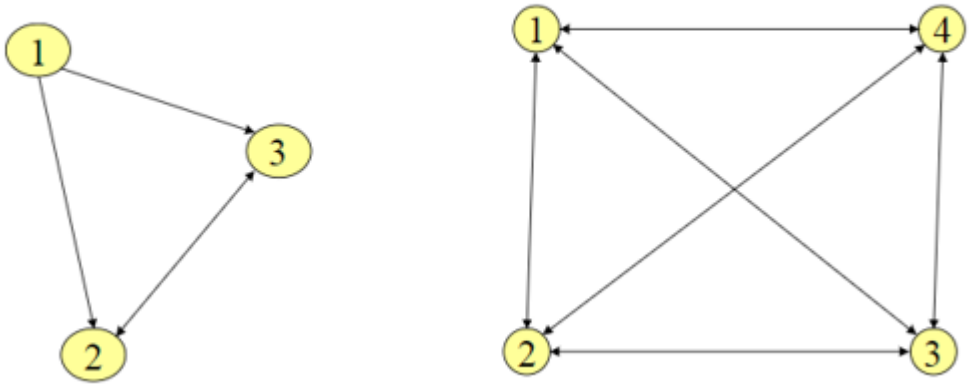


Figure 1.8: Le graphe G (à gauche) et un graphe partiel de G (à droite)

Remarque 1.2.2. *Pour un graphe d'ordre n, il existe deux cas extrêmes pour l'ensemble de ses arcs : soit le graphe n'a aucun arc, soit tous les arcs possibles pouvant relier les sommets 2 à 2 sont présents (graphe complet). L'ordre de la plus grande clique de G est noté $\omega(G)$. Dans le graphe présenté par la figure 3.5, il y a deux cliques d'ordre 3 définis par les ensembles de sommets $\{d, g, h\}$ et $\{d, e, h\}$.*

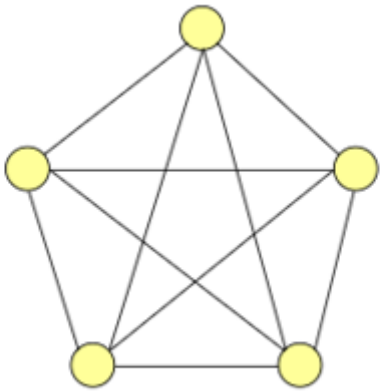


Figure 1.9: Exemples de graphes cliques

Graphe biparti

Si ses sommets peuvent être divisés en deux ensembles X et Y, de sorte que chaque arc du graphe relie un sommet dans X à un sommet dans Y.
Dans l'exemple ci-dessous, un graphe $G = (V, E)$ est décrit par l'ensemble de ses sommets $V = \{1, 2, 3, 4, 5\}$ et l'ensemble de ses arcs $E = \{(1, 2), (1, 4), (2, 3), (2, 5), (3, 4), (4, 5)\}$, on a bien deux parties $X = \{1, 3, 5\}$ et $Y = \{2, 4\}$, ou vice versa.

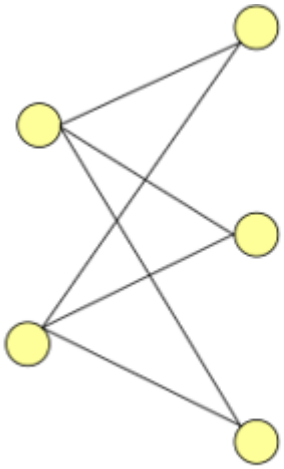
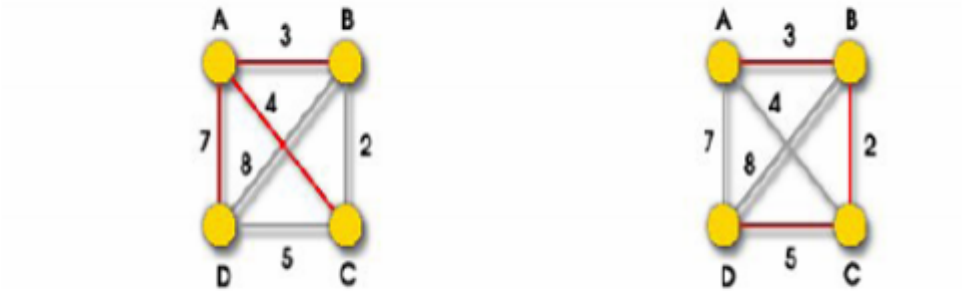


Figure 1.10: Un graphe biparti

Graphe et arbres

Définition 1.2.13. *Les arbres sont un cas particulier de graphes. Étant donné un graphe non orienté et connexe, un arbre couvrant ce graphe est un sous-ensemble qui est un arbre et qui connecte tous les sommets ensemble. Un graphe peut comporter plusieurs arbres couvrants différents. Lorsqu'on peut associer un poids à chaque arc, le poids de l'arbre couvrant correspond à la somme des poids des arcs de l'arbre couvrant. Un arbre couvrant de poids minimal (ACM) est un arbre couvrant dont le poids est inférieur ou égal à celui de tous les autres arbres couvrants du graphe. Un arbre couvrant de poids minimum est en général différent de l'arbre des plus courts chemins ($SPT´ShortestPathsTree˘$) construit, par exemple, avec l'algorithme de parcours en largeur (ou BFS, pour Breadth First Search[8]) ou avec l'algorithme de Dijsktra [Dijkstra,*1959*]. Un arbre des plus courts chemins est bien un arbre couvrant, mais il minimise la distance de la racine à chaque sommet et non la somme des poids associés aux arcs .*



Un arbre des plus courts chemins (SPT) de racine *A*, sa hauteur est *7*, son poids est *14*

Un arbre couvrant minimal (ACM) sa hauteur est *10*, son poids est *10*

Figure 1.11: L'arbre courant minimum (ACM) et le Shortest Path Tree (SPT)

Graphe connexe

Définition 1.2.14. *Un graphe est dit connexe s'il existe pour chaque paire de sommets sont connectées par une chaîne . Autrement dit, on peut toujours passer d'un sommet à un autre soit directement soit en passant par un ou plusieurs autres sommets. [2]*

Graphe planaire

Définition 1.2.15. *Dans la théorie des graphes, un graphe planaire est un graphe quelconque qui a la particularité de pouvoir se représenter sur un plan sans qu'aucun arc n'en croise un autre.*

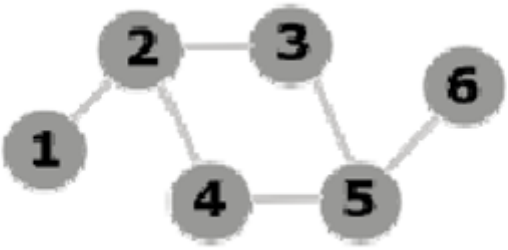


Figure 1.12: Un graphe planaire

1.2.8 Notions de base sur l'appariement de graphes

La recherche d'appariements dans un graphe vise à trouver des couples de sommets compatibles dans des graphes. Un appariement est un ensemble d'arêtes indépendantes E_i dans un graphe $G = (V, E)$. L'indépendance d'arêtes est définie par l'absence de sommets incidents communs entre elles, il s'agit d'un appariement de $V_0 \subseteq V$.

Si tous les sommets de V_0 sont couverts par E_i, c'est-à-dire s'il existe une arête dans E_i qui relie chaque sommet dans V_i à un autre [3]. L'appariement de graphes consiste donc à rechercher une correspondance entre les nœuds et les arêtes des graphes, tout en assurant la satisfaction de certaines contraintes. Généralement, les méthodes d'appariement sont divisées en deux grandes catégories : la première contient les méthodes d'appariements exacts qui nécessitent une correspondance stricte entre des graphes ou au moins entre des sous-graphes. La deuxième catégorie définit les méthodes d'appariements inexacts, où une mise en correspondance peut se produire entre deux graphes même s'ils ont des structures différentes.

1.3 Conclusion

Les graphes sont devenus, au fil du temps, un domaine majeur de recherche en informatique. Les graphes constituent donc une méthode qui permet de modéliser une grande variété de problèmes concrets. La théorie des graphes permet de générer des circuits, des réseaux (routiers, de communication,. . .).

Comme initiation à notre étude, nous avons commencé, dans le cadre ce chapitre, par présenter quelques notions de base de la théorie des graphes qui a été introduite à la base pour exprimer des problèmes combinatoires et qui a été utilisé par la suite pour résoudre efficacement des problèmes réels.

Chapitre 2

Méthode d'appariement de graphes

2.1 Introduction

En effet, la représentation sous forme de graphe est utilisée dans différentes applications d'analyse d'images. Dans ces applications, le graphe est utilisé pour représenter les objets et les relations spatiales entre objets. Cette représentation structurelle a prouvé sa flexibilité dans plusieurs domaines [4].

Nous présentons dans cette chapitre des travaux sur la représentation sous forme de graphe, ensuite nous détaillerons quelques travaux sur les mesures de similarité entre les graphes.

2.2 Appariement exact de graphes

Dans cette section, nous présentons différentes techniques proposées dans la littérature pour comparer deux graphes G et G_0 de manière exacte. Une première forme de comparaison exacte est de déterminer si les deux graphes sont isomorphes. Plus précisément, on cherche à déterminer s'il existe un morphisme f entre les deux graphes tel que, pour toute chaîne $v1, v2$ dans G, il existe une unique image v'_1 e'_1 v'_2 **dans** G_0 **via** f**.**

Une forme moins contrainte de comparaison consiste à rechercher les isomorphismes entre les sous-graphes des deux graphes. D'autres encore moins contraintes vont en relâcher certaines, comme la nécessité de mettre en correspondance chaque nœud d'un graphe avec un et un seul nœud de l'autre graphe.

Ces différents problèmes sont presque tous NP-difficiles (pour certains cas cela reste à déterminer), et ont par conséquent des temps de résolution exponentiels de manière générale. Il reste cependant possible de réduire cette complexité, par exemple en ne considérant que certaines catégories de graphes. Or en se limitant à des graphes dont le nombre de sommets est faible, les algorithmes peuvent être lancés pour un temps raisonnable compte tenu de la puissance des machines actuelles.

2.2.1 Arbre de recherche

La plupart des algorithmes de comparaison exacte sont basés sur une représentation par arbre de recherche. L'idée est de représenter chaque mise en correspondance sous la forme d'une chaîne dans un arbre de recherche. Pour ce faire, on représente chaque mise en correspondance entre deux sommets sous la forme d'un nœud dans l'arbre de recherche, et chaque mise en correspondance entre deux arêtes sous la forme d'un lien entre deux nœuds de l'arbre de recherche. Cela permet de diviser les calculs de mise en correspondance dans le but d'éviter de refaire plusieurs fois le même calcul. Cela permet aussi d'éviter certains calculs en élaguant certaines branches de l'arbre. Notons qu'une des conditions pour obtenir une solution exacte est que toutes les solutions soient atteignables en explorant l'arbre de recherche. Les algorithmes sont généralement basés sur des recherches en profondeur d'abord ou sur le branch-and-bound.

L'une des premières méthodes qui a rencontré un franc succès dans la littérature est celle d'Ullman, cette méthode peut résoudre les problèmes d'isomorphisme et d'isomorphisme de sous-graphe.
Des améliorations ont été proposées, comme [5], mais avec un coût en mémoire qui limite leur usage aux petits graphes.
Des propositions plus récentes utilisent des heuristiques plus efficaces [6], ainsi qu'une utilisation mémoire linéaire avec le nombre de sommets dans le graphe, ce qui permet de traiter des graphes de plus grande taille [7].

2.2.2 Algorithme d'Ullmann

Proposé par Ullmann en 1976. Cet algorithme fonctionne sur des graphes uniques non typés à bords dirigés ou non dirigés [8], il est considéré comme l'un des algorithmes les plus rapides pour résoudre le problème d'isomorphisme de (sous-graphe) [9]. Cette méthode est basée sur le backtracking, c'est une procédure de raffinement.
L'algorithme est conçu à la fois pour l'isomorphisme des graphes et de sous-graphes. La figure 2.1 résume les étapes de l'algorithme Ullmann.

Algorithme d'Ullmann (G_1,G_2)

Entrée : deux graphes attribués $G1 = (V, E, L_V, L_E)$, $G2 = (V', E', L'_V, L'_E)$.

Sortie : $f[1 \ldots |V|]$ vecteur représentant un isomorphisme de sous-graphes du G_1 à G_2.

/* $f[i] = j$ montre que le sommet $v_i \in V$ a été associé avec le sommet $w_j \in V'$ */

Soit P = [1… |V| ; 1…|V'|] une matrice de compatibilité entre le sommet du G_1 et G_2.

0. **Début**

1. **Pour** $i = 1$ à $|V|$

2. **Pour** $j = 1$ à $|V'|$

3. **Si** $L_V(v_i) = L'_V(w_j)$ **alors** $P[i, j] := 1$;

4. **Sinon** $P[i, j] := 0$;

5. **Pour** $i = 1$ à $|V|$

6. $f[i] := 0$;

7. **Retourner** Backtracking (P, 1, f)

8. **Fin.**

Figure 2.1: Algorithme d'Ullmann

2.2.3 Algorithme Vf2

Cet algorithme est une amélioration par rapport à l'algorithme d'Ullmann.

La première étape de l'algorithme VF2 est la même que l'étape une de l'algorithme d'Ullmann. la deuxième étape ne s'applique pas de la même manière que dans l'algorithme d'Ullmann.
L'algorithme VF2 est donné dans la figure 2.2. Etant donné deux graphes $G_1(N_1, B_1)$ et $G_2(N_2, B_2)$, la mise en correspondance des graphes est décrite par la notion de « State Space Representation ». L'objectif est de trouver un appariement $M \subset N_1 \times N_2$ (un ensemble de couples $(n, m), n \in G_1$ et $m \in G_2$) tel que M soit bijectif et préserve la structure des arcs entre les deux graphes. Un état s du processus représente une solution partielle de la correspondance entre deux graphes. M est l'ensemble des solutions partielles, M(s) est un sous-ensemble de M représentant la solution partielle courante à l'état s correspondant aux sous-graphes $G_1(s)$ et $G_2(s)$ de G_1 et G_2 respectivement.

Algorithme VF2(s)

Entrée : un état intermédiaire s ou l'état initial s0 tel que M (s0) = NULL
Sortie: Les appariements entre les deux graphes
0. début
1. si M(s) comprend tous les sommets de G2 alors renvoyer M(s)
2. sinon
3. Calculer P(s) ensemble de couples candidats pour l'inclusion dans M(s)
4. pour chaque (n, m) ∈ P(s)
5. si F (s, n, m) alors
6. Calculer l'état s' (obtenu en ajoutant le couple (n, m) à M(s))
7. Appeler VF2 (s')
8. fin si
9. fin pour
10. Stocker la structure de données
11. fin si
12. fin

Figure 2.2: L'algorithme VF2 de Cordella [24]

2.2.4 Algorithme Nauty

L'Algorithme Nauty est probablement l'algorithme pour les graphes isomorphes le plus intéressant qui n'est pas basé sur l'arbre de recherche. Cet algorithme de McKay [10] est considéré par plusieurs auteurs comme l'algorithme d'isomorphisme de graphes le plus rapide, il consiste à ordonner les nœuds de chaque graphe en se basant sur un étiquetage canonique. L'algorithme calcule, pour chaque nœud u d'un graphe, une étiquette unique en se basant sur un ensemble de caractéristiques décrivant les relations entre u et les autres nœuds du graphe. Ensuite, les étiquettes sont utilisées pour l'ordonnancement des nœuds de chaque graphe. Enfin, l'isomorphisme entre deux graphes est calculé en vérifiant l'égalité de leurs représentations canoniques, il est basé sur la théorie des groupes, il est aussi très intéressant pour comparer un nouveau graphe avec un grand nombre de graphes dont les représentations canoniques ont été pré-calculées. Cette vérification prend moins de temps que prenne la construction de l'étiquetage qui nécessite un temps qui est dans la plupart des cas exponentiel.

2.3 Appariement inexact de graphes

Pour avoir un isomorphisme entre deux graphes donnés, cela nécessite la préservation de la structure. Cette contrainte ne rend pas l'appariement exact de graphes applicable à tous les applications mais à un nombre restreint d'applications. De plus, l'inconvénient majeur des méthodes d'appariement exact des graphes est leur grande complexité de calcul.

Dans la littérature, ils existent deux catégories d'algorithmes d'appariement inexact, la première englobe les algorithmes qui estiment le coût global d'appariement et restitue les graphes dont le coût est minimal, cela implique que si la solution exacte existe, elle sera retrouvée par de tels algorithmes. Ces algorithmes sont généralement appelés algorithmes optimaux.

2.3.1 Les techniques basées sur l'arbre de recherche

Les arbres de recherche peuvent aussi être utilisés pour la comparaison inexacte de graphes. Dans ce cas, la recherche est dirigée par le coût de la mise en correspondance obtenue jusqu'à présent et par une heuristique qui estime le coût des futures mises en correspondance. Cette information est utilisée pour élaguer l'arbre mais aussi pour déterminer l'ordre de traitement des nœuds de l'arbre.

Des méthodes pour paralléliser les calculs on aussi été proposées dans ce domaine, comme [11], qui propose une version parallélisée du branch-and-bound pour calculer une distance de graphe.

2.3.2 Les techniques basées sur la théorie spectrale

Le premier qui a abordé l'appariement spectral est Umeyama [12] en 1988. Il basait sur la théorie spectrale des graphes,il cherchait à analyser les valeurs et les vecteurs propres des matrices d'adjacence des deux graphes à apparier, du fait que les valeurs propres sont indépendantes des permutations de sommets. Si deux graphes sont isomorphes, leurs matrices d'adjacence ont les mêmes valeurs propres et les mêmes vecteurs propres (c'est-à-dire la même décomposition), mais l'inverse n'est pas nécessairement vrai. Les méthodes spectrales pour l'appariement de graphes ont reçu une attention considérable du fait que le calcul des valeurs propres et des vecteurs propres est un problème d'une complexité temporelle polynomiale. Dans le chapitre qui va suivre nous allons détailler cet algorithme et nous allons l'implémenter.

2.3.3 Les techniques basées sur l'apprentissage

Les algorithmes d'apprentissage sont aussi appliqués dans l'appariement de graphes pour déterminer les paramètres de la fonction à optimiser. Dans un cadre applicatif il s'agit en particulier des coûts et des poids associés à la correction d'erreurs aux niveaux des attributs et de la structure du graphe [13].

À côté de ces approches, il existe de nombreuses autres techniques. On peut citer les méthodes basées sur les réseaux de neurones artificiels Suganthan et al [14]. La plus part de ces méthodes utilisent un réseau de neurone de type Hopfield où l'appariement de graphes est formulé sous forme de minimisation d'énergie. Une autre approche présentée dans [15] par Mauro et al., qui a proposé l'utilisation d'un réseau de neurones récurrents pour calculer la distance entre les graphes orientés acycliques. Cette technique se base sur la projection des graphes dans un espace vectoriel et sur l'utilisation de la distance euclidienne [16]. Il existe des approches génétiques pour la résolution du problème d'isomorphisme de graphes. Leurs différences techniques se trouvent principalement dans le choix des opérateurs. Deux types de codage différents sont proposés : le codage binaire et la représentation de chaînes de nombres entiers qui correspond au codage de permutations. Ils proposent le croisement et la mutation par couleur. L'idée fondamentale est d'employer une classification donnée des nœuds puis réduire l'espace de recherche en

ne permettant que les alignements dont tous les nœuds correspondants possèdent la même classe. La classification doit être telle que la distance la plus grande à l'intérieur de n'importe quelle classe soit inférieure à la distance la moins élevée entre deux éléments de classes différentes [13].

2.4 Mesure de similarité des graphes

La structure des graphes est caractérisée principalement par la flexibilité, l'universalité et l'utilisation dans des domaines d'applications variés. Ce fait a conduit au développement de plusieurs mesures de similarité pour les graphes qui sont optimisées pour différentes applications. La plupart de ces mesures ont une caractéristique commune qui consiste en quelque sorte à utiliser des opérations d'édition (edit operations). Le principe est de définir l'effort nécessaire pour rendre les graphes identiques. Dans la suite, nous détaillerons les mesures les plus connues dans la littérature.

2.4.1 Distance d'éditions

Définition 2.4.1. *La distance d'édition pour les graphes est l'extension de la distance d'édition pour les chaînes des caractères (ou distance de Levenshtein) [17]. C'est une mesure très commune pour évaluer la similarité entre les graphes. La distance d'édition entre deux graphes est le nombre minimum des opérations d'édition qui sont nécessaires pour transformer un graphe en un autre.*

Les principales opérations d'édition sont, la suppression ou l'insertion des sommets ou arcs, et le changement des attributs des sommets ou des arcs. Pour calculer la distance entre deux graphes Robles-Kelly [18] propose une méthode de conversion d'un graphe à une séquence de chaînes de caractères afin d'appliquer des techniques de mesure de la distance d'édition de chaîne de caractères. La problématique principale de la distance d'édition est la détermination du coût minimal des opérations d'éditions effectuées. Les solutions proposées à cette problématique sont dans la majorité probabiliste [19].

2.4.2 Mesure de similarité-distance

Une autre mesure de similarité pour les graphes est proposée par Chartrand, Kubicki et Schultz dans [20]. Cette mesure est basée sur des fonctions entre les ensembles des sommets des graphes. Elle est définie pour les graphes connexes de même ordre. Soient deux graphes connexes $G_1 = (V_1, E_1)$ et $G_2 = (V_2, E_2)$ ayant le même ordre n, et l'application linéaire $\emptyset : V1 \rightarrow V2$. La distance entre G_1 et G_2 est définie par :

$$D\emptyset(G_1, G_2) =\mid lp(U, V) - lp(\emptyset U, \emptyset V) \mid U; V \quad \texttt{Où lp(U,V)}$$

C'est la longueur du plus court chemin entre u et v dans G_1, et la somme est prise pour toutes les paires des sommets u et v distincts dans G_1. La mesure de similarité entre deux graphes connexes G_1 et G_2 avec le même ordre est :

$$D(G_1, G_2) = min(\emptyset(G_1, G_2))$$

Le minimum est pris parmi toutes les applications $\emptyset$ possibles de V_1 à V_2.

Les auteurs Chartrand et al. montrent que leur mesure de similarité respectent les propriétés des espaces métriques, mais ils ne proposent aucun algorithme pour calculer cette mesure.

2.5 Conclusion

Dans cette section nous avons passé en revue des différentes méthodes d'appariement de graphes. Différentes mesures de similarité ont été proposées dont nous avons cité une liste non-exhaustive. Ces travaux ont été choisis puisqu'ils gardent la structure générale des graphes. Mais les mesures de similarité sont très générales et très complexes. Il y a d'autres travaux essayant de faire l'appariement en transformant un graphe en arbre [21], [22], [23]. Ces travaux ne sont pas présentés dans notre état de l'art puisque la transformation d'un graphe en arbre implique une grande perte d'information.

Chapitre 3

L'algorithme VF2 et VF3

3.1 Introduction

Plusieurs applications basées sur des graphes nécessitent de détecter et de localiser les occurrences d'un graphe de motif dans un graphe cible plus grand. L'isomorphisme sous-graphe est une formalisation largement adoptée de ce problème, alors que l'isomorphisme des sous-graphes est NP-Complet. Dans le cas général, ils existent des algorithmes qui peuvent le résoudre en un temps raisonnable sur les graphes moyens, nous avons introduit un de ces algorithmes $VF2$, spécialement conçu pour les grands graphiques rencontrés dans les applications bio-informatiques. À son tour, $VF2$ s'est avéré beaucoup plus rapide que son prédécesseur et parmi les algorithmes les plus rapides sur les graphiques bio-informatiques. Dans ce mémoire, nous présentons une nouvelle évolution, nommée $VF3$, qui ajoute de nouvelles améliorations spécifiquement destinées à améliorer les performances sur des graphiques à la fois volumineux et denses, qui sont actuellement le cas le plus problématique pour l'état de l'art et algorithmes. Dans ce chapitre, nous décrirons $VF3$, c'est un nouvel algorithme capable de traiter différents problèmes d'appariement de graphes exacts en raison de sa structure générale. Malgré cela, dans ce mémoire, nous avons concentré l'attention sur l'isomorphisme du sous-graphe, nous discutons donc en détail la façon dont l'algorithme traite ce problème. Nous discuterons d'abord la structure de $VF2$ héritée par $VF3$, puis nous discuterons de ce qui a été amélioré dans $VF3$. Enfin, nous comparerons $VF3$ avec $VF2$ et d'autres algorithmes.

3.2 L'algorithme VF2

Définition 3.2.1. *$VF2$ [24] est un algorithme très flexible capable de traiter différents problèmes de correspondance exacte de graphes de plusieurs types. Il est basé sur une représentation de l'espace d'état où chaque état est une partie de la correspondance entre les deux graphes donnés. Tous les états dont la cartographie est complète, c'est-à-dire qu'elle ne peut pas être étendue davantage en ajoutant de nouveaux couples de nœuds, sont des solutions possibles. Mais, seuls ceux satisfaisant les contraintes imposées par la spéciéité du problème sont considérés comme les états objectifs. Par exemple, si l'algorithme recherche un isomorphisme de sous-graphe, considéré comme un état objectif, un état correspondant à une application qui implique tous les nœuds du plus petit graphe et satisfait les contraintes préservant la structure.*

L'espace d'état est naturellement représenté sous forme de graphe, il peut donc être exploré de différentes manières, comme d'autres algorithmes basés sur une recherche arborescente, $VF2$ adopte une stratégie de profondeur d'abord avec retour en arrière, pour son efficacité en termes d'espace et de temps, mais la vraie innovation de $VF2$ a été l'introduction de règles de faisabilité pour élaguer, en chemins de recherche avancés et infructueux combinés à une mémoire efficace représentant de l'espace d'état. Ces deux éléments ont permis d'avoir une complexité spatiale linéaire par rapport à la taille du plus petit graphe et une complexité temporelle qui est quadratique en le cas moyen. Il convient de souligner qu'un article récent de N. Dahm et H. Bunke [**?**] ont reconfirmé l'efficacité des règles de faisabilité en termes de réduction de l'espace de recherche et de complexité. Néanmoins, la nature exponentielle du problème d'isomorphisme du sous-graphe et la grande variété des domaines d'application soulèvent le besoin d'un algorithme générique par rapport aux contextes spécifiques, mais facile à spécialiser, en utilisant si possible des heuristiques simples afin de réduire l'espace de recherche exploré et par conséquent le temps pour trouver les solutions.

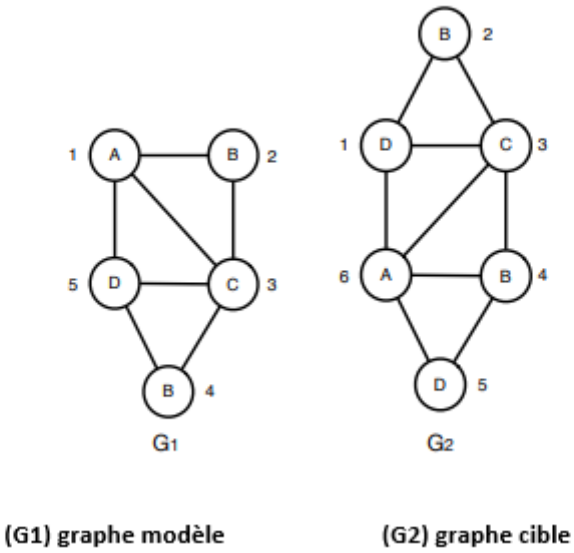


Figure 3.1: Graphes modèle (à gauche) et cible (à droite) utilisés comme dans la suite pour simuler le fonctionnement de VF2 et VF3

3.2.1 Représentation du problème

Comme présenté ci-dessus (figure 3.1), le processus de recherche d'une correspondance exacte entre deux graphes peut être résolu au moyen d'une représentation dans l'espace d'état (SSR : Space States Representation). Plus formellement, étant donné les graphes $G1 = (V1, E1)$ et $G2 = (V2, E2)$ présente une application partielle $M(s)$ entre $V1$ et $V2$, qui ne fait intervenir qu'un sous-ensemble de tous les couples de nœuds dans l'application cible M.
Ainsi, chaque application partielle $M(s) \equiv M1(s) \times M2(s)$ est composée d'un ensemble de couples de nœuds ordonnés (U, V) où $U \in M1(s) \subseteq V1$ et $V \in M2(s) \subseteq V2$.

Rappelons le fait qu'un sous-graphe peut être extrait d'un graphe en considérant un sous-ensemble de ses nœuds, il est trivial d'imaginer que les sous-ensembles $M1(s)$ et $M2(s)$ induisent deux sous-graphes

$$G1(s) = (M1(s), B1(s)) \; et \; G2(s) = (M2(s), B2(s))$$

respectivement de $G1$ et $G2$, où intuitivement, les ensembles $B1(s)$ et $B2(s)$ ne contiennent que les arêtes de $G1(s)$ et $G2(s)$ reliant les nœuds dans $M1(s)$ et $M2(s)$. Par conséquent, si ces sous-graphes satisfont les contraintes de l'appariement souhaité, alors l'état s sera cohérent. Par exemple, si l'algorithme recherche

un isomorphisme de graphe alors, s sera cohérent si $G1(s)$ et $G2(s)$ sont isomorphes. Un exemple des ensembles décrits ci-dessus est illustré dans la figure 3.2 et la figure 3.3.

Explorer l'espace

Selon la représentation décrite dans la section précédente, le processus d'appariement est une recherche à l'intérieur du SSR où l'algorithme commence à partir d'un état s_0, qui représente un mapping vide $M(s_0) = \emptyset$ recherche un ou plusieurs états cibles sg, où $M(sg) \equiv M$.

Un nouvel état s' est généré à partir d'un état parent s en ajoutant un nouveau couple ordonné (U, V), où $U, V \notin (s)$.

Le passage d'état de s à s' correspond à l'addition du nœud U à $G1(s)$ et du nœud V à $G2(s)$.

Il est facile de comprendre que si nous générons tous les états possibles, par une exploration exhaustive, nous trouverons tous les états buts, s'il en existe au moins un. Mais, du fait de la nature combinatoire du problème, le coût de calcul d'une exploration exhaustive est factoriel par rapport à la taille des graphes. Néanmoins, seuls les états cohérents conduiront l'algorithme vers une solution et le nombre de tels états est sensiblement plus petit que la taille totale de l'espace d'état. En effet, un état non cohérent s ne générera aucun état cohérent. Ceci est dû au fait que, si $G1(s)$ n'est pas isomorphe avec $G2(s)$ alors $G1(s^{'})$ et $G2(s^{'})$ obtenus en ajoutant U à $G1(s)$ et v à $G2(s)$ ne seront pas isomorphe. Par conséquent, il est possible de concentrer le processus de recherche sur des états cohérents uniquement.

Comme le montre l'algorithme 1, $VF2$ explore l'espace de recherche selon une stratégie de recherche en profondeur avec retour en arrière. A chaque itération, l'algorithme recherche un nouveau couple candidat (u, v) pour générer un nouvel état $s' = s \bigcup (u, v)$.
Mais il n'est pas exploré dès que le nouveau couple est ajouté. Avant cela, $VF2$ vérifie d'abord, si l'état s' est cohérent, alors, s'il a au moins un descendant cohérent En effet, si la dernière condition n'est pas vérifiée, il est sûr que l'on ne trouvera pas d'état but en l'explorant, donc s' peut être coupé. À cette fin, $VF2$ utilise un ensemble de règles d'anticipation à travers lesquelles l'algorithme est capable de déterminer si un nouveau couple est réalisable pour générer un état cohérent avant qu'il ne soit exploré. Il convient de souligner que les conditions vérifiées par ces règles sont nécessaires, mais pas suffisantes pour satisfaire les contraintes d'appa-

riement. Mais une telle stratégie permet à l'algorithme de réduire significativement le nombre d'états explorés.
Tous les principaux aspects qui ont été introduits dans cette section seront expliqués en détail dans ce qui suit.

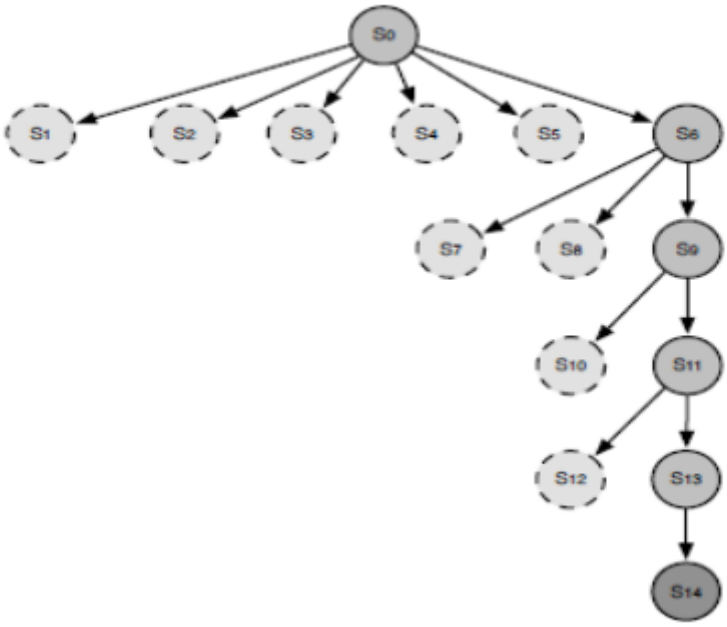


Figure 3.2: Montre tous les états trouvés par VF2

Etats	Correspondances trouvés
s_0	$M(s_0) = \emptyset$
s_1	$M(s_1) = \{(1,1)\}$
s_2	$M(s_2) = \{(1,2)\}$
s_3	$M(s_3) = \{(1,3)\}$
s_4	$M(s_4) = \{(1,4)\}$
s_5	$M(s_5) = \{(1,6)\}$
s_6	$\mathbf{M(s_6) = \{(1,6)\}}$
s_7	$M(s_7) = \{(1,6),(2,1)\}$
s_8	$M(s_8) = \{(1,6),(2,3)\}$
s_9	$\mathbf{M(s_9) = \{(1,6),(2,4)\}}$
s_{10}	$M(s_{10}) = \{(1,6),(2,4),(3,1)\}$
s_{11}	$\mathbf{M(s_{11}) = \{(1,6),(2,4),(3,3)\}}$
s_{12}	$M(s_{12}) = \{(1,6),(2,4),(3,3),(4,1)\}$
s_{13}	$\mathbf{M(s_{13}) = \{(1,6),(2,4),(3,3),(4,2)\}}$
s_{14}	$\mathbf{M(s_{14}) = \{(1,6),(2,4),(3,3),(4,2),(5,1)\}}$

Figure 3.3: Montre les ensembles de base (lignes continues) et l'ensemble terminal (ligne pointillées) liés aux états

Dans les deux figures sont considérés les deux graphes de la figure 3.1, elle montre tous les états trouvés par$VF2$ lors de l'exploration. Les états représentés en pointillés sont ceux générés mais non explorés car irréalisables. En revanche, les états représentés par des traits pleins sont ceux cohérents. $S14$ est un état objectif car il est à la fois complet et cohérent. Notez que la numérotation dépend de l'ordre dans lequel les états sont générés par l'algorithme. Enfin, dans le tableau est montré l'ensemble de base pour chaque état, les mappages cohérents sont mis en évidence en utilisant un texte en gras.

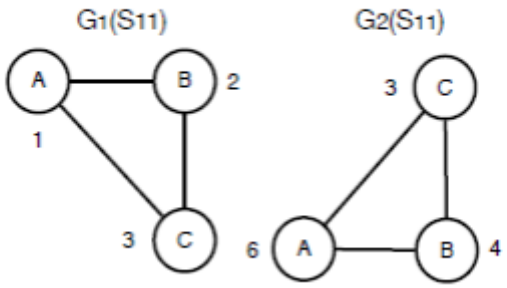


Figure 3.4: Les deux sous-graphes isomorphes induits par les nœuds qui sont à l'intérieur du mapping à l'état s11. Il est clair que s11 est consistant car il satisfait les contraintes imposées

3.2.2 Règles de faisabilité

La notion de faisabilité est un point clé de $VF2$ et toute la structure de cet algorithme est basée sur elle. En effet, l'idée de couple de nœuds réalisable est directement liée à celle de cohérence d'état. Puisque l'algorithme doit explorer uniquement les états cohérents, la fonction de transaction $s' = s(u, v)$, utilisée pour générer un nouvel état s'.

Partir d'un état cohérent s, doit garantir que l'addition de la paire (u, v) conduira l'algorithme vers un nouvel état cohérent. Ainsi, avant de générer un nouvel état, $VF2$ analyse la faisabilité d'un couple candidat (u, v) en utilisant une fonction de faisabilité $F(s, u, v)$ qui prend en compte à la fois les informations structurelles et sémantiques de chaque nœud :

$$F(s, u, v) = F_{sem}(s, u, v) \wedge F_{str}(s, u, v) \tag{3.1}$$

```
1: function Match(s, G1, G2)
 2:
 3:    if IsGoal(s) then
 4:       return True
 5:    end if
 6:
 7:    if IsDead(s) then
 8:       return False
 9:    end if
10:
11:   Set u = ϵ ∧ v = ϵ
12:   (u',v') = Get N extCandidate(s,(u,v), G1, G2)
13:   while u'≠ ϵ ∧ v' ≠ ϵ do
14:     if IsFeasible((u',v'), G1, G2) then
15:        s' = sU(u',v')
16:        if Match(s', NG1, G1G1, G2) is True then
17:           return True
18:        end if
19:     end if
20:     (u',v') = Get N extCandidate(s,(u',v'), G1, G2)
21:   end while
22:   return False
23: end function
```

Algorithm 1: Structure de la procédure d'appariement utilisée par $VF2$. Les entrées fournies à la procédure sont l'état de départ s et les deux graphes G_1, G_2. La procédure retourne vrai si la solution existe, sinon faux.

Le terme sémantique $F_{sem}(s, u, v)$ de la fonction ne dépend que des attributs des deux nœuds et sert à évaluer si l'information sémantique des deux nœuds est équivalente. Le terme structurel $F_{str}(s, u, v)$ est plus complexe car il analyse le voisinage de chaque nœud en considérant la cohérence de trois sous-ensembles diérents : les voisins qui sont déjà dans le mapping set $M(s)$ de l'état courant s, les voisins qui ne sont pas dans $M(s)$ mais qui sont connectés à des nœuds dans $M(s)$ et ceux qui ne sont pas dans $M(s)$ et ne sont pas des nœuds connectés dans $M(s)$. A cet effet, $VF2$ pour chaque état utilise deux ensembles distincts,

un pour chaque graphe : les ensembles de base, $M_1(s) \subseteq V_1(s)$ et $M_2(s) \subseteq V_1(s)$, pour stocker les nœuds déjà dans $M(s)$ et les ensembles terminaux, $T(s) \subseteq V_1(s)$ et $T_2(s) \subseteq V_1(s)$,contenant les nœuds connectés à ceux-ci sont à l'intérieur de l'ensemble central. Ensuite, le terme structurel de la fonction de faisabilité peut être évalué en considérant trois règles différentes, une pour chaque sous-ensemble défini ci-dessus :

$$Fstr(s,u,v) = R_{core}(s,u,v) \wedge R_{term}(s,u,v) \wedge R_{new}(s,u,v) \tag{3.2}$$

$R_{core}(s,u,v)$ est la règle principale utilisée pour évaluer si l'algorithme va vers un nouvel état cohérent ou non en ajoutant le nouveau couple (u,v).
La règle vérifie si toutes les contraintes imposées par le problème d'appariement auquel l'algorithme est confronté sont respectées.
Par exemple, en cas d'isomorphisme $R_{core}(s,u,v)$ assure qu'en ajoutant le nœud u au sous-graphe $G_1(s)$ et le nœud v au sous-graphe $G_2(s), G_1(s')$ et $G_2(s')$ sont toujours isomorphes.

$$\begin{aligned} R_{core}(s,u,v) \iff & (\forall U' \in adj(G_1,U) \cap M_1(s)) \ (\exists! V' \in adj(G_2,V) \cap M_2(s)) \\ & (U',V') \in M(s) \\ & (\forall U' \in adj(G_2,V) \cap M_2(s)) \ (\exists! U' \in adj(G_1,U) \cap M_1(s)) \\ & (U',V') \in M(s) \end{aligned} \tag{3.3}$$

A la différence de Rcore, les deux autres règles définissent des conditions qui sont nécessaires mais pas suffisantes pour déterminer si un état est cohérent. Néanmoins, elles sont utiles pour réduire le nombre d'états explorés en anticipant. En effet, $R_{term}(s,u,v)$ et $R_{new}(s,u,v)$ sont utilisés pour rechercher la cohérence respective un et deux pas après l'état courant dans le but de déterminer si parmi tous les descendants de s il y aura ou pas du moins un état objectif.
Donc, s'ils sont faux, nous sommes sûrs que l'algorithme ne trouvera aucun état but en explorant s. Une considération importante concernant les règles d'anticipation est liée à la complexité de calcul.
En effet, plus l'algorithme regarde loin, plus le coût de calcul de la règle augmente. Par conséquent, afin de limiter un tel coût, les deux règles d'anticipation ne considèrent que la cardinalité des ensembles utilisés pour analyser la cohérence.

$$R_{term}(s,u,v) \iff | \, adj(G_1,U) \cap T_1(s) \, | \leq | \, adj(G_2,V) \cap T_2(s) \, | \tag{3.4}$$

$$R_{new}(s,u,v) \iff | \, adj(G_1,U) \cap \nu_1(s) \, | \leq | \, adj(G_2,V) \cap \nu_2(s) \, | \tag{3.5}$$

Dans l'équation (3.5), deux nouveaux ensembles ont été introduits, $\nu_1(s) \subseteq V_1(s)$ et $\nu_2(s) \subseteq V_2(s)$, pour représenter les nœuds qui ne sont ni dans $M(s)$ ni dans les ensembles terminaux.

Extension aux graphes orientés

Les règles de faisabilité peuvent être facilement étendues pour les graphes orientés en considérant différents ensembles pour les directions des arêtes, respectivement $Prec(G_1, U)$, $Prec(G_2, V)$ pour les arêtes entrantes et $Succ(G_1, U)$, $Succ(G_2, V)$ pour les bords sortants.

Les ensembles terminaux $T_1(s)$ et $T_2(s)$ seront divisés chacun en deux sous-ensembles $T_1^{in}(s)$, $T_1^{out}(s)$, $T_2^{in}(s)$ et $T_2^{out}(s)$. Dans ce cas, la règle $R_{core}(s, u, v)$ est composée de deux sous-règles $R_{succ}(s, u, v)$ et $R_{pred}(s, u, v)$ utilisées pour vérifier les cohérences sur les deux directions. De même, les règles d'anticipation devraient être adaptées pour considérer séparément les deux types d'arêtes.

$$R_{core}(s, u, v) \Longleftrightarrow R_{succ}(s, u, v) \wedge R_{pred}(s, u, v) \tag{3.6}$$

$$R_{succ}(s, u, v) \Longleftrightarrow \forall U' \in Succ(G_1, U) \cap M_1(s) \exists ! V' \in Succ(G_2, V) \cap M_2(s) : (U', V') \in M(s) \tag{3.7}$$

$$\bigwedge \forall V^{'} \in Succ(G_1, V) \cap M_1(s)$$
$$\exists\ !U^{'} \in Succ(G_2, U) \cap M_2(s) : (U^{'}, V^{'}) \in M(s)$$

$$R_{Perd}(s, u, v) \Longleftrightarrow \forall U' \in Perd(G_1, U) \cap M_1(s) \exists ! V' \in Perd(G_2, V) \cap M_2(s) : (U', V') \in M(s) \tag{3.8}$$

$$\bigwedge \forall V^{'} \in Perd(G_1, V) \cap M_1(s)$$
$$\exists\ !U^{'} \in Perd(G_2, U) \cap M_2(s) : (U^{'}, V^{'}) \in M(s)$$

$$\begin{aligned} R_{term}(s, u, v) \quad \Longleftrightarrow \quad & \mid Pred(G_1, U) \cap T_1(s) \mid \leq \mid \mid Pred(G_2, V) \cap T_2(s) \mid \\ & \wedge \mid Succ(G_1, U) \cap T_1(s) \mid \leq \mid Succ(G_2, V) \cap T_2(s) \mid \end{aligned} \tag{3.9}$$

$$\begin{aligned} R_{new}(s, u, v) \quad \Longleftrightarrow \quad & \mid Pred(G_1, U) \cap \nu_1(s) \mid \leq \mid \mid Pred(G_2, V) \cap \nu_2(s) \mid \\ & \wedge \mid Succ(G_1, U) \cap \nu_1(s) \mid \leq \mid Succ(G_2, V) \cap \nu_2(s) \mid \end{aligned} \tag{3.10}$$

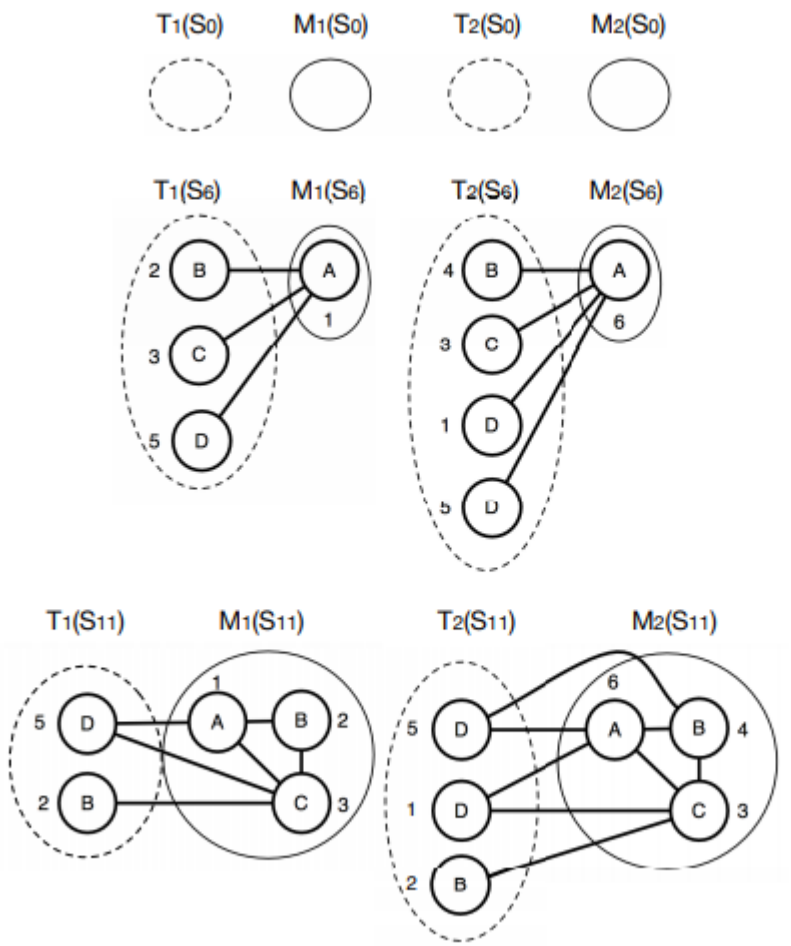


Figure 3.5: Les Core sets (traits pleins) et le Terminal set (traits pointillés) lié à l'état S0, S6 et S11 en VF2

3.3 L'algorithme VF3

Après plus de dix ans, les problèmes auxquels l'isomorphisme de sous-graphe est appliqué ont changé et les caractéristiques structurelles et sémantiques des graphes impliqués également. Dans plusieurs nouvelles applications, $VF2$ n'est plus compétitif par rapport à l'état de l'art, comme cela a été montré dans certains articles [25], [26], [27], [28]. Par conséquent, de nouvelles stratégies sont nécessaires pour être en mesure de rivaliser avec les défis qui se sont récemment posés. $VF3$ est un nouvel algorithme qui a été conçu dans le but d'améliorer $VF2$ en préservant sa structure basée sur un SSR, un DFS avec retour en arrière et un ensemble de règles de faisabilité. À cet égard, $VF3$ a introduit une étape de prétraitement de modèle basée sur de nouvelles relations d'ordre total, une nouvelle procédure pour sélectionner le couple candidat et enfin, un ensemble de règles d'anticipation basées sur les classes qui améliorent le pouvoir d'élagage de ceux d'origine.

3.3.1 Prétraitement des motifs

$VF3$ est l'algorithme le plus rapide pour résoudre la parité de sous-graphes dans des graphes larges et denses. C'est très efficace en temps et en mémoire.

3.3.2 Une nouvelle relation de commande totale

Comme décrit dans la section précédente, $VF2$ définit une relation d'ordre arbitraire sur le second graphe pour traiter le problème des cycles dans l'espace d'état. cependant $VF3$ a pour objectif supplémentaire de préparer une séquence d'exploration de nœuds pour le graphe de motifs, qui est successivement utilisée pour prétraiter le graphe de motifs et préparer à l'avance toute la structure nécessaire pour explorer le graphe.

A la différence de recheche d'infomations, la nouvelle relation d'ordre utilisée par $VF3$ ne prend pas en compte uniquement les caractéristiques structurelles du graphe modèle, mais essaie de les combiner avec les caractéristiques structurelles et sémantiques du graphe cible. En fait, l'idée de base est de donner la priorité aux nœuds ayant la plus faible probabilité de trouver un candidat réalisable et le plus grand nombre de connexions avec les nœuds déjà insérés dans la séquence. De cette façon, l'algorithme explorera d'abord les nœuds avec le plus grand nombre de contraintes. Pour cela, entrons dans les détails et définissons les concepts de base.
La procédure GenerateNodeSequence a pour but d'explorer le graphe G_1 et de

générer une séquence d'exploration de nœuds N_{G_1} en appliquant les relations d'ordre.

Il doit connaître, pour chaque nœud $U \in G_1$, la probabilité $P_{faisable}(U)$ de trouver un candidat réalisable $V \in G_2$.

Cette probabilité est obtenue en combinant $P_{lab}(L)$ et $P_{deg}(d)$ qui sont respectivement la probabilité de trouver un nœud de label L et la probabilité de trouver un nœud de degré d dans G_2. Dans le cas du sous-graphe, l'isomorphisme $P_{faisable}(U)$ peut être calculé comme défini dans l'équation (3.11).

$$P_{feasible_{subiso}}(u) = P_{lab}(lab(u)) \cdot \sum_{d' \geq deg(U)} P_{deg}(d') \tag{3.11}$$

Il est intéressant de noter que dans le cas de l'isomorphisme de graphe, la contrainte structurelle donnée par le degré est plus forte et $V \in G_2$ sera compatible avec $U \in G_1$ si $deg(U) = deg(V)$. Cette contrainte affecte la définition de $P_{faisable}(U)$, comme le montre l'équation (3.12).

$$P_{feasibleiso}(u) = P_{lab}(lab(u)) \cdot P_{deg}(deg(u)) \tag{3.12}$$

Une fois la probabilité calculée, il existe une autre information utilisée par la procédure pour prendre en compte les contraintes structurelles provenant des nœuds déjà dans NG1. Dans ce but, nous avons introduit le concept de degré de mappage de nœuds, à savoir le $degré_M$. Étant donné un nœud $U' \in G_1$, il est défini comme le nombre d'arêtes entrantes et sortantes entre U' et tous les nœuds qui sont à l'intérieur de N_{G_1}. Ainsi, à chaque étape, lorsqu'un nouveau nœud est inséré dans N_{G_1},la procédure doit mettre à jour la contrainte de tous les nœuds $U' \notin N_{G_1}$.

```
1: function GENERaAtTeENODESEQUENCE(G1,Pfeasible)
2:     ∀n ∈ G1 set degM(n) = 0
3:     Extract n0 = minn∈G1 Pfeasible(n)∧ maxn∈G1 deg(n)
4:     Add n0 in NG1
5:     Update the set degM
6:     for all n ∈ G1 ∧n ∉ NG1 do
7:       ni= maxn∈G1 degM(n)∧ minn∈G1 Pfeasible(n)∧ maxn∈G1deg(n)
8:       Add ni in SG1
9:       ∀n ∈ G1 ∧n ∉ NG1 update degM(n)
10:    end for
11:    return NG1
12: end function
```

Algorithme 2: Procédure pour générer une séquence d'exploration. Les entrées fournies à la procédure sont le graphe G_1 dont les nœuds doivent être ordonnés et les probabilités de trouver un couple réalisable $P_{faisable}$ pour chaque nœud de G_1. La sortie fournie est N_{G_1}, une séquence d'exploration de nœud sur G_1.

Il convient de souligner que la séquence d'exploration générée par la procédure représente un chemin de parcours fixe utilisé par $VF3$ pour visiter le premier graphe. Chaque position de la séquence correspond à un niveau spécifique de la recherche en profondeur d'abord. Par conséquent, tous les états au niveau i seront générés en ajoutant à ceux au niveau i_1 les couples qui partagent le même nœud du graphe de motifs. Par exemple, si nous considérons une séquence de nœuds arbitraire $N_G = 1, 2, 4, 3$, chaque état s, au deuxième niveau, est généré en ajoutant un couple qui a 2 comme premier nœud.

3.3.3 Pré Calcul des structures d'état

Avoir une séquence d'exploration fixe pour le premier graphe fournit un avantage secondaire à l'algorithme. En effet, en utilisant une telle séquence, $VF3$ est capable de pré-traiter le graphe avant de lancer l'appariement et de calculer, à chaque niveau de DFS, l'état de chaque ensemble (ensembles terminaux et ensemble central). De plus, lors du pré-traitement, l'algorithme est capable de générer un arbre de couverture utilisé successivement pour améliorer significativement la sélection d'un nouveau couple de nœuds.

L'algorithme 3 montre la procédure utilisée pour pré-traiter le premier graphe. Il explore itérativement le voisinage de chaque nœud u dans la séquence S_{G_1}. Si un voisin u' n'est pas encore dans l'ensemble terminal T_1 de G_1, il sera inséré dans T_1 et u deviendra le parent de u'.

Étant donné que l'un des objectifs d'une telle procédure est de calculer l'état de T_1 à chaque niveau de DFS, lorsqu'un nœud est inséré dans l'ensemble, la procédure prendra note du niveau auquel le nœud a été inséré. Cette information peut être facilement obtenue par la procédure rappelant la relation entre le niveau de DFS et la position d'un nœud dans la séquence S_{G_1}.
Enfin, l'arbre de couverture est obtenu sans effort en utilisant l'ensemble parent renvoyé par la procédure. De toute évidence, le premier nœud de la séquence a un parent fictif, généralement identifié comme un nœud nul.

Exemple 3.3.1. *Dans cette section, nous allons fournir un exemple simple pour clarifier comment le tri et le prétraitement sont effectués par VF3. Comme discuté ci-dessus, la première étape est la génération d'une séquence d'exploration N_{G_1}; dans ce but, il calcule les fréquences d'étiquette et de degré sur le graphe cible G_2 puis il calcule la probabilité de trouver une paire réalisable pour tous les nœuds de G_1.*

Les tableaux 3.1 et 3.2 présentent les fréquences et probabilités concernant les graphiques de la figure 3.6. Au début N_{g_1} est vide, donc le premier nœud à insérer est sélectionné uniquement sur la base de la probabilité. En se référant au tableau 3.2, le nœud 3 a la probabilité la plus faible, alors l'algorithme le choisira en premier. Ainsi,$N_{G_1} = 3$, tous les autres nœuds sont connectés à 3, ils ont donc le même deg_M[29].

```
1: function PREPROCESSFIRSTGRAPH(G1, SG1, T1)
2:    i = 0
3:    for all u ∈ SG1 do
4:      for all u' ∈ adj(u) do
5:        if u' ∉ NG1 T1 then
6:          Put u' in T1 at level i
7:           Parent(u') = u
8:        end if
9:        i = i + 1
10:     end for
11:   end for
12:   return Parent
13: end function
```

Algorithme 3: Procédure pour prétraiter le premier graphe G_1 étant donné une séquence de nœuds d'exploration S_{G_1}. Il renvoie l'ensemble des parents pour chaque nœud de la séquence.

Étiqueter	Fréquence	Degré	Fréquence
A	0.16	1	0
B	0.34	2	0.34
C	0.16	3	0.34
D	0.34	4	0.34

Table 3.1: Fréquences d'étiquettes et de degrés extraites de G_2

Nœud	Degré	Étiqueter	Total
1	0.66	0.16	0.11
2	1	0.34	0.34
3	0.34	0.16	0.05
4	1	0.34	0.34
5	0.66	0.34	0.22

Table 3.2: Probabilités pour les nœuds en G_1

Parmi eux, 1 et 5 ont le degré le plus élevé, mais 1 a la probabilité la plus faible, il est donc sélectionné. Maintenant $N_{G_1} = 3, 1,$ 2 et 5 ont le degré le plus élevé, mais 5 est choisi en raison de son degré le plus élevé. Enfin, les 2 et 4 sont insérés dans la séquence. La séquence d'exploration résultante est $N_{G_1} = \{3, 1, 5, 2, 4\}$. Une fois la séquence prête, $VF3$ peut explorer le graphe G_1 afin de préparer les ensembles terminaux (voir Tableau 3.3) et générer l'arbre de couverture illustré à la Figure 3.6. Notez que la profondeur maximale est égale à la taille de G_1.
A la fin de ce processus, $VF3$ est enfin prêt à démarrer la correspondance.

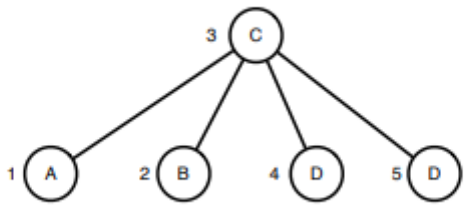


Figure 3.6: Arbre de couverture produit par VF3 en utilisant la séquence d'exploration $N_{G_1} = \{3, 1, 5, 2, 4\}$

Niveau	ensemble de base	Ensemble de bornes
0	$\emptyset$	$\emptyset$
1	$\{3_3\}$	$\{2_2, 4_2, 5_1\}$
2	$\{3_3, 1_4\}$	$\{2_2, 4_2, 5_1\}$
3	$\{3_3, 1_4, 5_1\}$	$\{2_2, 4_2\}$
4	$\{3_3, 1_4, 5_1, 2_2\}$	$\{4_2\}$
5	$\{3_3, 1_4, 5_1, 2_2, 4_2\}$	$\emptyset$

Table 3.3: Ensembles Core et Terminal de G_1

Pour chaque niveau de la recherche en profondeur, pour chaque nœud de l'ensemble de terminaux a été indiquée la classe à laquelle il appartient.

3.4 Conclusion

Maintenant nous devons comprendre si $VF3$ est vraiment capable de remplacer $VF2$ et surtout, s'il est capable de rivaliser avec l'état de l'art. Comme discuté dans la section précédente, nous avons effectué une large analyse afin de montrer à la fois les points forts et les faiblesses de $VF3$. Une autre remarque préliminaire à faire concerne le $VF3$. Nous laissons l'algorithme générer une classe pour chaque étiquette différente et, comme décrit dans la section précédente, nous ne sommes pas en mesure d'utiliser des informations structurelles dans la fonction de classification. Ainsi, ce dernier n'affecte pas les résultats sur les graphes non étiquetés et dans ce cas, $VF3$ est essentiellement récompensé par la procédure de tri, le prétraitement et la nouvelle procédure de sélection des couples candidats. Sur les graphiques étiquetés, la fonction peut être à la fois un avantage et un inconvénient. En effet, même si les nouvelles règles basées sur les classes d'anticipation fournissent un élagage plus fort des états irréalisables, lorsque le nombre d'étiquettes différentes augmente, la classification nécessite une quantité de mémoire supplémentaire.

Chapitre 4

Implémentation et résultats

4.1 Introduction :

Dans ce chapitre, nous commençons par un simple aperçu du langage Python que nous avons utilisé pour développer notre application. Nous présentons ensuite notre expérimentation visant à implémenter les algorithmes que nous avons détaillés dans le chapitre précédent et qui sont les algorithmes de référence pour toute comparaison de performances avec des algorithmes de parité pour les (sous)graphes. L'algorithme VF3 a amélioré l'algorithme VF2 et fait passer ce domaine au niveau supérieur en incorporant des règles de faisabilité dans l'algorithme. Notre expérience consiste alors à trouver la symétrie de deux (sous) graphes en appliquant des algorithmes et à agrandir deux graphes à la fois. Et au final, nous présentons le résultat que nous avons obtenu grâce à cette application.

4.2 Conception architecturale

Il est primordiale à la conception de tout système informatique de choisir le modèle d'architecture qui lui sera adéquat pouvant assurer un bon fonctionnement, des meilleurs performances ainsi que la réutilisation et l'interconnexion fiable de ce système avec d'autres. C'est à cet effet que nous optons pour le modèle MVC qui sera également très pratique pour gérer l'interaction entre les différents composants de notre application. Nous décrivons cette architecture dans la section suivante.

Architecture MVC L'architecture MVC (modèle, vue et contrôleur) est une architecture à trois couches utilisée pour la programmation client/serveur et d'interface graphique. C'est un modèle architectural très puissant qui intervient dans la réalisation d'une applica- tion. Il tire sa puissance de son concept de base qui est la séparation des données (modèle), de l'affichage (vue) et des actions (contrôleur). C'est trois couches sont décrites comme suit : - Modèle Il correspond aux données stockées généralement dans une base de données. Dans un langage orientée objet ces données sont exploitées sous forme de classes. Le modèle peut aussi agir sur la vue en mettant à jour ses données. - Vue Ne contenant que les informations liées à l'affichage, la vue se contente d'afficher le contenu qu'elle reçoit sans avoir connaissance des données. En bref, c'est l'interface homme machine de l'application. - Contrôleur Le contrôleur sert de base à récupérer les informations, de les traiter en fonction des paramètres demandés par la vue (par l'utilisateur), puis de renvoyer à la vue les données afin d'être affichées. C'est donc l'élément qui va utiliser les données pour les envoyer à la vue. L'interaction entre ces trois couches est décrite à l'aide de la figure 4.1.

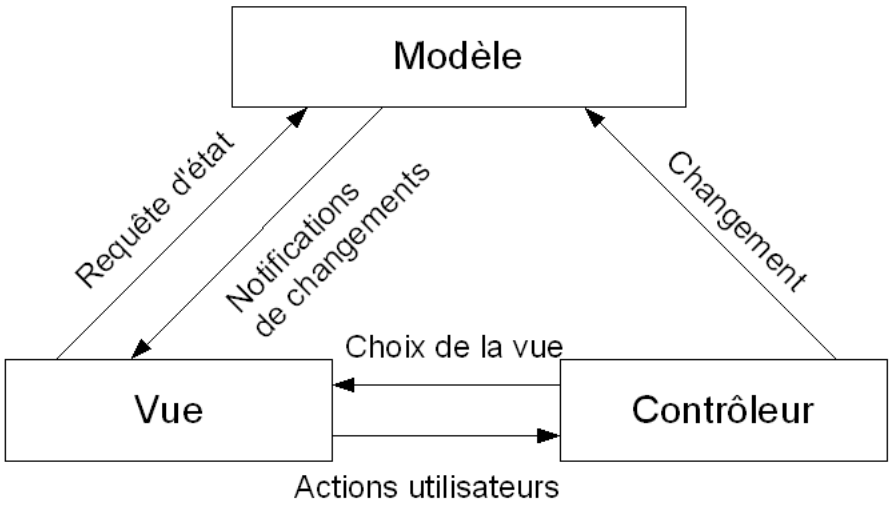


Figure 4.1: Architecture MVC

4.3 Historique sur le langage python:

Figure 4.2: Logo de langage python.

Python est une plateforme complète et généraliste pour le développement logiciel, très facile d'accès et capable de se spécialiser de manière très pointue dans la quasi-totalité des domaines informatiques. Python est utilisé par un public très large, des développeurs web professionnels, des chercheurs en intelligence artificielle ou en bioinformatique, des administrateurs systèmes, ou même des programmeurs occasionnels. C'est le mélange de polyvalence et de facilité qui fait la force de Python. Avec un bref apprentissage et un minimum d'efforts, vous

serez capable d'envisager n'importe quel type d'application de manière extrêmement efficace et de la terminer (ou de la faire terminer) en temps voulu.

Le développement de Python ayant commencé à la fin des années 1980, son déploiement dans l'industrie a commencé vers les années 2000. Aujourd'hui, Python est devenu très populaire auprès des développeurs : beaucoup de projets viennent peupler un écosystème déjà très riche, et ce dans tous les domaines. La plateforme bénéficie donc d'une visibilité croissante, qui s'accentuera encore dans les prochaines années.

La plateforme Python ressemble vue d'en haut :

- Un langage dynamique, interactif, interopérable et très lisible
- Un vaste ensemble de bibliothèques et frameworks spécialisés
- Des outils d'édition, de test et d'industrialisation
- Le support d'une communauté d'entreprises, d'individus et d'associations
- Un marché en forte croissance

4.4 Environnement python:

4.4.1 présentation de python:

Python est un langage interprété, c'est-à-dire que chaque ligne de code est lue puis interprétée afin d'être exécutée par l'ordinateur. Pour vous en rendre compte, ouvrez un shell puis lancez la commande :

`python`

La commande précédente va lancer l'interpréteur Python. Vous devriez obtenir quelque chose de ce style pour Windows :

```
PS C:\Users\pierre> python
Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)] [..
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 4.3: Le style de commande pour windows .

pour Mac OS X :

```
iMac-de-pierre:Downloads$ python
Python 3.7.1 (default, Dec 14 2018, 19:28:38)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 4.4: Le style de commande pour Mac OS X .

Ou pour Linux :

```
pierre@jeera:~$ python
Python 3.7.1 (default, Dec 14 2018, 19:28:38)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 4.5: Le style de commande pour Linux.

4.4.2 Les bibliothéque utiliser:

NetworkX:

NetworkX est une bibliothèque traitant de la création , importation , exportation , manipulation , algorithmes , traçage de la base de données graphique. Vous pouvez commencer à utiliser plusieurs types de graphiques de réseau.

```
import networkx as nx
```

Comme avec tout autre package Python, NetworkX peut être installé à l'aide de pip, Miniconda , Anaconda et du code source.

```
pip install networkx
```

Pour utiliser pip, vous devez avoir setuptools installé.

Matplotlib:

Pour utiliser matplotlib, il faut d'abord l'importer. Comme tous les modules de Python, la syntaxe à utiliser est import. Le module intéressant est en fait un sous-module de matplotlib qui se nomme matplolib.pyplot . et d'importer sous la forme :

pip install matplotlib

C'est à dire en utilisant un alias qui rebaptise ce module plt (vous pouvez bien sûr choisir un autre nom, mais plt est la convention qui, désormais, revient le plus souvent). Une fois importé sous ce nom, les différentes fonctions et constantes de ce module s'appelleront en faisant précéder leur nom de plt., comme par exemple : plt.figure().

PyQt5:

est un binding de Qt pour le langage Python. La documentation de PyQt n'est pas aussi complète que celle de Qt → il faut parfois se référer à celle de Qt (en C++, mais la transcription en Python est assez facile).

Application minimale de PyQt:

```
import sys
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *

class MaFenetre(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Pyqt')

def main():
    app = QApplication(sys.argv)
    fenetre = MaFenetre()
    fenetre.show()
    app.exec()

if __name__ == '__main__':
    main()
```

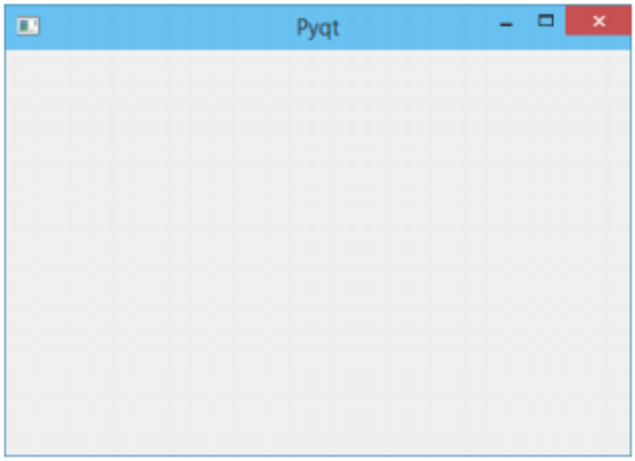


Figure 4.6: Application minimale de PyQt.

PySide2:

PySide est un binding qui permet de lier le langage Python avec la bibliothèque Qt, disponible à l'origine en C++. Il constitue une alternative aux interfaces Tkinter.

PySide se distingue de PyQt, le binding historique, par le fait qu'il est disponible sous licence LGPL, c'est d'ailleurs ce qui a déclenché son développement.

4.5 Présentation de l'application:

Après la presentation des différents concepts nécessaires pour la bonne compréhension et la réalisation de notre travail, nous passons à l'étape de réalisation de notre application qui a pour objectif d'implémenter l'algorithme d'appariement de graphes VF2. L'architecture du système contient trois modules : le module (Graph Creation) , (Graph Visualization), et le module (Matching graphs).

4.5.1 Module (création de graphe):

Ce Module vous permet de creer de nouveau graphes via une interface graphique, la fonction principale de ce module est de fournir une interface utilisateur adaptee aux besoins de l'utilisateur pour creer des graphes d'une facon simple et innovante.

4.5.2 Module (Visualisation de graphe):

Ce module permet de voir chaque graphe dans une interface simple qui possede des fonctionalites comme le zoom, enregistrement du graphe sous format .png ,...etc

4.5.3 Module (Matching graphe):

Ce module permet d'apparier deux graphes en appliquant l'algorithme VF2 qui est un algorithme d'isomorphisme de (sous-)graphes reconnu.

L'organigramme de notre application est représentée par la figure 4.7

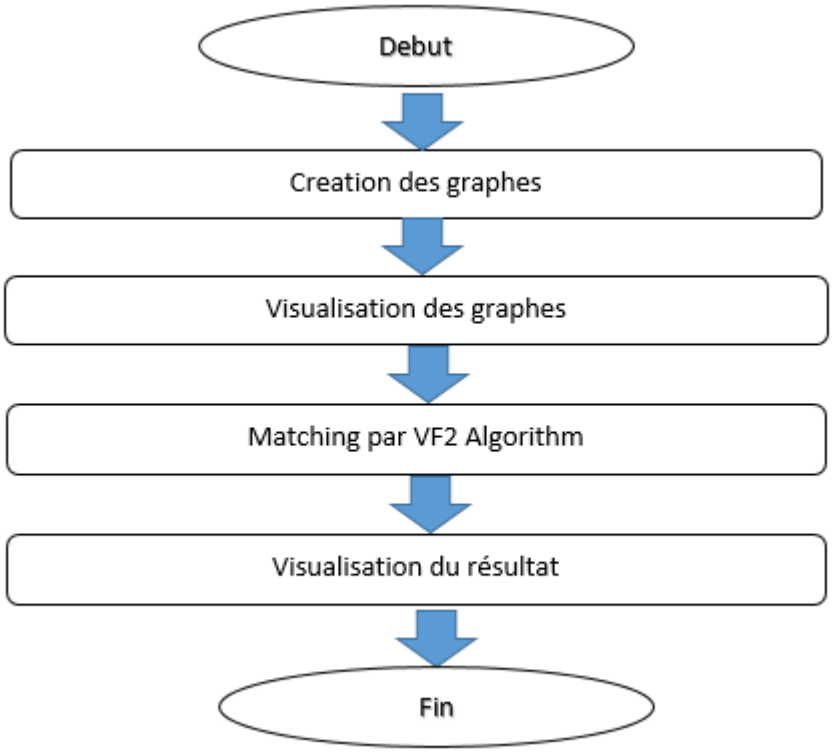


Figure 4.7: Organigramme de l'application

4.5.4 Présentation de l'interface principale de l'application

Apres avoir cree les graphes puis execute l'algorithme VF2, le resultat s'affiche en bas de l'application

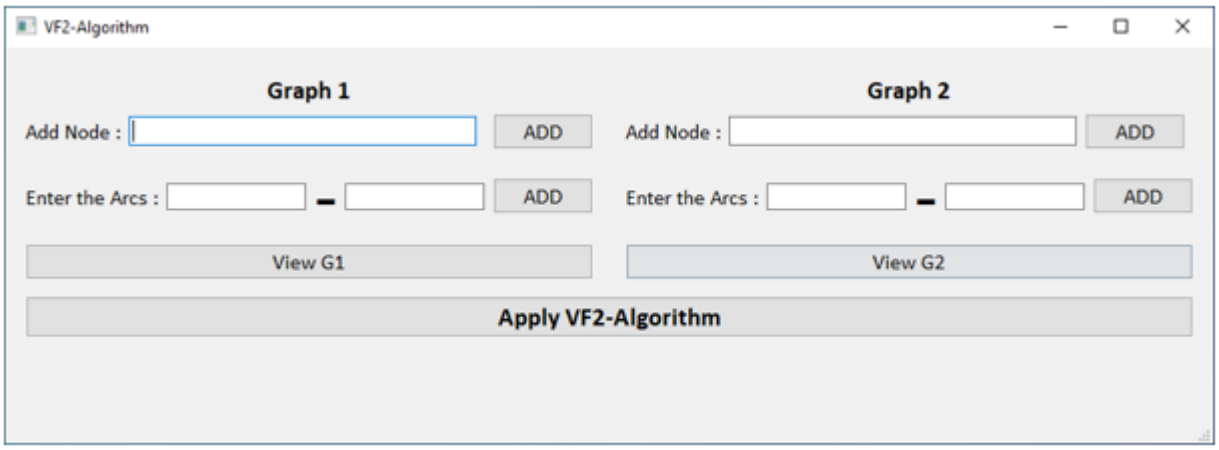


Figure 4.8: Interface principale de l'application

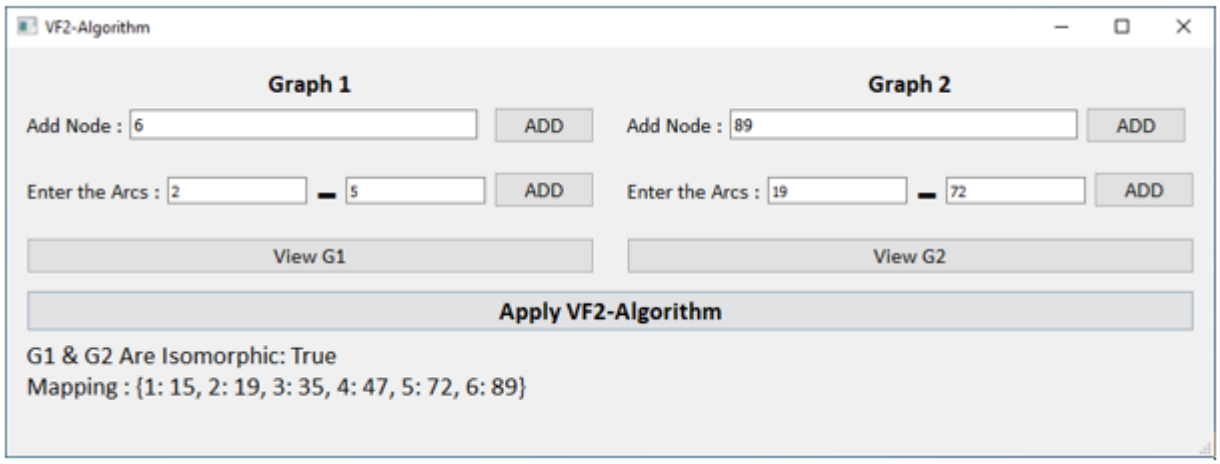


Figure 4.9: Résultats obtenus en appliquant l'algorithme VF2

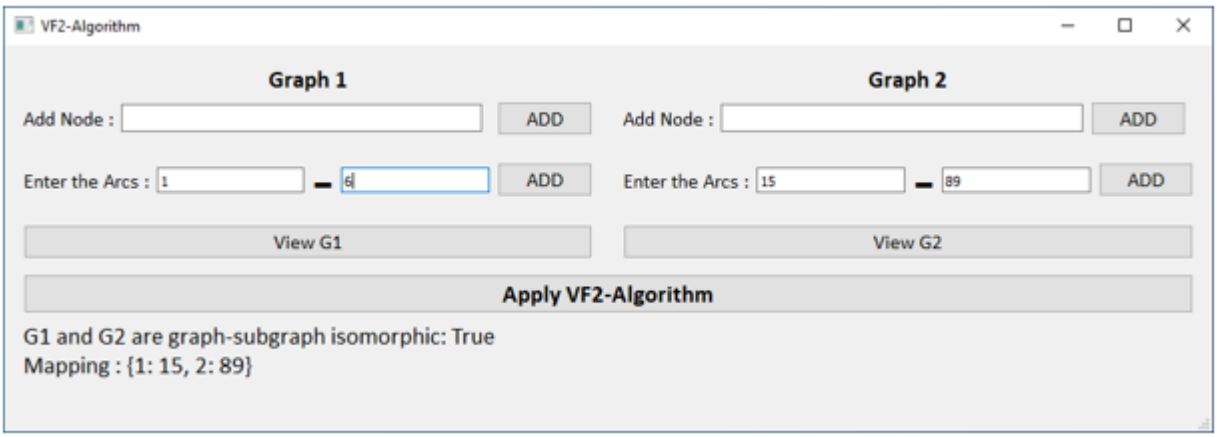


Figure 4.10: Résultats obtenus en appliquant l'algorithme VF2 dans le cas ou un sous graphe de G1 est isomorphic à G2

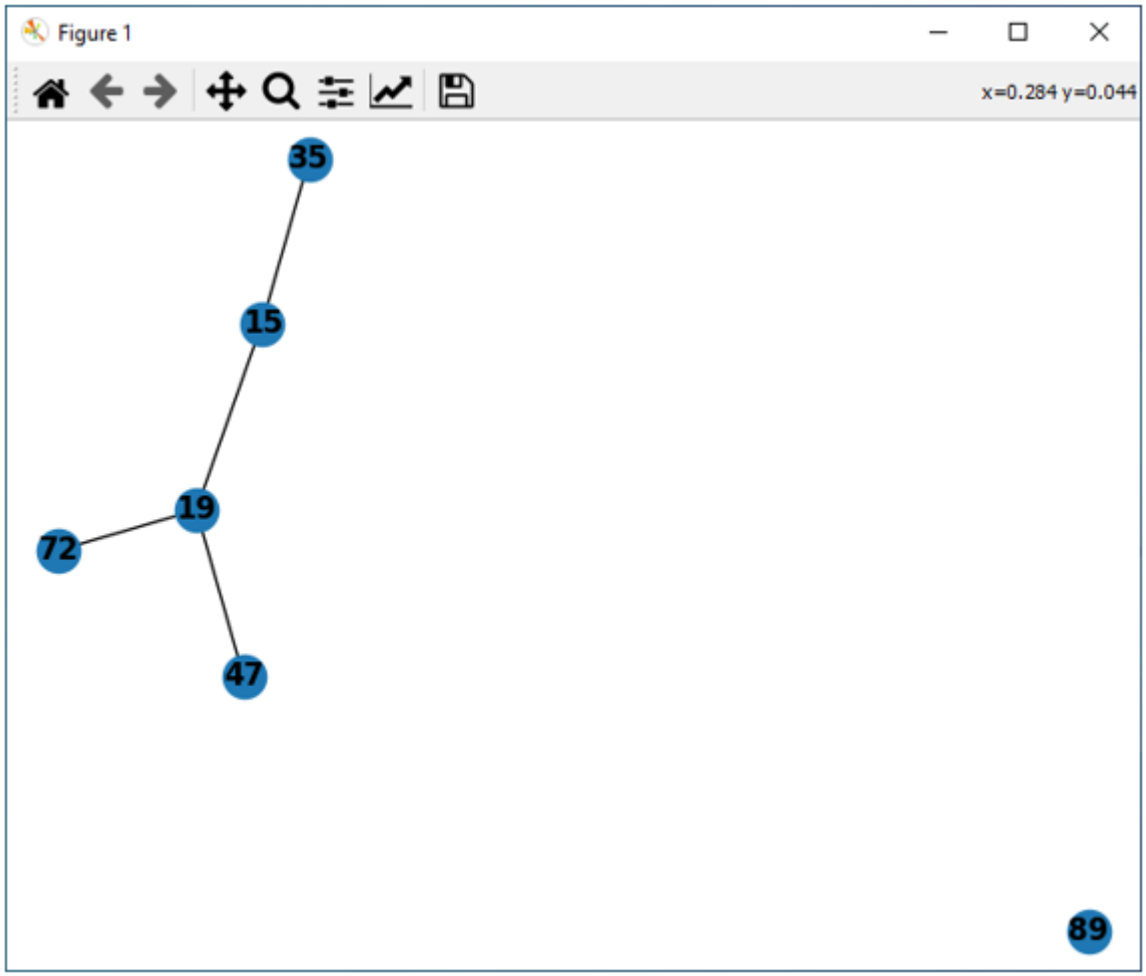


Figure 4.11: Representation d'un graphe

4.6 Conclusion:

Il était difficile de comparer deux graphes, mais après l'émergence de certains algorithmes, dont les plus importants sont $VF2$ et $VF3$.

Il est devenu facile de comparer deux graphes, et cela a été expliqué dans le dernier chapitre.

Conclusion générale

L'appariement de graphes est essentiel dans plusieurs domaines qui utilisent des informations structurées. À l'exception de classes spéciales de graphes, l'appariement de graphes a, dans le pire des cas, une complexité exponentielle ; cependant, il existe des algorithmes qui montrent un temps d'exécution acceptable, tant que les graphiques ne sont pas trop grands et pas trop denses. Dans ce mémoire, nous introduisons un algorithme d'isomorphisme de sous-graphe, VF2 et VF3, particulièrement efficace dans le cas difficile des graphes avec des milliers de nœuds et une densité de bords élevée.

Et dans notre mémoire, nous avons étudié et développé une application qui vise à comparer deux graphes à l'aide de l'algorithme $VF2$.

Liste des tableaux

Table des figures

Bibliographie

[1] Matthew Wyatt Saltz B.S.A, Fast Algorithm for Subgraph Pattern Matching on Large Labeled Graphs- University of Georgia, August 2013

[2] N. Saadia, B. Ghezala Etat de l'art des algorithmes d'appariement des graphes (Etude et Implémentation), Mémoire fin d'étude Master en Informatique, 2017 / 2018.

[3] Thomas Bärecke, Isomorphisme Inexact de Graphes par optimisation évolutionnaire Université Pierre et Marie Curie , 2009.

[4] D. Blostein, "Graph transformation in document image analysis : approaches and challenges", Workshop on graph-based representations in pattern recognition, 2005.

[5] Didier Müller-Introduction à la théorie des graphes. http ://www.apprendre-en-ligne.net/graphes/

[6] Rashid Jalal Qureshi Reconnaissance de formes et symboles graphiques complexes dans les images de documents Université François Rabelais Tours, Mars 2008.

[7] Salim Jouili Indexation de masses de documents graphiques : approches structurelles Université Nancy, France 2011 .

[8] Moultazem Ghazal Contribution à la gestion des données géographiques : Modélisation et interrogation par croquis- Université de Toulouse III - Paul Sabatier ,Juillet 2010.

[9] Mme Sabine De Blieck Gastton Lagrafe arrive, Faites Graphe à vous ! défis pour découvrir la théorie des graphes UCL-FSA, 2010.

[10] Bouchaour hamza cherif Abstraction fonctionnelle de circuit par isomorphisme de graphe Université d'Oran, 2011.

[11] R. Allen, L. Cinque, S. Tanimoto, L. G. Shapiro, and D. Yasuda, “A parallel algorithm for graph matching and its maspar implementation,” IEEE Transactions on Parallel and Distributed Systems,1997.

[12] S. Umeyama An eigendecomposition approach to weighted graph matching problems. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1988.

[13] Thomas Bärecke, Isomorphisme Inexact de Graphes par optimisation évolutionnaire Université Pierre et Marie Curie , Paris France 2009.

[14] P. N. Suganthan. Structural pattern recognition using genetic algorithms. Pattern Recognition, 2002.

[15] C. Mauro, M. Diligenti, M. Gori and M. Maggini, "Similarity learning for graphbased image representations", Workshop on graph-based representations in pattern recognition, Ischia, Italy, 2001.

[16] Moultazem Ghazal, Contribution à la gestion des données géographiques : Modélisation et interrogation par croquis , Université de Toulouse III , Paul Sabatier Juillet 2010

[17] R. A. Wagner and M. J. Fisher, “ The string-to-string correction problem”. Journal of the ACM , 1974.

[18] A. Robles-Kelly and E. Hancock, “Graph edit distance from spectral seriation”, IEEE Transactions on pattern analysis and machine intelligence, 2005.

[19] M. Neuhaus and H. Bunke, “A probabilistic approach to learning costs for graph edit distance”, Conference on pattern recognition, 2004.

[20] G.Chartrand, G. Kubicki, and M. Schultz, “Graph similarity and distance in graphs”. Journal of aequationes mathematicae, 1998.

[21] O. Grygorash, Y. Zhou, Z. Jorgensen, Minimum Spanning Tree Based Clustering Algorithms,18th IEEE International Conference on Tools with Artificial Intelligence, 2006.

[22] A. ALILAOUAR, “Contribution à l’interrogation flexible de données semi-structurées”, thèse de doctorat, Université Paul Sabatier, 2007

[23] C. ZAYANI, Contribution à la définition et à la mise en œuvre de mécanismes d’adaptation de documents semi-structurés, thèse de doctorat, Université Paul Sabatier, 2008

[24] L. Cordella, P. Foggia, C. Sansone, and M. Vento, “A (sub)graph isomorphism algorithm for matching large graphs,” IEEE Transactions on Pattern Analysis and Machine Intelligence, 2004.

[25] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, A subgraph isomorphism algorithm and its application to biochemical data, BMC Bioinformatics, 2013.

[26] C. Solnon, Alldifferent-based filtering for subgraph isomorphism Artificial Intelligence, 2010.

[27] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, An indepth comparison of subgraph isomorphism algorithms in graph databases, Proc. VLDB Endow., 2012.

[28] V. Carletti, P. Foggia, and M. Vento, “Performance Comparison of Five Exact Graph Matching Algorithms on Biological Databases,” New Trends in Image Analysis and Processing - ICIAP, 2013.

[29] V. Carletti, P. Foggia, M. Vento, and X. Jiang, " Report on the first contest on graph matching algorithms for pattern search in biological databases," in Proc. of the 10th IAPR-TC15 Intl. Workshop on Graph-based Representations in Pattern Recognition (GbR2015).