الجمهورية الجزائرية الديمقراطية الشعبية

République Algérienne Démocratique et Populaire

وزارة التعليم العالي والبحث العلمي

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Dr. Tahar Moulay
SAIDA
Faculté : Technologie
Département : Informatique

جامعة د. الطاهر مولاي سعيدة

كلية: التكنولوجيا
قسم: الإعلام الآلي

Mémoire de fin d'étude de Master en Informatique

Option : Modélisation Informatique des Connaissances et du Raisonnement

Thème

Etat de l'art des algorithmes d'appariement des graphes

(Etude et Implémentation)

Présenté par :

Mme NEZAI Saadia

Mme BENDJEBBOUR Ghezala

Encadré par :

Mr RAHMANI Mohamed

Année Universitaire : 2017 / 2018

Remerciements

Nous tenons à remercier très sincèrement Messieurs les Professeurs du département d'informatique de la faculté des sciences et de technologie de l'Université Moulay Tahar de Saida et spécialement le chef de département Monsieur Chawki Okbani.

Nous tenons à exprimer notre profonde gratitude à Monsieur Mohamed RAHMANI, superviseur de ce projet pour son suivi, ses conseils et ses remarques qui ont permis d'améliorer grandement la qualité de notre travail.

Nous remercions infiniment nos maris pour toute leur confiance et leur patience.

Un très grand merci à Monsieur Omar Fekir pour son aide et son assistance et à notre collègue Djamel Mostefai.

Nous tenons également à remercier les membres de jury pour l'honneur qu'il nous a apporté en participant à ce jury.

Mme Nezai Saadia

Mme Bendjebbour Ghezala

Dédicaces

Je dédie ce travail à :

Ma mère qui m'a très encouragé pour poursuivre mes études,

Mon mari.

Mes enfants Insaf, Khalil, Amira, Bilal et Abdelkader.

Toutes mes sœurs.

Mme Bendjebbour Ghezala

Je dédie ce travail à :

Mon mari Miloud pour son aide et sa compréhension.

Ma mère et mon père.

Mes petites filles Hayet et Batoul.

Mes sœurs Ilham et Assma

Tous mes frères.

Mme Nezai Saadia

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. Implémenter des algorithmes d'appariement de graphes et faire une étude comparative était un deuxième axe de notre travail.
Mots-clés : Appariement ; graphe ; isomorphisme ; mise en correspondance, algorithme d'Ullmann et algorithme VF2.

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. Implementing graph matching algorithms and doing a comparative study was a second axis of our work.

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

ملخص
الهدف من عملنا هو القيام بحوصلة شاملة عن خوارزميات مطابقة الرسم البياني. توجد فئتان رئيسيتان من الطرق لحل مشكلة تطابق الرسم البياني: المطابقة التامة والمطابقة غير التامة. في الفئة الأولى ، نبحث عن التماثل الذي يتطلب الحفاظ على هياكل الرسم البياني في حين أن الفئة الثانية هدفها هو حساب مقياس التشابه لتحديد مدى تشابه الرسوم البيانية. كان تنفيذ خوارزميات مطابقة الرسم البياني وإجراء دراسة مقارنة بينها محورًا ثانيًا لعملنا.

كلمات البحث: تطابق الرسم البياني. التماثل. المطابقة، خوارزمية Ullmann وخوارزمية.VF2

Sommaire

Remerciements i
Dédicaces ii
Sommaire vi
Liste des figures ix
Liste des Tableaux x
Introduction Générale xi
1 Introduction et problématique 1
2. Objectif 2
3. Organisation du document 2
Chapitre I 3
Notions de base sur les graphes et leur appariement 3
1. Introduction 4
2. Théorie de graphes 4
2.1 Quelques définitions 5
2.1.1 Graphe : 5
2.1.2 Ordre, degré et distance d'un graphe 6
2.1.3 Boucle, chaîne, chemin, cycle et circuit 6
2.2 Quelques types de graphes 7
2.2.1 Graphe simple 7
2.2.2 Multi-graphe 7
2.2.3 Graphe connexe et non connexe 8
2.2.4 Graphe complet ou clique 8
2.2.5 Graphe biparti 8
2.2.6 Graphe planaire 8
2.2.7 Graphe étiqueté 9
2.2.8 Sous-graphe 9
2.2.9 Graphe partiel 9
2.2.10 Arbre et de foret 10
2.3 Parcours de graphes 10
2.4 Représentation de graphe 10
2.4.1 Matrices d'incidence et d'adjacence 10
2.4.2 Liste d'adjacence 12
2.5 Notions de base sur l'appariement de graphes 13

2.5.1 Isomorphisme et homomorphisme ... 13
2.5.2 Isomorphisme de sous-graphes ... 14
2.5.3 Complexité d'appariement de graphes ... 14
2.5.4 Notions de similarité entre graphes ... 15
3. Conclusion ... 17
I. Chapitre 2 : ... 18
II. Etat de l'art des algorithmes d'appariement de graphes ... 18
1 Introduction ... 19
2 Appariement exact de graphes ... 19
2.1 Algorithmes basés sur la représentation canonique des graphes ... 21
2.1.1 Algorithme Nauty ... 21
2.2 Algorithmes basés sur l'arbre de recherche ... 22
2.1.1 Algorithme de Corneil et Gotlieb ... 22
2.2.2 Algorithme d'Ullmann ... 23
2.2.3 Algorithmes VF/VF2 ... 24
2.3 Méthodes basées sur la décomposition ... 24
3 Appariement inexact ... 25
3.1 Les techniques basées sur l'arbre de recherche ... 26
3.2 Les techniques basées sur l'optimisation continue ... 26
3.3 Les techniques basées sur la théorie spectrale ... 28
3.4. Autres techniques de mise en correspondance de graphes ... 28
3.4.1 Les méthodes basées sur le clustering ... 29
3.4.2 Les méthodes basées sur les noyaux de graphes ... 29
3.4.3 Les méthodes basées sur l'apprentissage ... 30
4 Conclusion ... 31
Chapitre 3 : ... 32
L'algorithme d'Ullmann et l'algorithme VF2 ... 32
1. Introduction ... 33
2 Algorithme d'Ullmann ... 33
2.1 Algorithme d'énumération ... 36
2.2 Algorithme employant la procédure de raffinement ... 37
2.3 Exemple illustratif du fonctionnement de l'algorithme d'Ullmann ... 39
3 Algorithmes VF et VF2 ... 42
3.1 Présentation de l'algorithme VF2 ... 43
2.3. Description de l'algorithme VF2 ... 44
2.4. Définition de la fonction de faisabilité ... 45

2.5. Exemple du fonctionnement de l'algorithme VF2 48
4 Conclusion 51
III. Chapitre 4 .... **52**
IV. Implémentation et résultats **52**
1 Introduction 53
2 Approche orienté objet 53
3 Historique sur le langage Java 54
4 Environnement Java 56
4.1. Présentation de Java Eclipse 56
4.2 JavaFX 58
4.3 SceneBuilder 59
4.4 Bibliothèque JUNG 60
4.5 Matériel utilisé 60
6. Présentation de l'application 60
6.1-Module 'Construire / Importer Graphe' : 61
6.2-Module 'Matching Graphe' : 61
6.3 Présentation de l'interface principale de l'application 62
6.4 Interprétation des résultats 64
7 Conclusion 65
Conclusion Générale **66**
Bibliographie **68**

Liste des figures

Figure 1. 1 Un graphe orienté a 8 sommets 6
Figure 1. 3 Exemple de chemin et cycle 7
Figure 1. 4 Exemple d'un multi-graphe 7
Figure 1. 5 Exemple de graphe non connexe possedant 2 composantes connexes. 8
Figure 1. 6 Exemple de graphe biparti (a) - exemple de graphe planaire (b) 8
Figure 1. 7 Exemple de sous graphe et graphe partiel 10
Figure 1. 8 Exemple d'arbre et de foret 10
Figure 1. 9 Exemple de representation d'un graphe par sa matrice d'adjacence 11
Figure 1. 10 Representation d'un graphe simple oriente par sa matrice d'incidence. 12
Figure 1. 11 Liste d'adjacence d'un graphe 12
Figure 1. 12 Liste d'adjacence d'un graphe etiquete 13
Figure 1. 13 Un exemple d'une sequence d'operations d'edition 15
Figure 1. 14 Exemple de sous graphe maximum commun entre deux graphes. 16
Figure 2.1 Exemple de représentation d'un graphe par sa matrice d'adjacence 20
Figure 2.2 Exemple d'un arbre de recherche d'isomorphes de graphes 22
Figure 3.1 Algorithme d'Ullmann 34
Figure 3.2 Algorithme de backtraking 35
Figure 3.3 Algorithme forward cheking 36
Figure 3.4 Deux graphes G1 et G2 avec leur matrice d'adjacence 39
Figure 3.5 Deux graphes G1 et G2 avec leur matrices d'adjacence 40
Figure 3.6 Matrice M^d générée 41
Figure 3.7 Les isomomorphismes trouvés 42
Figure 3.8 Algorithme d'appariement VF2 44
Figure 3.9 Algorithme de calcul de p(s) 46
Figure 3.10 Exemplede deux grapes à apparier avec l'algorithme VF2 48

Figure 3.11 Les états trouvés par VF2 49
Figure 4.1 Application javafx et son graphe de scene 58
Figure 4.2 Module construire/import graphes 61
Figure 4.3 Module matching graphs 61
Figure 4.4 Organigramme de l'application 62
Figure 4.5 Interface principale de l'application 62
Figure 4.6 Résultats obtenus en appliquant l'algorithme VF2 63
Figure 4.7 Résultats obtenus en appliquant l'algorithme d'Ullmann 63
Figure 4.8 Comparison de résultats entre les deux algorithme sur un jeu d'essai 64

Liste des Tableaux

Tableau 1 : Resultats obtenus en appliquant l'algorithme d'Ullmann et VF2 64

Introduction Générale

1 Introduction et problématique

Les graphes forment un outil très efficace pour la modélisation des données structurées. La structure de graphe est caractérisée principalement par la flexibilité et la simplicité qui permettent l'utilisation de graphes dans des domaines d'applications variés. Quand les graphes sont employés pour la représentation d'objets, la comparaison d'objets revient à comparer les graphes correspondants. Ainsi, pour comparer deux graphes, on ne peut pas contenter de comparer leurs descriptions matricielles. Dans la littérature, le problème de comparaison de graphes est généralement abordé comme un problème d'appariement de graphes (*graph matching*).

L'appariement entre graphes consiste à rechercher une mise en correspondance entre les nœuds et les arêtes des deux graphes, tout en assurant la satisfaction de certaines contraintes. Généralement, les méthodes d'appariement de graphes sont divisées en deux grandes catégories : la première englobe les méthodes d'appariement exact qui nécessitent une mise en correspondance stricte entre deux graphes ou au moins entre des sous-graphes. Dans le cas général, l'appariement exact cherche à trouver l'isomorphisme entre deux graphes qui ont la même topologie. La deuxième catégorie définit les méthodes d'appariement inexact, où une mise en correspondance peut se produire entre deux graphes même s'ils sont structurellement différents. Dans cette approche, l'appariement de graphe consiste à calculer une mesure de similarité ou une distance entre deux graphes pour déterminer à quel point ils se ressemblent.

Dans la littérature, le problème d'appariement de graphes est considéré comme un problème NP-complet, car définir une mise en correspondance entre deux graphes revient à trouver toutes les mises en correspondances entres les nœuds des deux graphes ainsi que les arrêtes, c'est-à-dire trouver toutes les combinaisons qui existent. La complexité augmente lorsque les graphes deviennent d'une taille très grande.

Plusieurs travaux ont contribué à la résolution du problème d'appariement de graphes. Parmi les algorithmes les plus efficaces, on trouve les algorithmes qui sont basés sur l'arbre de recherche et qui introduisent généralement des heuristiques permettant de réduire l'espace d'exploration dans l'arbre pour réduire ainsi la complexité spatiale et temporelle engendré par ce problème.

2. Objectif

L'objectif de notre étude est d'établir, en premier lieu, un état de l'art sur les algorithmes d'appariement de graphe qui existent dans la littérature et en deuxième lieu faire une expérimentation pour implémenter deux algorithmes d'appariements de graphes basés sur l'arbre de recherche. Enfin traduire les résultats obtenus en faisant une étude comparative entre les deux algorithmes implémentés.

3. Organisation du document

Ce manuscrit commence par une introduction générale dans laquelle on présente le problème d'appariement de graphe et les méthodes dédiés à le résoudre. Dans 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. Dans le **chapitre 2**, un état de l'art des méthodes d'appariement de graphes est présenté. Dans le **chapitre 3,** on présente le détail de deux algorithmes les plus connus et les plus efficaces pour résoudre le problème de recherche d'isomorphisme de (sous-)graphe. En fin dans le **chapitre 4**, on présente notre expérimentation qui vise à implémenter l'algorithme d'Ullmann et l'algorithme VF2 et comparer les résultats obtenus en appliquant les deux algorithmes.

Chapitre I

Notions de base sur les graphes et leur appariement

1. Introduction

A cause de leur simplicité et leur puissance, les graphes sont devenus des techniques de représentation flexibles qui sont appliquées dans de nombreux domaines et récemment dans le domaine de l'informatique, en particulier dans la vision par ordinateur et la reconnaissance des formes. Il existe plusieurs exemples de graphes réels, tel que les liens entre les pages web sur Internet ou les relations entre les membres d'un réseau social. L'importance des graphes a conduit à les étudier dans des contextes théoriques et pratiques pendant plusieurs décennies car ils fournissent une façon intuitive et simple de modéliser différents types de données.

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].

Les graphes permettent alors de manipuler plus facilement des objets et leurs relations avec une représentation naïve. L'ensemble des techniques et outils mathématiques mis au point en théorie des graphes permettent de démontrer aisément des propriétés, d'en déduire des méthodes de résolution, des algorithmes, ... etc. dans le but de résoudre un problème donné.

Avant de définir notre problème qui est l'appariement de graphes, nous introduisons, dans ce chapitre, quelques notions de la théorie de graphes en définissant quelques concepts et termes propres à cette théorie, que nous avons jugé, utiles pour notre étude.

2. Théorie de graphes

L'histoire de la théorie des graphes débute avec les travaux de Léonard Euler avec le problème des ponts de Königsberg. Euler a modélisé ce problème par un graphe : un sommet est associé à chaque parcelle de terre délimitée par la rivière et une arête est associée à chaque pont les reliant [4]. Depuis, cette théorie a donné naissance à de

multiples techniques dans de nombreux domaines, en mathématique, chimie, physique, biologie, informatique, etc.

Cette théorie a deux principaux objectifs : Le premier est d'offrir un modèle pour représenter le problème, généralement l'ensemble des possibilités. Dans ce cas, le graphe est une représentation des différents éléments et des relations binaires entre eux. Le deuxième objectif de cette théorie est d'offrir des outils pour permettre de trouver les meilleures possibilités dans le but de résoudre un problème [3]**.** Dans notre contexte, nous nous intéressons aux outils qui permettent d'apparier deux graphes.

2.1 Quelques définitions

Les définitions qui suivent et les différents types de graphes sont reproduits de [2, 6 , 8 et 9].

2.1.1 Graphe :

Un graphe fini G = (V, E) est défini par l'ensemble fini V ={v_1, v_2, . . . , v_n} dont les éléments sont appelés **sommets** (vertices en anglais), et par l'ensemble fini E={e_1, e_2, . . . , e_m} dont les éléments sont appelés **arêtes** (edges en anglais).

Une arête e de l'ensemble E est définie par une paire **non ordonnée** de sommets, appelés les **extrémités** de e. Si l'arête e relie les sommets a et b, on dira que ces sommets sont **adjacents**, ou **incidents** avec e, ou bien que l'arête e est incidente avec les sommets a et b. Le graphe de la figure 1.1 représente un **graphe non orienté** à 8 sommets, nommés a à h, comportant 10 arêtes. Ce graphe est définit de la façon suivante :

G= (V, E)

V= {a, b, c, d, e, f, g, h}

E= {(a, d), (b, c), (b, d), (d, e), (e, c), (e, h), (h, d), (f, g), (d, g), (g, h)}

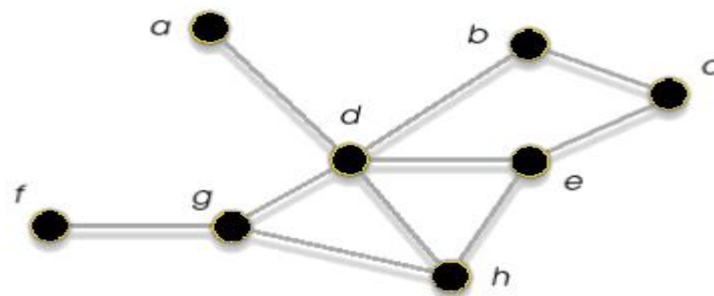


Figure 1. 1 Un graphe non oriente à 8 sommets

Un graphe orienté est un graphe dont les arêtes sont orientées et par conséquent, le couple (x, y) est différent du couple (y, x) puisque leur orientation est différente. Les arcs d'un graphe orienté, sont représentés sous forme de flèches. On les appelle **arcs** pour les distinguer des arêtes non orientées.

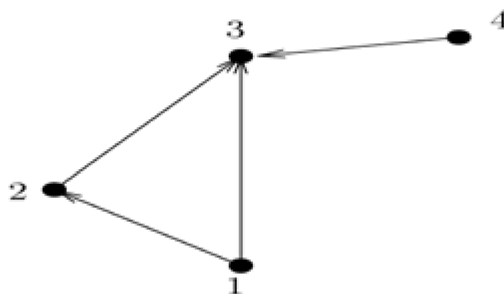


Figure 0-2 Un graphe orienté à 4 sommets

2.1.2 Ordre, degré et distance d'un graphe

On appelle **ordre** d'un graphe le nombre de sommets n de ce graphe, il est noté **|X|**. Le **degré d'un sommet** x d'un graphe est le nombre d'arêtes incidentes à x. Il est noté d(x). Pour un graphe simple le degré de x correspond également au nombre de sommets adjacents à x. La **distance entre deux sommets** u et v est le nombre d'arêtes dans le plus court (la plus petite longueur) chemin entre u et v [4].

2.1.3 Boucle, chaîne, chemin, cycle et circuit

Une arête dont les extrémités sont identiques est appelée **une boucle**. **Une chaîne simple** ou élémentaire est une chaîne 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. **Un chemin** est une chaîne dont tous les arcs sont orientés dans le même sens. La figure 1.3 montre deux exemples de chemin, le chemin simple (f, g, d, b) en rouge et le chemin (f, g, d, h, e, d, b) qui n'est pas simple car le sommet d est visité deux fois.

Un cycle ou **circuit** est une suite d'arcs partant et finissant au même sommet. Notons également qu'un cycle simple sera automatiquement une chaîne simple, dans la figure 1.3, le chemin (d, h, e, d) en vert représente un cycle.

Remarque: Les termes ‘ chemin’ et ‘circuit’ s'emploient en propre pour les graphes orientés. Pour les graphes non orientés, on parle de chaine et de cycle. Cependant la définition formelle est exactement la même dans les deux cas, seule change la structure (graphe orienté ou non) sur laquelle ils sont définis.

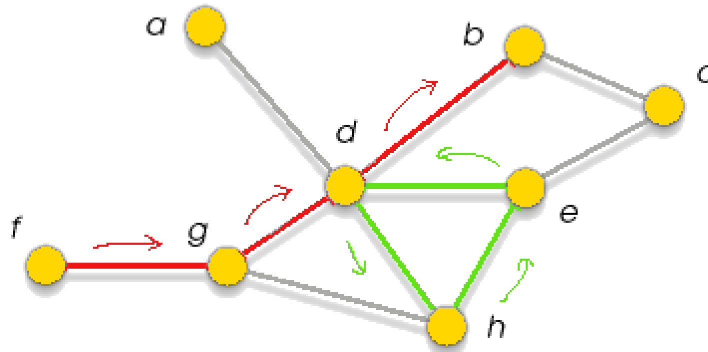


Figure 1. 2 Exemple de chemin et cycle

2.2 Quelques types de graphes

Il existe plusieurs familles de graphes, nous citons ci-après, quelques familles simples et importantes de graphes, notons que le choix d’un type de graphe dépend du problème à résoudre :

2.2.1 Graphe simple

Un graphe est **simple** si au plus une arête relie deux sommets et s’il n’y a pas de boucle sur un sommet.

2.2.2 Multi-graphe

On peut imaginer des graphes avec une arête qui relie un sommet à lui-même (une boucle), ou plusieurs arêtes reliant les deux mêmes sommets. On appelle ces graphes des multi-graphes.

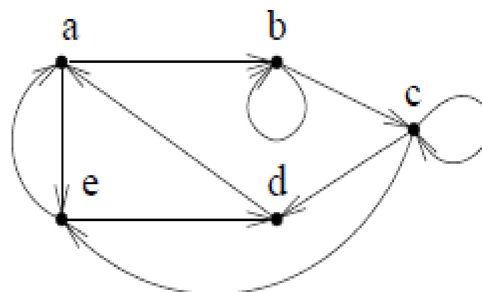


Figure 1. 3 Exemple d’un multi-graphe

2.2.3 Graphe connexe et non connexe

Un graphe est **connexe** s'il est possible, à partir de n'importe quel sommet, de rejoindre tous les autres en suivant les arêtes. Un graphe **non connexe** se décompose en **composantes connexes**.

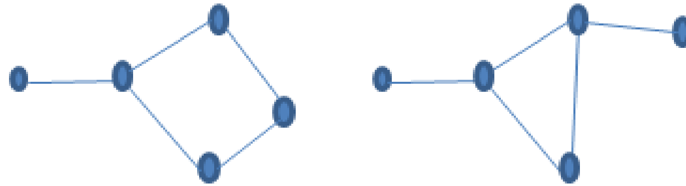


Figure 1. 4 Exemple de graphe non connexe possédant 2 composantes connexes.

2.2.4 Graphe complet ou clique

Un graphe est **complet** si chaque sommet du graphe est relié directement à tous les autres sommets.

2.2.5 Graphe biparti

Un graphe est **biparti** si ses sommets peuvent être divisés en deux ensembles *X* et *Y*, de sorte que toutes les arêtes du graphe relient un sommet dans *X* à un sommet dans Y comme illustré dans la figure 1.6 (a) :

2.2.6 Graphe planaire

Un graphe planaire est un graphe qui a la particularité de pouvoir se représenter dans un plan sans qu'aucune arête, courbe ou rectiligne, ne se croise (voir l'exemple de la figure 1.6 (b).

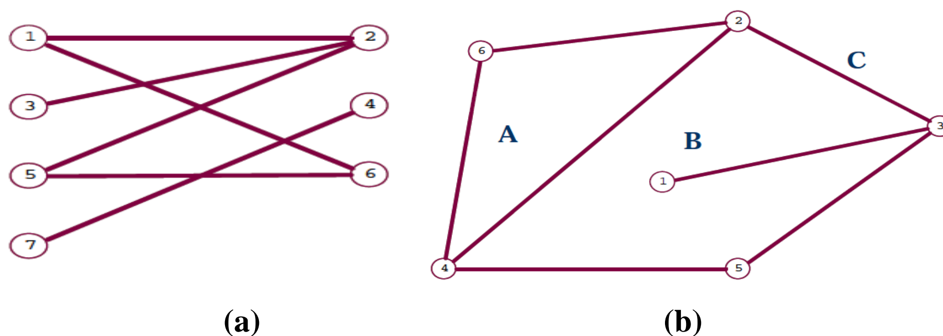


Figure 1. 5 Exemple de graphe biparti (a) - Exemple de graphe planaire (b)

2.2.7 Graphe étiqueté

Un graphe étiqueté est un graphe dont les arêtes et/ou les nœuds sont affectés d'étiquettes.

2.2.8 Sous-graphe

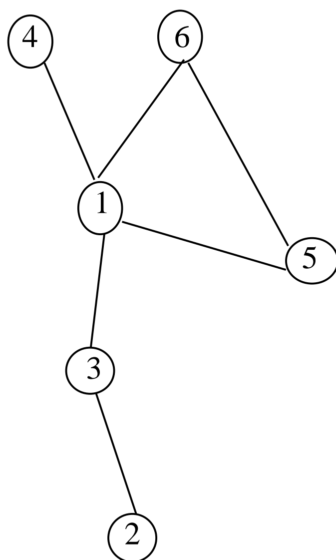
Un sous graphe est une partie d'un graphe. Toutefois, dans un sous-graphe, il n'est pas obligatoire que toutes les parties du graphe principal y apparaissent. Dans le cas où le sous-graphe est parfait, c'est-à-dire qu'il représente fidèlement et complètement une partie du graphe, ce sous-graphe est appelé sous-graphe engendré.

Autrement dit, un sous-graphe de G est un graphe H= (V ; A(V)) tel que V est un sous ensemble de X, et A(V) sont les arêtes induites par A sur V, c'est-à-dire les arêtes de A dont les deux extrémités sont des sommets de V.

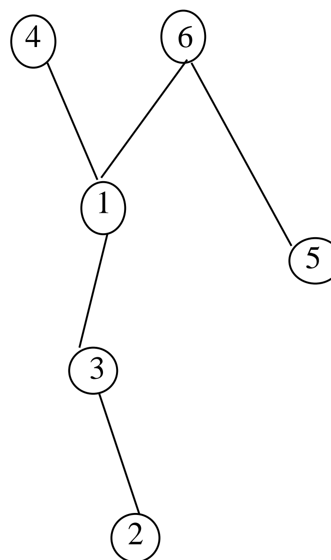
2.2.9 Graphe partiel

Soit G=(E,V) un graphe. Le graphe G_1= (E_1,V_1) est un graphe partiel de G, si V_1 est inclus dans V. Autrement dit, on obtient G_1 en enlevant une ou plusieurs arêtes du graphe G.

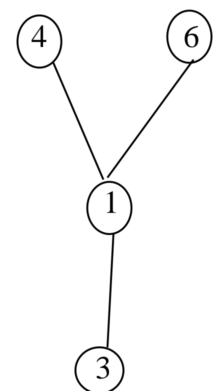
Etant donné le graphe G=(E,V) de la figure 1.7 où e={1, 2, 3, 4, 5, 6} et V={ (1,3), (1,4), (1,5), (1,6), (2,3), (3,5) ,(5,6)}. Le graphe partiel G_1= (E1,V_1) de G est induit par les sommets E_1={1, 2, 3, 4, 5, 6} et les arrêtes V_1={ (1,3), (1,4), (1,6), (2,3),(5,6)}. Le sous-graphe G_2= (E_2,V_2) de G est induit par les sommets E_2={1, 2, 4, 6} et les arrêtes V_2={(1,3), (1,4), (1,6)}



Graphe G



Graphe partiel G_1= (E_1,V_1)



Sous-graphe G_2= (E_2,V_2)

Figure 1. 6 Exemple de sous graphe et graphe partiel

2.2.10 Arbre et de foret

On appelle **arbre** tout graphe connexe sans cycle. Un graphe sans cycle mais non connexe est appelé une **forêt**.

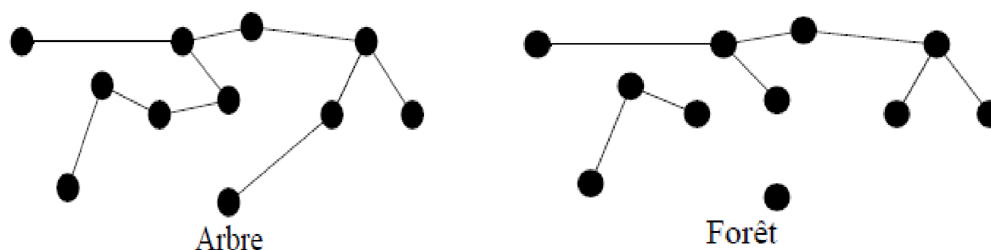


Figure 1. 7 Exemple d'arbre et de forêt

2.3 Parcours de graphes

Beaucoup de problèmes sur les graphes nécessitent que l'on parcourt l'ensemble des sommets et des arcs/arêtes du graphe. Deux principales stratégies d'exploration existent :

- **Le parcours en largeur** consiste à explorer les sommets du graphe niveau par niveau, à partir d'un sommet donné ;
- **Le parcours en profondeur** consiste, à partir d'un sommet donné, à suivre le chemin le plus loin possible puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment.

2.4 Représentation de graphe

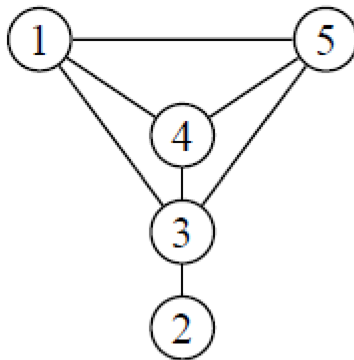
Tout graphe peut être représenté par une matrice d'incidence et d'adjacence ou par une liste d'adjacence.

2.4.1 Matrices d'incidence et d'adjacence

Bien que le dessin soit un moyen pratique pour définir un graphe, il n'est pas adapté ni au stockage des graphes dans la mémoire d'un ordinateur, ni à l'application des méthodes mathématiques pour étudier leurs propriétés. Pour ces usages, il existe deux matrices associées à un graphe : sa matrice d'incidence et sa matrice d'adjacence.

On peut représenter un graphe simple par une **matrice d'adjacences**. Une matrice (n×m) est un tableau de n lignes et m colonnes. (i, j) désigne l'intersection de la ligne i

et de la colonne j. Dans une matrice d'adjacences, les lignes et les colonnes représentent les sommets du graphe. Un « 1 » à la position (i, j) signifie que le sommet i est adjacent au sommet j.



$$M = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

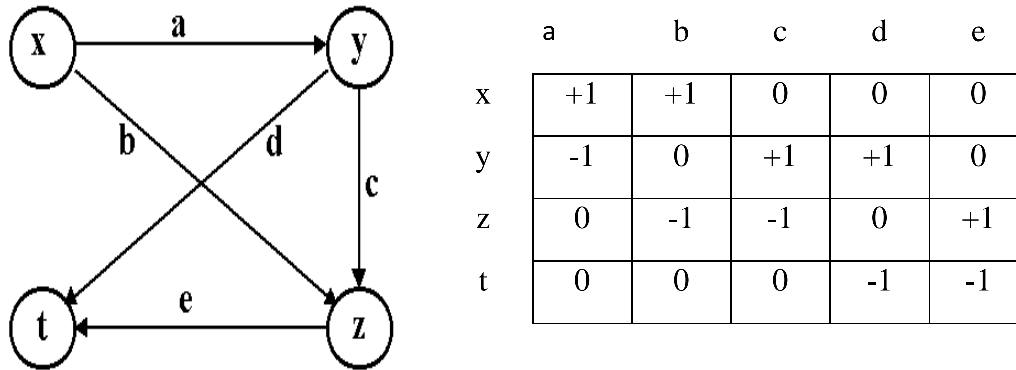
Figure 1. 8 Exemple de représentation d'un graphe par sa matrice d'adjacence

Cette matrice a plusieurs caractéristiques :

1. Elle est carrée : il y a autant de lignes que de colonnes.
2. Il n'y a que des zéros sur la diagonale allant du coin supérieur gauche au coin inférieur droit. Un « 1 » sur la diagonale indique qu'il existe une boucle.
3. Elle est symétrique : $m_{ij} = m_{ji}$. On peut dire que la diagonale est un axe de symétrie.
4. Une fois que l'on fixe l'ordre des sommets, il existe une matrice d'adjacences unique pour chaque graphe.

La matrice d'incidence sommets-arcs d'un graphe *G* est la matrice $M = (m_{ij})$ de taille n×m, définie comme suit :

$$m_{ij} = \begin{cases} 1 \text{ si } i \text{ est le sommet initoal de } j, \\ -1 \text{ si } i \text{ est le sommet terminal de } j, \\ 0 \text{ dans les autres cas.} \end{cases}$$



	a	b	c	d	e
x	+1	+1	0	0	0
y	-1	0	+1	+1	0
z	0	-1	-1	0	+1
t	0	0	0	-1	-1

Figure 1. 9 Représentation d'un graphe simple orienté par sa matrice d'incidence.

2.4.2 Liste d'adjacence

Une autre idée pour représenter les graphes en mémoire : les listes d'adjacences. Dans ce type de structure, on a un tableau de liste de sommets. Le tableau comporte autant de cases qu'il y a de sommets. Chacune des cases pointe vers une liste de sommets. Cette liste n'est rien d'autre que les successeurs du sommet considéré. La figure 1.11 montre un exemple de graphe et sa liste d'adjacence associée :

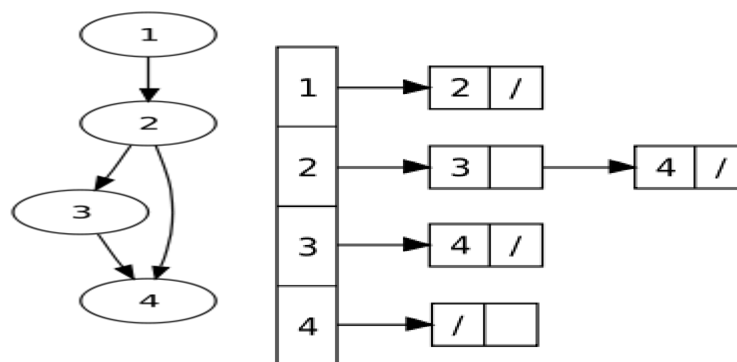


Figure 1. 10 Liste d'adjacence d'un graphe

Pour un graphe étiqueté, il faut mémoriser, dans les nœuds, de la liste la valeur du nœud. On peut donc avoir le graphe étiqueté précédent et sa liste d'adjacence associée comme suit:

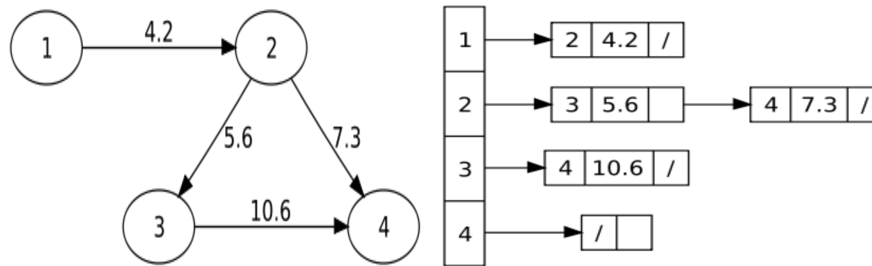


Figure 1. 11 Liste d'adjacence d'un graphe étiqueté

2.5 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 *Ei* 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 *Ei*, c'est-à-dire s'il existe une arête dans *Ei* qui relie chaque sommet dans *Vi* à un autre [7].

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'appariement exact qui nécessitent une correspondance stricte entre des graphes ou au moins entre des sous-graphes. La deuxième catégorie définit les méthodes d'appariement inexact, où une mise en correspondance peut se produire entre deux graphes même s'ils ont des structures différentes. On définira en détail, dans le chapitre 2, les deux méthodes d'appariement avec un état de l'art des algorithmes les plus connus dans la littérature.

2.5.1 Isomorphisme et homomorphisme

L'isomorphisme de graphes est une relation univoque des nœuds entre deux graphes qui respecte leurs attributs et ceux des arêtes. Dans le cas d'isomorphisme de sous-graphes, on considère deux graphes G_1 et G_2 de taille différente, $|G_2|= n_2 \leq |G_1|=n_1$. Afin de déterminer si G_2 est un sous-graphe de G_1, une telle relation est recherchée entre les nœuds du graphe G_1 et tous les sous-graphes du graphe G_2 induits par n_1 nœuds. De manière générale, on essaie de déterminer si deux graphes donnés sont égaux. Généralement, les notions d'homomorphisme et d'isomorphisme sont définies comme suit :

Homomorphisme de graphes :

Soient $G_1=(V_1,E_1)$ et $G_2=(V_2,E_2)$ deux graphes. (dans ce document, nous désignons par G_1 un graphe de données et G_2 un graphe modèle ou graphe requête).

S'il existe une relation univoque $m : V_1 \rightarrow V_2$ telle que $(v_1,v_2) \in E \Rightarrow (m(v_1),m(v_2)) \in E_1$, alors G_1 et G_2 sont homomorphes.

Isomorphisme de graphes :Soient $G_1 = (V_1,E_1)$ et $G_2 = (V_2,E_2)$ deux graphes. S'il existe une relation univoque $m : V_1 \rightarrow V_2$ telle que $(v_1,v_2) \in E \Leftrightarrow (m(v_1), m(v_2)) \in E_1$, alors G_1 et G_2 sont isomorphes.

Par conséquent, un homomorphisme est un isomorphisme lorsque la relation m est bijective. En effet, un homomorphisme de G_1 vers G_2 permet qu'il y ait une arête entre deux sommets dans G_2 faisant partie de la relation, sans qu'il y en ait entre leurs antécédents dans G_1. Dans le cas de l'isomorphisme, une telle situation est impossible[7].

2.5.2 Isomorphisme de sous-graphes

Le problème d'isomorphisme de sous-graphes consiste à trouver une application partielle entre les sommets de deux graphes, telle que les conditions définies précédemment soient satisfaites. Soient G_1 et G_2 deux graphes avec $n_2 \leq n_1$, on cherche un sous-graphe de G_1 induit par n_1 sommets. Si l'on considérait des sous-graphes en général au lieu de sous-graphes induits par des sommets, on ne traiterait pas le problème d'isomorphisme mais celui d'homomorphisme.

2.5.3 Complexité d'appariement de graphes

L'appariement de graphes est considéré comme l'un des problèmes les plus complexes. Cette complexité est due à sa nature combinatoire. Les problèmes d'appariement exact de graphes ont été jusqu'à présent classifiés comme étant dans P ou NP −complet. Quelques articles dans la littérature essayent de prouver qu'elle est NP − complet quand les deux graphes à apparier sont de types particuliers ou possèdent quelques contraintes spécifiques. D'autre part, pour quelques types de graphes, ils ont démontré que la complexité du problème d'isomorphisme est polynomiale, par exemple pour l'isomorphisme des graphes planaires [64]. Cependant, certains types de graphes

spécifiques peuvent avoir une complexité moins importante. Par exemple, le cas particulier, où le grand graphe est une forêt et le petit est un arbre. Finalement, dans l'appariement inexact de graphes, la complexité a été prouvée comme étant NP – complet [12].

2.5.4 Notions de similarité entre graphes

Il n'existe pas une règle pour calculer la similarité entre les graphes. Cela dépend généralement du type de l'application. Nous présentons, ci-après, les mesures les plus utilisées pour déterminer les similarités entre graphes.

La distance d'édition de graphes

La distance d'édition de graphes proposée par Bunke [23] recherche l'ensemble d'opérations d'édition (insertion, suppression ou substitution des nœuds et des arêtes) nécessaires pour transformer, avec un coût minimal, un graphe en un autre graphe. Si les graphes ne sont pas étiquetés, seules les opérations de suppression et d'insertion sont utilisées. Chaque opération est associée à une fonction de coût qui varie selon la quantité de distorsion introduite par la transformation.

L'ensemble d'opérations le moins couteux permettant de transformer G_1 en un graphe isomorphe à G_2 définit la distance d'édition entre G_1 et G_2.

La figure 1.13 représente un exemple illustrant l'ensemble d'opérations d'éditions qui permettent de transformer le graphe G_1 en G_2.

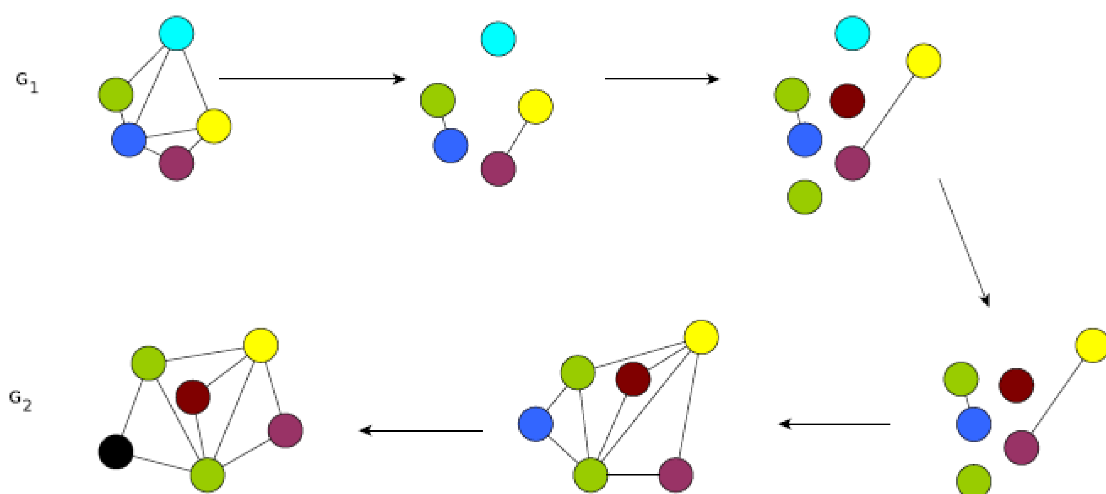


Figure 1. 12 Un exemple d'une séquence d'opérations d'édition pour transformer G_1 à G_2

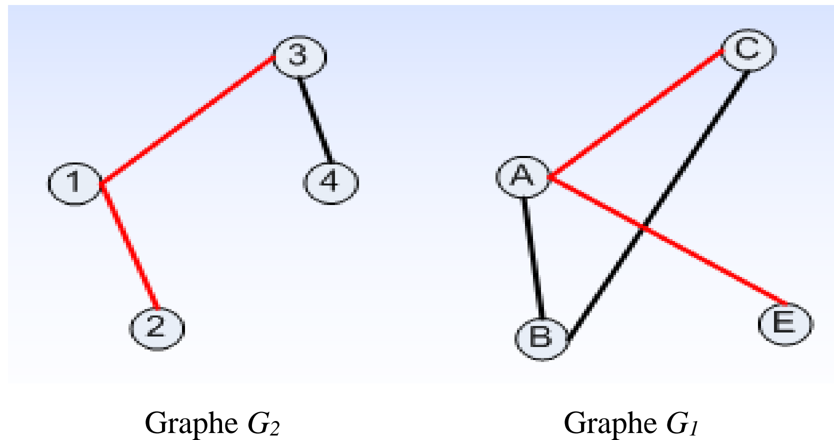
La distance basée sur la notion de sous-graphe commun maximal (SCM)

L'avantage principal de cette approche basée sur le SCM est la non utilisation des fonctions de coût, palliant ainsi l'inconvénient principal de distance basée sur les séquences d'opérations d'édition. Il a été montré par Bunke et Shearer [24] qu'une telle mesure est une métrique et conduit à une distance ayant des valeurs dans l'intervalle [0,1].

Soit deux graphes G_1 et G_2, la similarité basée sur le SCM est définie comme suit :

$$SimSCM(G1, G2) = \frac{|SCM(G1, G2)|}{|\max(G1, G2)|}$$

Où $|max(G_1,G_2)| = max(|G_1|,|G_2|)$ et $|SCM(G_1,G_2)|$ est le nombre d'arête dans $SCM(G_1,G_2)$ [13]. La figure 1.14 montre un exemple d'un sous graphe maximum commun entre deux graphes G_1 et G_2 colorié en rouge,



Graphe G_2 Graphe G_1

Figure 1. 13 Exemple de sous graphe maximum commun entre deux graphes.

Autres distances

Parmi les autres distances qui existent pour calculer la similarité entre graphes, on peut citer la distance de Wallis et all [25] qui est une mesure basée sur l'union de graphes (UG). Cette distance est utilisée pour modéliser la taille du problème. Elle est définit comme suit :

$$SimUG(G1, G2) = \frac{|SCM(G1, G2)|}{|G1| + |G2| - SCM(G1, G2)|}$$

Une autre distance présentée par Fernandez et al. [26] qui ont proposé de combiner le sous-graphe commun maximal et le super-graphe minimal commun pour calculer la distance entre deux graphes. Selon cette distance, pour que deux graphes G_1 et G_2 soient similaires, il faut que le sous-graphe commun maximal et le super-graphe minimal commun de G_1 et G_2 soient similaires [13].

3. Conclusion

Les graphes sont devenus, au fil du temps, un domaine majeur de recherche en informatique. Leur utilisation dans la représentation des données est devenue à l'instant, très fréquente. L'intérêt de l'utilisation des graphes réside dans leur capacité à représenter les variations et les différences structurelles entre plusieurs objets.

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,…), d'ordonnancer des tâches …etc. Ces nombreuses applications font de la théorie des graphes un outil appréciable d'aide à la décision et d'optimisation avec une mise en œuvre simple.

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. Nous avons aussi défini le problème d'appariement de graphes et la complexité de ce processus ainsi que les mesures de similarité qui permettent de comparer deux graphes donnés.

Chapitre 2 :

Etat de l'art des algorithmes d'appariement de graphes

1 Introduction

L'appariement de graphe consiste à mettre en correspondance deux (sous-) graphes afin de déterminer leur similarité. Dans le cas général ce processus revient à trouver un isomorphisme entre les graphes. Plusieurs approches ont été proposées, dans la littérature, pour la recherche d'un isomorphisme ou d'une distance entre deux graphes. Ces méthodes d'appariement de graphes sont réparties en deux classes d'approches : les approches exactes et les approches approximatives ou inexactes.

L'objectif de la première classe, est de déterminer un isomorphisme exact entre deux graphes. Ces méthodes sont faiblement utilisées dans les applications du monde réel à cause de leur complexité. L'objectif de l'appariement approximatif de graphes est de rendre l'appariement de graphes utilisable en pratique. Le problème d'appariement de graphes a connu un fort essor dans plusieurs domaines scientifiques. Ces travaux se sont concentrés sur le développement de nouvelles techniques de calcul d'appariement.

Dans ce chapitre, nous présentons les différentes techniques d'appariement de graphes de chaque classe ainsi que les algorithmes qui ont fait leurs épreuves pour la recherche d'isomorphisme entre graphes.

2 Appariement exact de graphes

L'appariement exact de graphes nécessite la préservation de la topologie des graphes. Pour cette raison, plusieurs conditions doivent être réalisées. Premièrement, il s'agit de trouver une fonction de mise en correspondance entre les nœuds des deux graphes tout en préservant la structure, i.e, chaque paire de nœuds adjacents du premier graphe correspond à une paire de nœuds adjacents du deuxième graphe. En plus de la préservation de la structure, un appariement exact entre deux graphes préserve aussi les étiquettes, i.e, chaque nœud d'un graphe correspond à un nœud ayant la même étiquette dans le deuxième graphe [13]. Nous donnons, par la suite, une taxonomie des approches d'appariement exact qui ont été proposées dans la littérature.

L'approche la plus stricte est l'isomorphisme de graphes où deux graphes isomorphes correspondent à deux graphes ayant exactement la même structure et les mêmes étiquettes si les graphes sont étiquetés [17].

Etant donné deux graphes $G_1 = (V_1, E_1, \alpha_1, \beta_1)$ et $G_2 = (V_2, E2, \alpha_2, \beta_2)$ où α_1, α_2 sont les attributs des nœuds de G_1 et G_2 respectivement et β_1, β_2 sont les attributs des arrêtes de G_1 et G_2 respectivement.

L'isomorphisme de graphes est une fonction bijective : $f : V_1 \rightarrow V_2$

Cette fonction doit satisfaire les conditions suivantes :

$\forall\ u \in V_1, \alpha_1 (u) = \alpha_2 (f(u))$,

$\forall\ e_1 = (u, v) \in E_1, \exists\ e_2 = (f(u), f(v)) \in E_2$, tel que $\beta1(e_1) = \beta2(e_2)$, et

$\forall\ e_2 = (u, v) \in E_2, \exists\ e1 = (f -1 (u), f -1 (v)) \in E_1$, tel que $\beta_1 (e_1) = \beta_2 (e_2)$

Pour vérifier si deux graphes sont isomorphes, il faut trouver une fonction bijective f qui met en correspondance les nœuds, un-à-un, des deux graphes tout en préservant la connectivité et les étiquettes des nœuds. Formellement, un nœud u de graphe G_1 est apparié à un nœud f(u) du graphe G_2 si et seulement si leurs étiquettes sont identiques, soit $\alpha_1 (u) = \alpha_2 (f(u))$. Également, une arête $e_1 = (u, v)$ du graphe G_1 est apparié à une arête $e_2 = f(e_1)$ du graphe G_2 si et seulement si leurs étiquettes sont identiques, soit $\beta1(e_1) = \beta_2 (e_2)$. De plus, deux nœuds u et v adjacents dans G_1 sont appariés respectivement à f(u) et à f(v) du G_2 si et seulement si f(u) et f(v) sont adjacents. La figure 2.1 (c) illustre l'isomorphisme entre le graphe (a) et le graphe (b) de la figure 2.1, les traits indiquent la correspondance entre les nœuds. Pour calculer la fonction d'isomorphisme f, il faut générer toutes les fonctions possibles et tester chaque fonction pour trouver une fonction d'isomorphisme f [17].

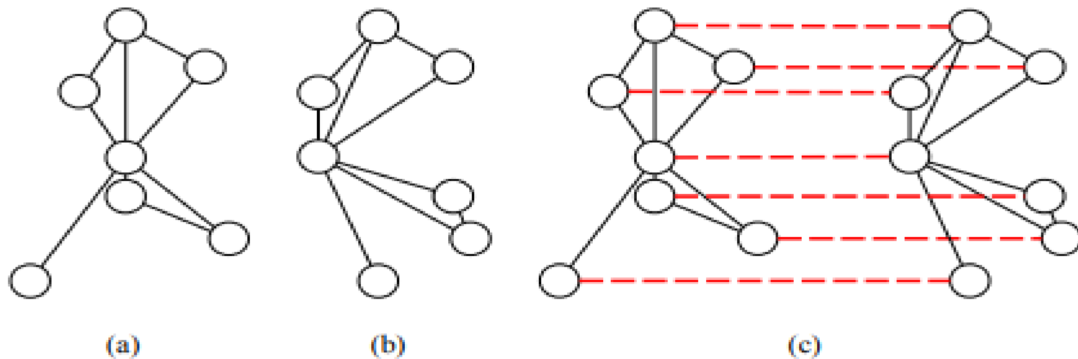


Figure 2.1 Exemple de représentation d'un graphe par sa matrice d'adjacence

On distingue deux catégories d'approches qui visent à optimiser la recherche d'isomorphisme de graphes. La première consiste à imposer des restrictions ou des contraintes sur les types de graphes à traiter. En fait, pour quelques types spécifiques de graphes, il existe des algorithmes polynomiaux pour le calcul d'isomorphisme tel que les arbres et les graphes planaires. La deuxième famille d'approches, qui est la plus utilisée dans la littérature, consiste à concevoir une nouvelle représentation de l'espace de recherche d'isomorphisme de graphes et à développer des algorithmes qui réduisent la taille de l'espace de recherche en éliminant les chemins jugés non adéquats [13]. Dans cette famille d'approches, on trouve plusieurs algorithmes dont ci-après les plus connus dans la littérature.

2.1 Algorithmes basés sur la représentation canonique des graphes

Parmi les plus célèbres algorithmes dans cette catégorie, on trouve l'algorithme Nauty qui est un algorithme très performant pour la résolution du problème de l'isomorphisme de graphes.

2.1.1 Algorithme Nauty

Cet algorithme de McKay [27] est considéré par plusieurs auteurs comme l'algorithme d'isomorphisme de graphes le plus rapide [5]. 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 [17].

Tout simplement, Mckay représente chaque graphe par sa forme canonique étiquetée en s'appuyant sur la théorie des groupes [7] et l'isomorphisme est vérifié en comparant les deux représentations canoniques des deux graphes. 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. Et malgré ça, il a été prouvé que cet algorithme a généralement une meilleure performance par rapport à d'autres algorithmes d'appariement de graphes [5].

2.2 Algorithmes basés sur l'arbre de recherche

Ces algorithmes consistent à structurer l'espace de recherche sous forme d'un arbre de recherche. La racine de l'arbre correspond à un appariement vide et les feuilles correspondent à un appariement complet entre deux graphes, s'il existe. La construction de chaque niveau de l'arbre correspond à l'ajout d'une paire de nœuds de deux graphes à apparier. La figure II.2 représente un exemple de deux graphes respectivement $G_1 = (V_1, E_1)$ et $G_2 = (V_2, E_2)$ avec un arbre de recherche d'isomorphisme entre G_1 et G2. À chaque niveau de l'arbre un nœud de $V_2 = \{a, b, c\}$ est apparié à un nœud de $V_1 = \{1, 2, 3\}$. À chaque feuille de l'arbre, la contrainte d'adjacence est vérifiée par l'examen des arêtes dans E_1 et E_2. De la même manière, les étiquettes sont vérifiées si le graphe est étiqueté. Finalement l'isomorphisme est trouvé si ces conditions (adjacence et étiquettes) sont valides dans une feuille de l'arbre de recherche [17]. Pour explorer efficacement l'arbre de recherche, plusieurs méthodes ont recours à la notion du retour arrière (backtracking). Parmi les méthodes les plus efficaces de cette catégorie, on trouve l'algorithme d'Ullmann et l'algorithme VF2.

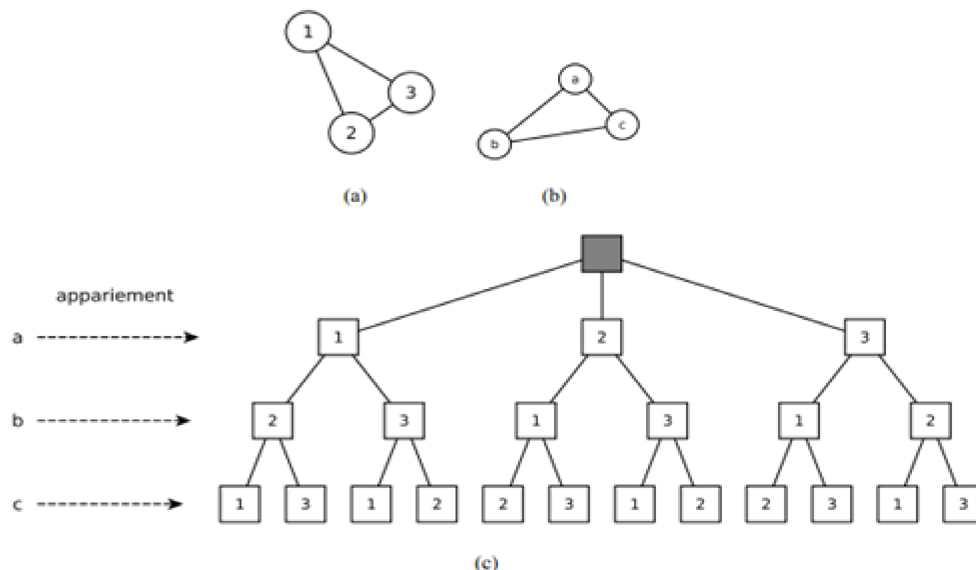


Figure 1.2 Exemple d'un arbre de recherche d'isomorphes de graphes

2.1.1 Algorithme de Corneil et Gotlieb

Concernant les algorithmes basés sur la technique de parcours d'arbres de recherche, celui proposé par Corneil et Gotlieb [28] est considéré comme l'un des premiers détaillés dans la littérature. Cet algorithme a été utilisé dans le domaine de la chimie pour détecter des similarités entre différentes formules chimiques. L'approche est basée sur un algorithme nommé « depth-first backtracking search ». Etant donné

deux graphes G_1 et G_2, les sommets de G_2 sont mis en correspondance avec les sommets de G_1, itérativement, sous la condition que la structure des arcs de G_2 soit préservée dans G_1 durant l'appariement des sommets. Donc, si le nombre de sommets dans les deux graphes est identique et si tous les sommets ont été mis en correspondance avec succès, il existe alors un isomorphisme entre ces deux graphes. La limite de cette méthode est liée au fait que la recherche dans le graphe est exhaustive (recherche par force brute). Donc, cet algorithme est très coûteux : $O(N^4)$ [22].

2.2.2 Algorithme d'Ullmann

C'est le plus populaire et l'un des algorithmes les plus efficaces pour la recherche d'isomorphisme de graphe, bien qu'il a été apparu en 1976. Ullmann a utilisé une procédure de raffinement qui consiste à élaguer le maximum de branches de l'arbre de recherche avec un parcours d'abord en profondeur. En fait, à chaque appariement (nœud de l'arbre), l'heuristique d'Ullmann filtre les appariements postérieurs afin d'exclure les appariements qui ne sont pas cohérents avec l'appariement en considération [5,17].

L'algorithme d'Ullmann est l'algorithme fondamental de la classe de recherche d'arbres pour l'isomorphisme de sous-graphes. A l'étape 1, l'algorithme obtient les correspondances possibles pour chaque sommet u en sélectionnant tous les sommets v dans le graphe de données. À l'étape 2, l'espace de recherche est globalement élagué en vérifiant que chaque sommet candidat v dans le graphe de données qui correspond au sommet u dans le graphe de requête doit avoir des enfants qui correspondent à tous les enfants de u. Ullmann utilise un algorithme itératif simple pour effectuer cet élagage. L'algorithme garantit qu'aucun sommet n'est utilisé pour correspondre à plus d'un sommet de requête. Ainsi, lorsque la procédure de recherche atteint la profondeur, en prenant en compte les conditions de la procédure d'élagage, l'isomorphisme est trouvé. L'algorithme effectue ensuite un retour arrière, jusqu'à ce que tous les chemins de recherche soient explorés. Ullmann suggère que les sommets des requêtes sont traités par ordre croissant, car cela permet de rencontrer et d'élaguer plus de branches au début du processus [1].

Par contre, la faiblesse de cet algorithme réside dans l'incapacité d'exploiter les attributs des nœuds et des arêtes dans le processus d'appariement

Plusieurs d'autres algorithmes qui se basent aussi sur des heuristiques pour optimiser le temps d'exploration dans l'arbre de recherche ont été proposés. Citons à titre d'exemple, l'algorithme VF et ses successeur VF2, VF++ et VF3.

2.2.3 Algorithmes VF/VF2

L'algorithme VF et VF2 sont une amélioration de l'algorithme d'Ullmann. La première étape de l'algorithme VF2 est la même que l'étape une de l'algorithme d'Ullmann. Cependant, Cordella et al. [29] ont modifié la démarche "prune down" par "build up", et donc la deuxième étape ne s'applique pas de la même manière que dans l'algorithme d'Ullmann. Au contraire, les chemins de recherche sont éliminés en tant que processus continu pendant la recherche. A partir de l'ensemble initial de mise en correspondances possibles, VF2 commence par ajouter une paire de nœuds correspondant à un ensemble de mise en correspondance partiel, avant d'effectuer une recherche approfondie dans le graphe de données et le graphe de requête en commençant par leurs noeuds. À chaque étape, il élimine les chemins de recherche en fonction des informations locales [1]. Cette heuristique qui utilise un parcours en profondeur associé à des règles de faisabilité [19] et dans le but de restreindre l'espace de recherche, a permis de réduire l'espace mémoire et de traiter des graphes de grandes tailles (milliers de nœuds).

2.3 Méthodes basées sur la décomposition

Cette approche vise à apparier un graphe et une base de graphes. Adapté par Messmer en 1995, cette méthode se base sur une décomposition récursive de chaque graphe de la base dans des sous graphes plus petits du fait que certaines parties sont communes à plusieurs graphes dans la base. Le processus d'appariement est exploité afin d'éviter la répétition de leur comparaison avec le graphe requête. De cette façon, le temps total d'appariement dépend du nombre de graphes de la base. Les auteurs de cet algorithme [30] ont proposé un autre algorithme plus performant pour l'isomorphisme entre un graphe et une base de graphes. En phase de prétraitement, un arbre de décision est construit à partir des graphes de la base. En utilisant cet arbre, un graphe d'entrée peut être comparé à toute la base et indépendamment du nombre de graphes dans la base [5].

Messmer & Bunke [31] ont proposé de décomposer les graphes et de construire un index basé sur cette décomposition. Par conséquent l'index contient des sous-graphes

de différentes tailles. L'idée principale est d'exploiter les sous-graphes qui sont communs à plusieurs graphes dans la base. La complexité diminue quand les similarités dans la base augmentent [7].

Dans le but d'accélérer le processus d'appariement par décomposition en utilisant les arbres de décision, plusieurs extensions de cet algorithme sont apparues. Nous citons à titre d'exemple les algorithmes présentés par Shearer et al. [32, 33] .

3 Appariement inexact

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.

Pour ces raisons, un nombre important d'algorithmes d'appariement inexact de graphes ont été proposés. Généralement, ces algorithmes n'imposent pas la contrainte de préservation de structure. Leur but est de déterminer un isomorphisme entre deux graphes de sorte que le coût global de l'appariement soit minimisé [13].

Dans la littérature, deux catégories d'algorithmes d'appariement inexact existent. La première englobe les algorithmes qui estiment le coût global d'appariement et restituent 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.

La deuxième catégorie est celle des algorithmes appelés approximatifs, elle cherche seulement à minimiser les coûts locaux (au niveau de chaque nœud) dans le processus d'appariement. Les résultats obtenus ne sont pas généralement très différents de ceux obtenus par les algorithmes optimaux. Leur inconvénient majeur est qu'ils ne garantissent pas de trouver la solution exacte si elle existe. Par contre, ils engendrent un gain du temps qui est généralement polynomial [5].

Comment définir le coût minimal d'appariement ?

Il existe différentes méthodes pour la définition du coût d'appariement. La majorité de ces méthodes sont basées sur :

- Un modèle explicite d'erreurs (déformations ou bruit) qui peuvent se produire (exemple nœuds manquants, etc.), en assignant un coût qui peut être différent

pour chaque type d'erreur. Ces algorithmes sont souvent désignés comme correcteur d'erreurs (error-correcting) ou tolérant aux erreurs (error-tolerant).

- Les autres façons de définir un coût d'appariement est la distance d'édition qui vise à introduire un coût pour chaque opérations d'édition de graphe (insertion de nœud par exemple, suppression de nœud, substitution de nœuds etc.), une fois qu'un coût est assigné à chaque opération, la succession la moins chère des opérations nécessaires pour transformer l'un des deux graphes en l'autre est calculée. Le coût de cette séquence est appelé le coût d'édition du graphe.

Dans la suite, nous présentons les techniques les plus connus de l'appariement inexact et qui sont regroupés en quatre catégories: l'arbre de recherche, l'optimisation continue, les méthodes spectrales et autres techniques.

3.1 Les techniques basées sur l'arbre de recherche

Les arbres de recherche peuvent aussi être utilisés pour l'appariement inexact de graphes. Dans ce cas, la recherche est dirigée par le coût de la mise en correspondance obtenu 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. Il existe beaucoup de méthodes pour distinguer le choix de la fonction coût et l'heuristique associée. Parmi celles-ci on retrouve assez souvent la notion de distance d'édition de graphes. Autour de cette idée, de nombreuses techniques sont proposées. Une autre distance de graphe utilisée pour estimer le coût est celle basée sur la mise en correspondance par graphe bipartie. Le principe est de représenter le calcul de distance sous la forme d'un graphe bipartie. On recherche alors la meilleure mise en correspondance entre deux jeux de sommets, avec la contrainte que chaque sommet ne peut être mis en correspondance qu'une et une seule fois. Ce problème peut être résolu à l'aide de l'algorithme $A*$.

3.2 Les techniques basées sur l'optimisation continue

Ces techniques ont pour objectif de transformer le problème d'appariement des graphes, qui est un problème d'optimisation discret, en un problème non linéaire ou en un problème d'optimisation continu. Cette idée donne des résultats satisfaisants dans plusieurs champs d'application, en raison de son coût de calcul extrêmement réduit, car elle dépend habituellement de la taille du graphe [15]. Premièrement, le problème de

correspondance de graphe est reformulé en tant que problème continu. Deuxièmement, le problème continu est résolu en utilisant un algorithme d'optimisation. Enfin, la solution trouvée est reconstruite au domaine discret initial.

Plusieurs algorithmes de mise en correspondance de graphes inexacts basés sur l'optimisation continue ont été proposés. Généralement, elles sont organisées en deux familles :

- la première famille se base sur l'étiquetage de relaxation probabiliste
- la deuxième sur le problème de mise en correspondance de graphe pondéré (WGM).

La première famille (l'étiquetage de relaxation probabiliste) est un processus itératif qui cherche à trouver une mise en correspondance entre les graphes en assignant une étiquette à chaque nœud ou sous-structure d'un graphe et ceci en fonction d'un certain nombre de contraintes.

La deuxième famille est basée sur une reformulation du problème sous la forme d'un problème de correspondance de graphe pondéré (WGM). Cette approche est basée sur l'utilisation d'une matrice de mapping M contenant des éléments de valeur réelle [0,1] afin de trouver une correspondance entre deux ensembles de nœuds ou sous-structures des deux graphes comparés. Une fonction objective définie qui dépend des poids des arêtes préservées par la correspondance doit être optimisée par l'appariement requis. Dû aux valeurs continues [0,1] des éléments de la matrice M, le problème des WGM est habituellement et naturellement transformé en un problème continu. L'un des désavantages importants de l'approche basée sur le WGM est que les poids des arêtes sont acceptés comme attributs et les nœuds ne peuvent pas avoir cette propriété. Plusieurs auteurs ont linéarisé et résolu ce problème en utilisant l'algorithme simplex. Autres auteurs ont proposé une approche basée sur le réseau de relaxation lagrangien pour l'appariement de graphes. Bien que pour approcher le problème d'isomorphisme de sous-graphe commun maximum, des auteurs ont proposé une méthode en produisant un graphe pondéré. Les poids obtenus indiquent la probabilité que le lien associé soit dans le sous-graphe commun maximal des deux graphes considérés [18].

D'autres méthodes d'appariement inexact de graphes basées sur l'optimisation continue ont été proposées. Parmi ceux-ci, nous pouvons citer la correspondance des graphes flous et les méthodes du noyau pour la mise en correspondance des graphes.

L'inconvénient de ces méthodes est qu'il n'y a aucune garantie pour atteindre un optimum local (solution sous-optimale). Cependant, l'approche basée sur l'optimisation

continue est très utile dans de nombreuses applications en raison de son coût de calcul très réduit qui est généralement polynomial [2].

3.3 Les techniques basées sur la théorie spectrale

Le premier qui a abordé l'appariement spectral est Umeyama [34] en 1988. Les méthodes basées sur la théorie spectrale des graphes cherchent 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. Leurs inconvénients majeurs [7,16] sont :

- Ces méthodes ne sont pas en mesure d'exploiter les attributs des nœuds et des arêtes, ce qui est souvent nécessaire dans le processus d'appariement.
- Elles sont sensibles aux erreurs structurelles, telles que les sommets manquants ou le bruit.
- Elles ne sont pas robustes pour des graphes de tailles différentes

Umeyama [34] a utilisé la décomposition en valeurs et vecteurs propres des matrices d'adjacence pour en déduire la matrice orthogonale qui va être optimisée par la suite. Pour cela il a supposé que les graphes sont, à priori, isomorphes. S'ils le sont, alors la méthode trouve la permutation optimale. Si les graphes sont proches de l'isomorphisme, alors la solution est sous-optimale. Toutefois, pour des graphes non-isomorphes la qualité des résultats diminue. La limitation majeure de l'approche d'Umeyama est l'impossibilité d'apparier des graphes de tailles différentes.

3.4. Autres techniques de mise en correspondance de graphes

Nous décrivons brièvement autres méthodes d'appariement de graphes et qui sont utilisées particulièrement dans le domaine de la reconnaissance des formes. Ces méthodes ne sont pas exactement considérées comme des formes d'appariement de graphes. Cependant, elles peuvent être associées à des méthodes de mise en correspondance de graphes soit parce qu'ils présentent une manière de comparer deux graphes ou parce qu'elles utilisent des graphes pour classer des objets, tels que les

méthodes basées sur les noyaux des graphes, le clustering de graphes et l'apprentissage graphique [18]. Ces approches présentent un moyen pour combiner les statistiques et les techniques structurelles dans la reconnaissance de formes. Pour plus de détail sur ces deux techniques, le lecteur peut se référer à [20] et [21].

3.4.1 Les méthodes basées sur le clustering

Plusieurs algorithmes ont été apparus et qui ont été basés sur l'idée proposée par Umeyama [34] en combinaison avec un regroupement (clustering).

Citons à titre d'exemple, l'idée de Carcassoni & Hancock [36] qui consiste à utiliser le clustering des sommets avant l'appariement pour permettre d'associer d'abord les clusters et ensuite les sommets. Une autre idée de Caelli & Kosinov [35] qui propose de projeter les sommets dans l'espace propre des graphes et d'utiliser le clustering dans cet espace afin de trouver les sommets à mettre en correspondance.

3.4.2 Les méthodes basées sur les noyaux de graphes

Le principe des méthodes à noyau est de transférer le problème de reconnaissance de formes vers un espace vectoriel au lieu d'utiliser l'espace des objets d'origine. Pour les graphes, au lieu de définir des outils mathématiques dans l'espace des graphes, tous les graphes sont projetés dans un espace vectoriel où plusieurs outils de calcul de distance performants sont disponibles [17].

Les noyaux de graphes représentent également une façon de quantifier la similarité entre graphes. L'idée fondamentale des noyaux est une transformation de l'espace des données X, dans un espace nommé de caractéristiques F (*feature space*). Cet espace représente mieux les données mais il est généralement d'une dimension élevée. Les algorithmes de noyaux se basent sur le calcul des produits scalaires entre les éléments de X. La fonction noyau permet de calculer le produit scalaire dans l'espace F en utilisant la représentation des données dans l'espace X.

L'utilisation des méthodes des noyaux de graphes permet de bénéficier à la fois du pouvoir de représentation des graphes et du grand nombre d'algorithmes vectoriels. Par conséquent, les méthodes du noyau sont plus appropriées pour les tâches difficiles de reconnaissance de formes que les méthodes traditionnelles dans certaines conditions [18]. La difficulté principale de ces approches réside dans la définition de la fonction noyau appropriée aux données qui d'une part, représente bien les données et, d'autre part, permet le calcul efficace des produits scalaires.

Il a été prouvé que le calcul d'un noyau de graphe qui utilise toute l'information disponible est NP-difficile. Ceci implique des limitations des approches par noyaux. Si l'on souhaite conserver toute la généralité du modèle de graphe, alors les noyaux ne sont pratiquement pas calculables. Afin de réduire le temps de calcul, il faut accepter la perte d'information en projetant les graphes dans un espace moins complexe et souvent moins discriminant [7].

Plusieurs travaux ont été réalisés dans la littérature, parmi lesquels on cite les travaux de [37 et 38] et aussi les noyaux basés sur la distance d'édition [39].

3.4.3 Les méthodes 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 poids associés à la correction d'erreurs aux niveaux des attributs et de la structure du graphe [7].

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 [40]. 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 [41] 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 [5].

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 [7].

De plus, les algorithmes génétiques sont également utilisés pour apparier les graphes où le problème est défini sous forme d'un ensemble d'états (i.e les

appariements) avec des degrés d'adaptation (fitness). Parmi les travaux qui se sont penché sur ces algorithmes, nous pouvons citer [42] où un algorithme micro-génétique est appliqué pour résoudre des AGP problème, Wang et al. [43] ont proposé un algorithme génétique pour trouver l'erreurs-correction. Un algorithme génétique intéressant est aussi présenté dans Khoo et al. dans [44] pour trouver le SCM entre deux graphes [5].

4 Conclusion

Dans ce chapitre, on a établi un état de l'art sur les méthodes d'appariement de graphes qui existent dans la littérature. Ces méthodes se répartissent en deux catégories: l'appariement exact et l'appariement approximatif. Dans la première catégorie, l'objectif est de tester si deux graphes sont isomorphes en vérifiant leur identité en terme de structures et d'étiquettes. Par ailleurs, dans la catégorie d'appariement approximatif, l'objectif est de déterminer une distance entre deux graphes qui peuvent être structurellement différents. Le premier constat de cette étude est la complexité élevée de l'appariement de graphes. Nous avons constaté aussi que l'appariement approximatif de graphes est plus adapté que l'exact dans différentes applications. Néanmoins, les méthodes d'appariement approximatif existantes ont des limites différentes. Principalement, nous en avons distingué deux : la première limite concerne la nécessité de prédéfinir les coûts des opérations d'édition pour certaines méthodes. La deuxième limite concerne les méthodes basées sur la théorie spectrale où les graphes sont contraints de ne contenir que des étiquettes numériques simples.

Dans le chapitre suivant, nous détaillerons deux algorithmes les plus connus dans la littérature puis dans un second chapitre nous allons les implémenté et faire une comparaison du point de vue complexité spatiale et temporelle.

Chapitre 3 :

L'algorithme d'Ullmann et l'algorithme VF2

1. Introduction

Dans le chapitre précédent, un état de l'art sur les différentes approches d'appariement de (sous-)graphes a été présenté. Ces approches se répartissent en deux grands axes : les algorithmes d'appariement exact qui se concentre sur la recherche d'un isomorphisme entre des graphes qui ont la même topologie, et les algorithmes d'appariement inexact qui se limite généralement à calculer une distance entre des graphes avec des structures différentes et qui admettent même des variations des attributs.

Dans ce chapitre, nous allons détailler deux algorithmes que nous avons vu dans l'état de l'art. Nous nous concentrons sur les algorithmes basés sur l'arbre de recherche qui ont connu un grand succès pour résoudre le problème d'isomorphisme de (sous-) graphes qui est considéré par plusieurs auteurs comme un problème NP-complet.

Notre étude vise à présenter et implémenter quelques algorithmes d'isomorphisme de (sous-)graphes pour permettre de comparer leur complexité.

2 Algorithme d'Ullmann

L'algorithme d'Ullmann [52] présenté dans la figure 3.1 selon la description de [51], est considéré comme l'un des algorithmes les plus rapides pour résoudre le problème d'isomorphisme de (sous-)graphe. Cette méthode est basée sur le backtracking et une procédure de raffinement. L'algorithme est conçu à la fois pour l'isomorphisme des graphes et de sous-graphes. Ullmann décrit d'abord un algorithme d'énumération simple pour l'isomorphisme en utilisant une première recherche d'arborescence en profondeur. Il présente ensuite une procédure de raffinement pour réduire l'espace de recherche et l'incorpore dans l'algorithme. L'algorithme d'énumération simple fonctionne comme décrit ci-dessus. La procédure de raffinement d'Ullmann est ensuite appelée. Le principe de cette procédure est d'éliminer certains des "1" dans les matrices d'adjacence M, pour éliminer ainsi certains nœuds successeurs de la recherche arborescente. Il teste chaque "1" dans M avec la condition suivante:

$$\forall_i (1 \preccurlyeq i \preccurlyeq p_a), \forall_j (1 \preccurlyeq j \preccurlyeq p_b) : a_{ij} = 1 \Rightarrow c_{ij} = 1 \text{ .... (1)}$$

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 … |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.**

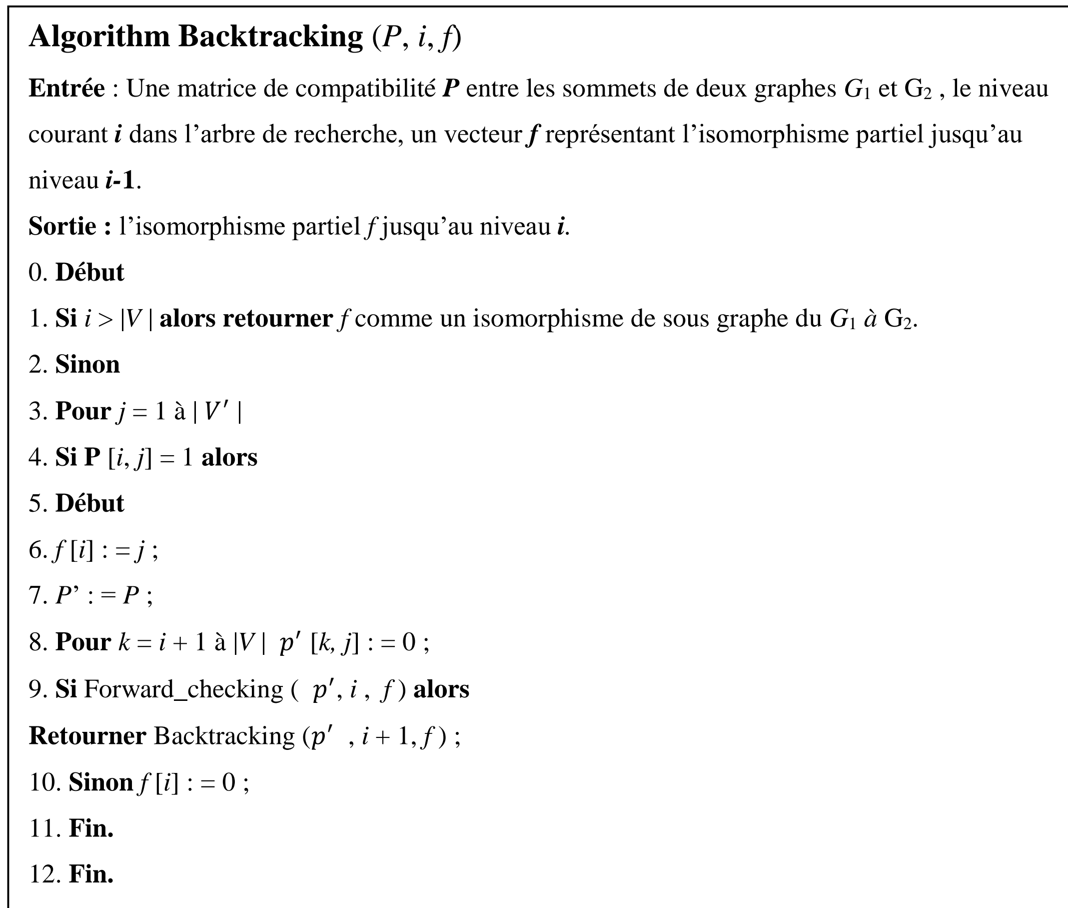
Figure 2.1 Algorithme d'Ullmann

Pour tout voisin x du nœud i, il doit exister un nœud y de G_1 tel que y soit voisin du nœud j et que x puisse correspondre à y. Il change le "1" en "0" si cette condition n'est pas satisfaite. Il itère jusqu'à ce qu'aucun "1" n'est changé en "0". Si M satisfait la condition d'être une matrice M^0 (c'est-à-dire, chaque rangée de M contient exactement "1" et chaque colonne de M ne contient plus d'un "1"), alors si la procédure de raffinement ne modifie pas M, M spécifie un isomorphisme entre (sous-)graphes. Cette procédure de raffinement est ensuite introduite dans l'algorithme simple d'énumération en profondeur [53].

L'algorithme d'Ullmann est basé sur la procédure récursive de backtracking décrite par [51] dans la figure 3.2 en combinaison avec une procédure de raffinement de l'espace de recherche (la procedure forward checking) présentée par le même auteur et décrite dans la figure 3.3.

L'algorithme d'Ullmann cherche à trouver tous les isomorphismes de (sous-) graphes possibles du graphe G_1 à partir du graphe G_2. Pour réduire le nombre

d'appariements à tester, une approche d'appariement incrémentale des nœuds a été proposée.



Algorithm Backtracking (P, i, f)

Entrée : Une matrice de compatibilité ***P*** entre les sommets de deux graphes G_1 et G_2 , le niveau courant ***i*** dans l'arbre de recherche, un vecteur ***f*** représentant l'isomorphisme partiel jusqu'au niveau ***i*-1**.

Sortie : l'isomorphisme partiel *f* jusqu'au niveau ***i***.

0. **Début**
1. **Si** $i > |V|$ **alors retourner** *f* comme un isomorphisme de sous graphe du G_1 *à* G_2.
2. **Sinon**
3. **Pour** $j = 1$ à $| V' |$
4. **Si P** $[i, j] = 1$ **alors**
5. **Début**
6. $f[i] := j$;
7. $P' := P$;
8. **Pour** $k = i + 1$ à $|V|$ $p' [k, j] := 0$;
9. **Si** Forward_checking (p', i , f) **alors**

Retourner Backtracking (p' , $i + 1, f$) ;

10. **Sinon** $f[i] := 0$;
11. **Fin.**
12. **Fin.**

Figure 3.2 Algorithme de backtracking

A chaque étape, un couple de nœuds est ajouté dans l'appariement courant et la condition d'isomorphisme de (sous-)graphes est testée. Si, pour un certain couple de nœuds, l'isomorphisme n'était pas trouvé, l'algorithme fait un retour en arrière «backtrack» et le nœud du graphe modèle utilisé précédemment va être mis en correspondance avec un autre nœud afin de tester une nouvelle condition d'isomorphisme. L'espace de recherche se réduit très efficacement grâce à la technique de « forward checking ». Elle a pour but de détecter par avance des instanciations inconsistantes en appliquant un critère de test de consistance sur les arcs pendant la recherche [51].

Algorithm Forward checking (P, i, f)

Entrée : Une matrice de compatibilité P entre les sommets de deux graphes G_1 et G_2, le niveau courant i dans l'arbre de recherche, un vecteur f représentant l'isomorphisme partiel jusqu'au niveau i-1.

Sortie : VRAI si pour chaque sommet $V_k \in V(k > i)$ il existe au moins un appariement compatible avec appariements des sommets existants présents dans *f*, FAUX, sinon.

0. **Début**
1. **Pour** $k = i + 1$ à $|V|$
2. **Pour** $j = 1$ à $|V'|$
3. **Si P** $[k, j] = 1$ **alors**
4. **Pour** $l = 1$ à i -1
5. **Début**
6. vérifier les contraintes suivantes de consistance des arcs :

   (1) Pour chaque arc e = $(vk, vl) \in E$ existe est-il un arc $e' = (wj, wf[l]) \in E'$ tel que $LE(e) = L'_E(e')$

   (2) pour chaque arc $e' = (wj, wf[l]) \in E'$ existe est-il un arc $e = (vk, vl) \in E$ tel que $LE(e) = L'_E(e')$
7. **Si** (1) ou (2) sont FAUX **alors** $P[k, j] := 0$; E
8. **Fin**
9. **Si** il existe une ligne **k** *dans* **P** *telle* que **P [*k, j*] = 0** pour $j = 1 \ldots |V'|$ **alors retourner** FAUX ;
10. **Sinon retourner** VRAI ;
11. **Fin**.

Figure 3.3 Algorithme de Forward checking

2.1 Algorithme d'énumération

L'idée de base de l'algorithme d'Ullmann est l'énumération. Premièrement, on introduit un algorithme d'énumération désigné pour trouver tous les isomorphismes entre un graphe donné $G_1 = (V_\alpha, E_\alpha)$ et les sous-graphes d'un autre graphe $G_2 = (V_\beta, E_\beta)$. Les nombres de noeuds et des arêtes de G_1 et G_2 sont p_α, q_α et p_β, q_β, respectivement. Les matrices d'adjacence de G_1 et G_2 sont $A = [a_{ij}]$ et $B = [b_{ij}]$, respectivement.

Soit M', une matrice de $p_\alpha * p_\beta$ dont les éléments sont des "1" et des "0" tels que chaque ligne contienne exactement un "1" et aucune colonne n'en contienne plus d'un "1". La

matrice M' peut être utilisée pour permuter les rangées et les colonnes de B pour produire une matrice supplémentaire C.

C $[c_{ij}]$ = M' * B * $(M')^T$ où T indique la transposée.

Si :

$\forall_i(1 \leqslant i \leqslant p_a), \forall_j(1 \leqslant j \leqslant p_b) : a_{ij} = 1 \mapsto c_{ij} = 1$ .... (1)

est vrai, alors M' spécifie un isomorphisme entre G_1 et un sous-graphe de G_2. Dans ce cas, si $m'_{ij} = 1$, alors le ième nœud de G_1 correspond au ième noeud de G_2.

Au début de l'algorithme, on construit une matrice $p_\alpha * p_\beta$, $M^0 = [m^0_{ij}]$ avec :

${m^0}_{ij}$ = 1 si le degré du j^{me} nœud de $G_1 \geqslant$ le degré du i^{me} noeud de G_2 , 0 sinon.

L'algorithme d'énumération génère toutes les matrices possibles M' telles que pour chaque élément de M' $(m'_{ji} = 1) \mapsto (m^0_{ji} = 1)$. Pour chaque matrice M', l'algorithme teste l'isomorphisme en appliquant la condition (1). Les matrices M' sont générées en changeant systématiquement en "0" toutes les rangées de $M°$ en ne laissant qu'un seul "1", et en vérifiant qu'il n'y a pas plus d'un "1" dans chaque colonne de la matrice M'.

Dans l'arbre de recherche, les nœuds terminaux sont en profondeur $d = p_\alpha$, où d est le degré et ils correspondent à des matrices distinctes M'. Chaque nœud non-terminal $d < p_\alpha$ correspond à une matrice distincte M qui diffère de M^0 de telle sorte que d rangées de cette matrice ont déjà été traitées (ne contiennent donc plus qu'un et un seul "1" par rangée). Cet algorithme d'énumération peut être amélioré en introduisant une procédure de raffinement pour définir les nœuds successeurs dans la recherche.

2.2 Algorithme employant la procédure de raffinement

Pour réduire la quantité de calcul nécessaire pour trouver des isomorphismes de sous-graphes, nous utilisons une procédure, que nous appelons la procédure de raffinement [52], qui élimine certains des "1" des matrices M, éliminant ainsi les nœuds successeurs de la recherche arborescente. Pour introduire la procédure de raffinement, considérons la matrice M associée avec n'importe quel nœud non-terminal donné dans l'arbre de recherche. Tout isomorphisme de sous-graphes correspond à une matrice particulière M'. On dit qu'un isomorphisme est un isomorphisme sous M si son nœud terminal dans l'arbre de recherche est un successeur du nœud auquel M est associé. Les "0" dans la matrice M excluent simplement les correspondances entre les points de V_α et V_β. Si $m`_{ij} = 0$ pour tous les correspondances sous M, alors si $m_{ij} = 1$ on peut changer $m_{ij} = 1$ avec $m_{ij} = 0$ sans perdre aucun des correspondances sous M: toutes ces correspondances peuvent être trouvées par la recherche arborescente. La condition ci-

dessous doit être nécessairement satisfaite. Si $m_{ij} = 1$ pour tous les isomorphismes sous M. Si cette condition n'est pas satisfaite et $m_{ij} = 1$, la procédure de raffinement change $m_{ij}= 1$ avec $m_{ij} = 0$ [52] .

Soit v_α, le ième point dans V_M, et soit v_β le troisième point de V_β et soit $\{v_{\alpha 1},\ldots,v_{\alpha x},\ldots,v_{\alpha y}\}$ l'ensemble de tous les points de $G2$ qui sont adjacents à $v_{\alpha i}$ dans G_2. A partir de la définition de l'isomorphisme du sous-graphe, il est nécessaire que si v_α correspond à v_β dans l'isomorphisme, alors pour chaque $x = 1, \ldots y$, on considère la matrice M' associée à tout isomorphisme donné. Il doit exister un point $v_{\beta y}$ dans V_β adjacent à $v_{\beta j}$, tel que $v_{\beta y}$ correspond à $v_{\beta x}$ dans l'isomorphisme. Si $v_{\beta y}$ correspond à $v_{\beta x}$ dans l'isomorphisme, alors l'élément de M' qui correspond à $\{v_{\alpha x} , v_{\alpha y}\}$ est égal à 1.

Donc si $v_{\alpha j}$ correspond à $v_{\beta j}$ dans tout isomorphisme sous M, alors pour chaque x=1,…y, il doit y avoir "1" dans M correspondant à $\{v_{\alpha x} , v_{\alpha y}\}$ tel que $v_{\beta y}$ est adjacent à $v_{\beta j}$. En d'autres termes, si $v_{\alpha i}$ correspond à $v_{\beta j}$ dans tout isomorphisme sous M, et :

$$(\forall \boldsymbol{x}) \quad (a_{ix = 1}) \Rightarrow \ \exists\ (y_j)\ (m_{xy} . b_{yj} = 1) \ \ldots\ldots\ldots\ldots \ (2)$$

$$x \leqslant 1 \leqslant p_\alpha \qquad y \leqslant 1 \leqslant p_\beta$$

La procédure de raffinement teste simplement chaque "1" dans M pour déterminer si la condition (2) est satisfaite. Pour tout $m_{ij} = 1$ tel que (2) n'est pas satisfaite, $m_{ij} = 1$ est changé en $m_{ij} = 0$. Ces changements peuvent rendre la condition (2) non satisfaite pour d'autres A dans M, de tel sorte que d'autres changements peuvent être faits, et ainsi de suite. En fait, la procédure de raffinement applique successivement la condition (2) à chaque i de M, jusqu'à ce qu'il y ait une itération dans laquelle tous les "1" de M sont traités et aucun d'eux n'est changé en "0". Notez qu'il n'y a aucune restriction sur l'ordre dans lequel les "1" dans M doivent être traités. Généralement, le résultat de la procédure de raffinement est de changer certains des "1" dans *M en* "0". Cependant, la procédure de raffinement peut laisser M inchangé, ce qui est particulièrement important lorsque M est une matrice M'. Une condition nécessaire et suffisante pour l'isomorphisme de sous-graphe est que la procédure de raffinement laisse M' inchangée. Ceci est dû au fait que si M' est inchangée par la procédure de raffinement, alors (2) est valable pour chaque "1" dans M'. Donc M' spécifie un mapping 1:1 de V_α, dans V_β tel que si deux points sont adjacents dans G_α, alors les deux points correspondants dans G_β sont adjacents. La procédure de raffinement peut donc être utiliser comme test pour l'isomorphisme de sous-graphe au lieu d'utiliser la condition (1) : si la procédure de raffinement consiste que chaque "1" dans M' est remplacé par "0", alors M' ne spécifie pas un isomorphisme.

Au cours de la procédure de raffinement, on doit vérifier continuellement si une rangée de M ne contient aucun "1". Si une rangée de M ne contient aucun "1", la procédure converge vers sa sortie échec, car il n'y a aucun avantage à poursuivre la procédure. Sinon, la procédure se termine à sa sortie succès.

2.3 Exemple illustratif du fonctionnement de l'algorithme d'Ullmann

Soient deux graphes de la figure 3.4 : $G_1 = (V_B,E_B)$ et $G_2 = (V_A,E_A)$, avec leur matrices d'adjacences.

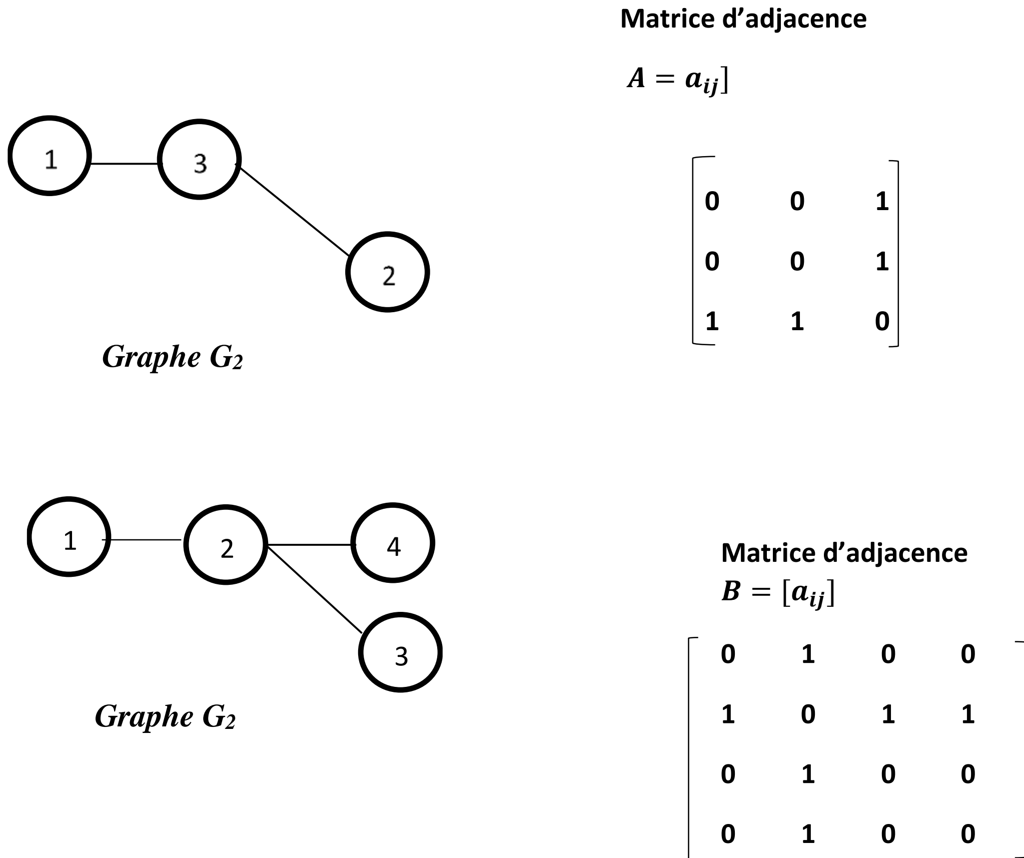


Figure 3.4 Deux graphes G_1 et G_2 avec leurs matrices d'adjacence.

Matrice de permutation :

La matrice de permutation M'(n × m) est une matrice de taille $p_a * p_b$ qui ne doit contenir que des '0' et des '1' avec :

- Exactement un seul '1' dans chaque ligne.
- Pas plus d'un '1' dans chaque colonne.

Les nœuds qui sont en correspondance des deux graphes G_1 et G_2 sont : (1,1), (2,3), (3,2) comme illustre les lignes pointillées de la figure 3.5 (a).

1- Calculer la matrice C= $M'(M'B)^T$ par la formule :

$\forall_i (1 \leqslant i \leqslant p_a), \forall_j (1 \leqslant j \leqslant p_b) : a_{ij} = 1 \Rightarrow c_{ij} = 1$

- **Calculer $M'B$** : Décaler la ligne j vers la ligne i, $M'_{ij} = 1$
- Permuter la matrice d'adjacence B en multipliant avec M'

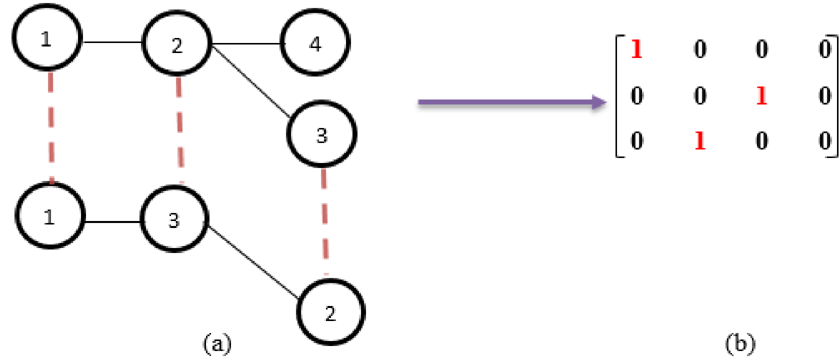


Figure 3.5 (a) représente les nœuds mis en correspondance des deux graphes G_1 et G_2.

(b) représente la matrice M' de taille 3x4

- **Calculer $(M'B)^T$** : Décaler la colonne j vers la colonne i
- **Calculer $C = M'(M'B)^T$** : Décaler la colonne j vers la colonne i et Décaler la ligne j vers la ligne i .

$$\underset{M'}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}} \times \left(\underset{M'}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}} \times \underset{B}{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}} \right)^T$$

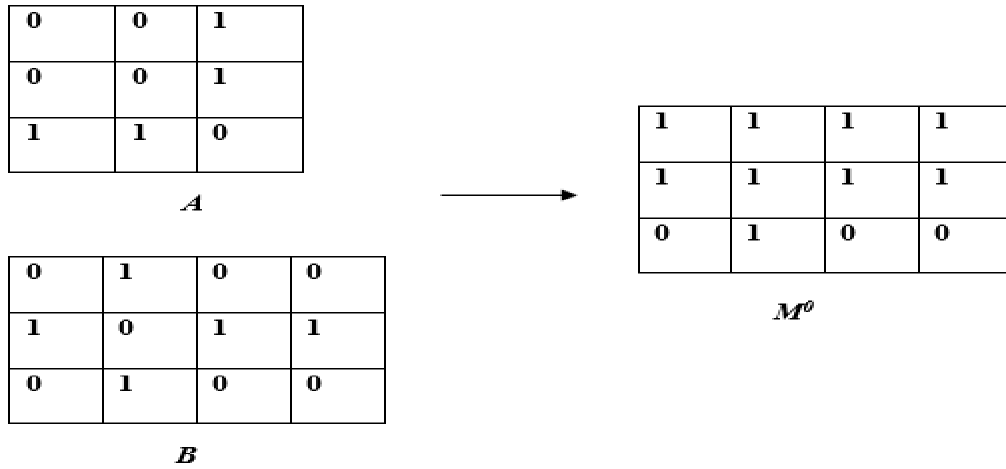
$$= \underset{M'}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}} \times \underset{(M' x B)^T}{\begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} = \mathbf{C}$$

Si la matrice C est égale à la matrice A, alors le graphe G_1 est isomorphe à G_2.

Calcul de la matrice M^0

Une matrice $M^0\ (p_a * p_b) = [m^0{}_{ij}]$ est définie par la formule suivante :

$$m^0{}_{ij} = \begin{cases} 1 \text{ si le degré du } G_B \geqslant \text{ le degré de } G_A \\ 0 \text{ sinon.} \end{cases}$$



0	0	1
0	0	1
1	1	0

A

0	1	0	0
1	0	1	1
0	1	0	0

B

1	1	1	1
1	1	1	1
0	1	0	0

M^0

Puis on génère systématiquement les matrices de permutation M^d comme le montre les figures : 3.6 et 3.7.

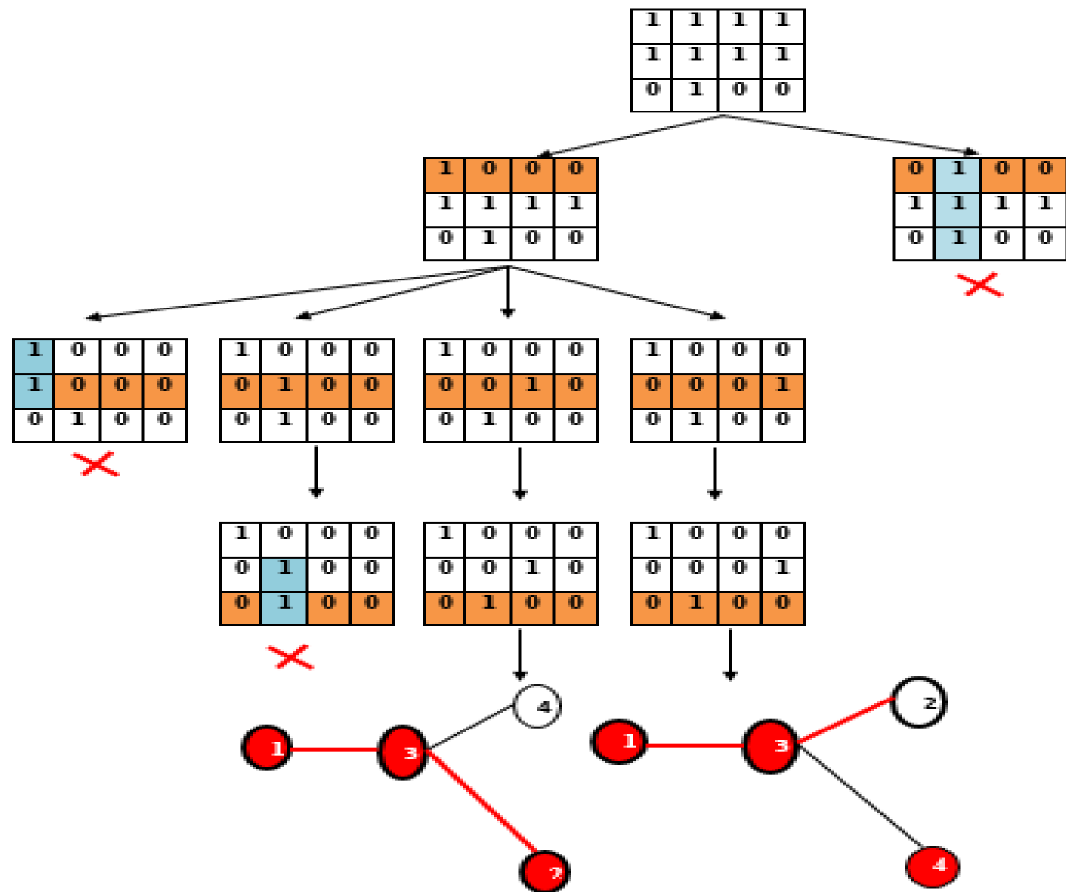


Figure 3.6 Matrices M^d générées

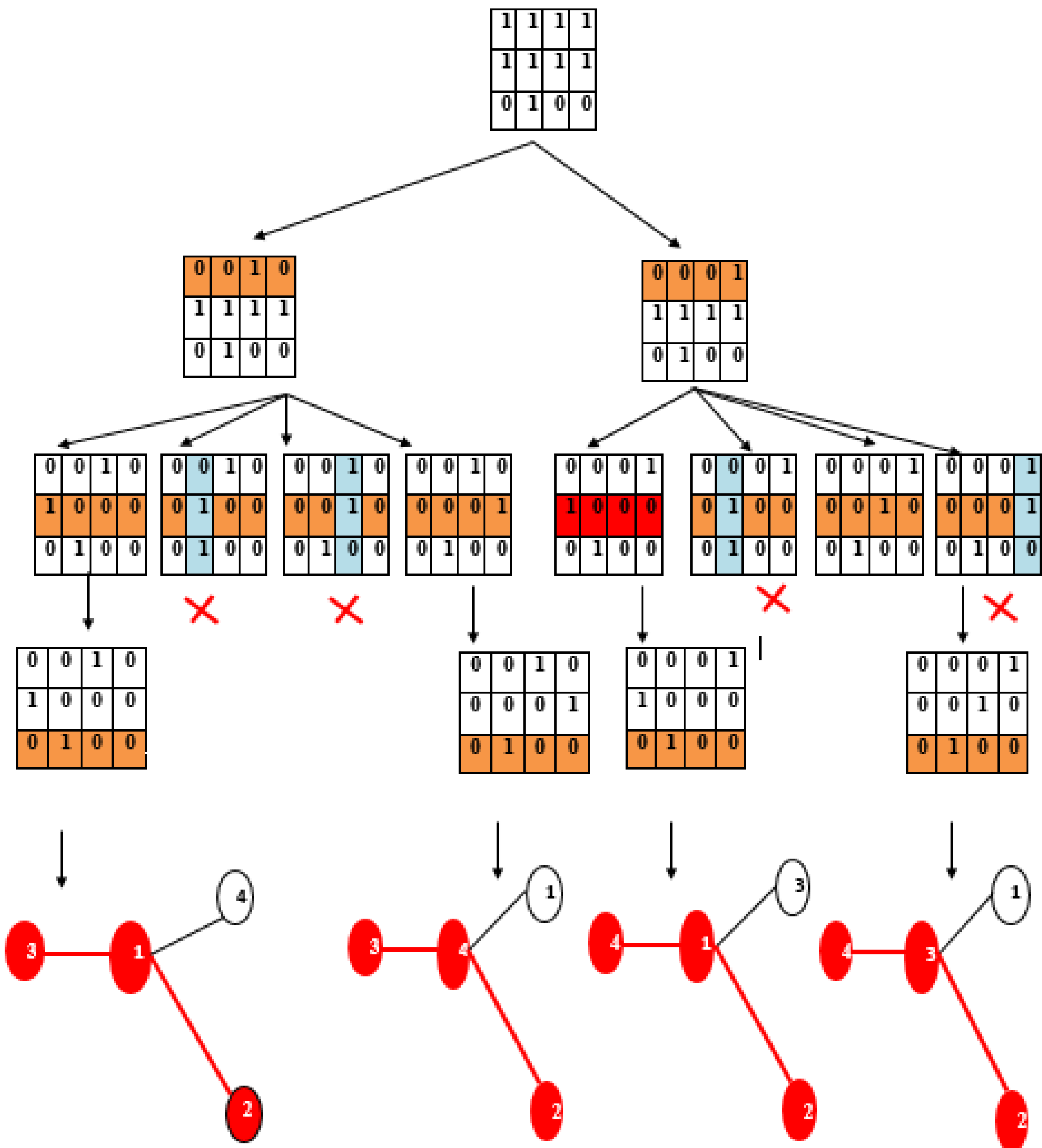


Figure 3.7 Les isomorphismes trouvés.

3 Algorithmes VF et VF2

L'algorithme VF [46] de cordelle et al. prend son nom de l'abréviation anglaise « Very Fast algorithm » par ce qu'il est considéré par plusieurs auteurs comme étant l'algorithme le plus efficace et le plus performant pour résoudre les problèmes d'isomorphisme de graphes et des sous graphes. Cet algorithme a montré une amélioration significative par rapport à l'algorithme d'Ullman grâce à une heuristique basée sur une analyse de l'ensemble des nœuds adjacents aux nœuds déjà présents dans

l'appariement courant. L'algorithme VF2 présenté dans [54 et 55] est une mise à jour de l'algorithme VF [46] des mêmes auteurs. Cet algorithme a réduit l'espace mémoire nécessaire ce qui entraîne une réduction de complexité de l'ordre $O(N^2)$ à $O(N)$ par rapport au nombre de nœuds dans les graphes. La principale différence entre les deux versions de VF est la structure de données appliquée, mais la génération des paires de nœuds candidats reste la même. Il possède plusieurs propriétés, citons par exemple qu'il est appliqué sur n'importe quel type de graphes et n'impose aucune restriction sur la topologie des graphes à apparier. Il est appliqué aussi pour les graphes de grande taille (plus de 1000 nœuds) à cause de sa consommation mémoire réduite. Un autre avantage de cet algorithme est qu'il exploite les informations sémantiques des graphes lorsqu'elles sont disponibles [45].

VF2 est un algorithme très souple capable de traiter différents problèmes d'appariement de graphes. Il est basé sur une représentation d'espace d'état (SSR : Space States Representation) où chaque état est un mapping partiel entre deux graphes donnés. Seuls les états qui satisferont les contraintes imposées sont considérés comme des états but. Par conséquent, l'objectif de cet algorithme est de partir d'un appariement vide, puis d'explorer l'espace d'état en augmentant de façon itérative le nombre de paires de nœuds impliqués par le mapping jusqu'à atteindre un état de but, s'il existe.

3.1 Présentation de l'algorithme VF2

Etant donné deux graphes $G_1 = (N_1, B_1)$ et $G_2 = (N_2, B_2)$, on dit qu'un mapping $M \sqsubset N_1 X N_2$ est un isomorphisme si M est une fonction bijective qui préserve la structure des arcs des deux graphes, c'est-à-dire, M mis en correspondance chaque nœud de G_1 avec un nœud de G_2 et vice versa. M est un isomorphisme de sous-graphe si et seulement si M est un isomorphisme entre G_2 et un sous-graphe de G_1. Notons que dans l'article présenté par [46], les graphes impliqués sont des graphes orientés.

Notations

Avant de présenter l'algorithme VF2, nous citons les notations utilisées dans l'algorithme.

- $G_1 = (N_1, B_1)$, $G_2 = (N_2, B_2)$ les deux graphes,
- état s,
- $M(s)$ appariement partiel associé à s,
- $M_1(s)$ nœud de M(s) dans N_1,
- $M_2(s)$ nœud de M(s) dans N_2,

- $P(s)$ ensemble des paires (dans N_1 x N_2) candidats pour être ajouter à s.
- $F(s,n,m)$ fonction booléenne (si l'ajout de (n,m) dans l'état s définit un isomorphisme partiel ou non ?)
- $T_1^{out}(s)$, ensemble des nœuds de G_1 (successeur) d'un nœud de $M_1(s)$.
- $T_2^{out}(s)$, ensemble des nœuds de G_2 (successeur) d'un nœud de $M_2(s)$.
- $T_1^{in}(s)$ ensemble des nœuds de G_1 (prédécesseur) d'un nœud de $M_1(s)$.
- $T_2^{in}(s)$ ensemble des nœuds de G_2 (prédécesseur) d'un nœud de $M_2(s)$.

2.3. Description de l'algorithme VF2

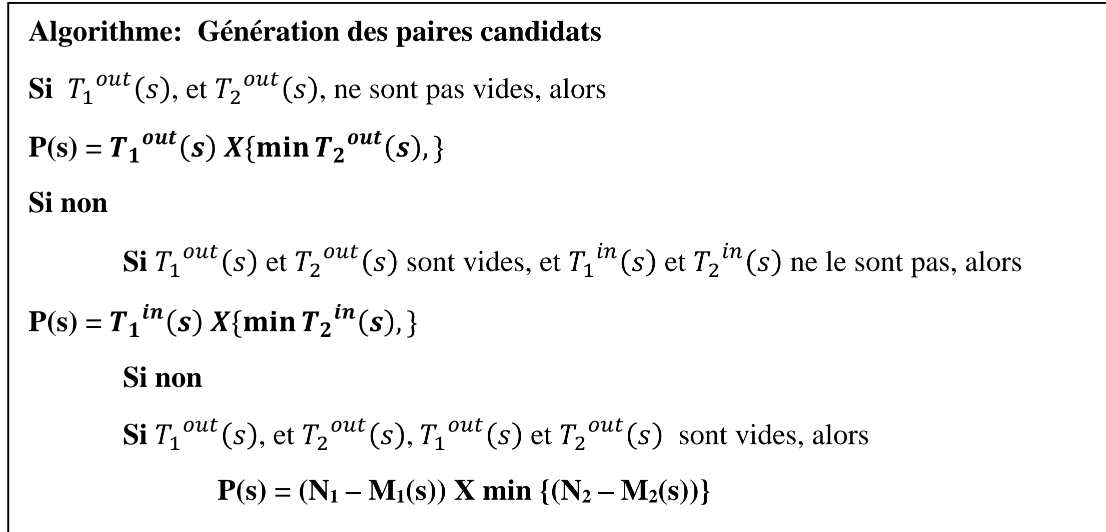
L'algorithme VF2, présenté dans la figure 3.8, commence à l'état s_0 par un appariement vide ($M(s_0)=\varnothing$). Le processus d'appariement peut être décrit de manière appropriée au moyen d'une représentation spatiale d'état (SSR).

```
Procédure Match(S)
Début
INPUT: un état intermédiaire s; l'état initial s0 possède M(s0) = ∅
OUTPUT: les mappings entre les deux graphes
IF
        M(s) couvre tous les nœuds de G2
THEN
        OUTPUT M(s)
ELSE
        Calculer l'ensemble P(s) des paires candidats pour être ajouter à M(s)
        FOREACH (n, m) ∈P(s)
                IF F(s, n, m)
                THEN
                        Calculer l'état s´ obtenu par l'ajout de (n, m) dans M(s)
                        Appeler  Match(s')
                END IF
        END FOREACH
Restaurer les structures de données
END IF
End Procédure
```

Figure 3.8 Algorithme d'appariement VF2

Chaque état s du processus de mise en correspondance peut être associé à une solution de mapping partiel $M(s)$, qui ne contient qu'un sous-ensemble de M. Une solution de mapping partiel identifie deux sous-graphes de G_1 et G_2, disons $G_1(s)$ et $G_2(s)$, obtenus en ne sélectionnant que les nœuds de G_1 ou G_2 inclus dans les composants de $M(s)$, et les arcs les reliant. A chaque état, l'algorithme génère un ensemble de paires de nœuds candidats, notées $P(s)$. Ces paires de nœuds doivent satisfaire les règles de

faisabilité pour être ajoutées au mapping partiel *M(s)*. L'algorithme qui construit *P(s)* est montré dans la figure 3.9. Dans ce contexte, min se réfère au nœud dans $T_2^{out}(s)$ qui a le plus petit label. Selon [46], n'importe quel critère d'ordre peut être utilisé.



Algorithme: Génération des paires candidats

Si $T_1^{out}(s)$, et $T_2^{out}(s)$, ne sont pas vides, alors

P(s) = $\boldsymbol{T_1^{out}(s)\ X\{\min T_2^{out}(s),\}}$

Si non

Si $T_1^{out}(s)$ et $T_2^{out}(s)$ sont vides, et $T_1^{in}(s)$ et $T_2^{in}(s)$ ne le sont pas, alors

P(s) = $\boldsymbol{T_1^{in}(s)\ X\{\min T_2^{in}(s),\}}$

Si non

Si $T_1^{out}(s)$, et $T_2^{out}(s)$, $T_1^{out}(s)$ et $T_2^{out}(s)$ sont vides, alors

P(s) = (N_1 – M_1(s)) X min {(N_2 – M_2(s))}

Figure 3.9 Algorithme calcul de P(s)

Dans le cas où un seul des ensembles $T_1^{in}(s), T_2^{in}(s)$ou un seul des ensembles $T_1^{out}(s), T_2^{out}(s)$ est vide, alors les états ne peuvent pas conduire à un appariement, et l'arbre ne peut pas être exploré davantage. Chose qui garantit que l'algorithme de recherche ne visite jamais deux fois le même état.

2.4. Définition de la fonction de faisabilité

La fonction de faisabilité *F(s, n, m)* est une fonction booléenne qui est utilisée pour réduire l'espace d'exploration et pour élaguer l'arbre de recherche [50]. Si sa valeur est vraie, alors l'état obtenu en ajoutant une paire de nœud candidat (n, m) à *s* garantit de trouver un isomorphisme partiel, par conséquent, l'état final est soit un isomorphisme entre G_1 et G_2, soit un isomorphisme entre un sous-graphe de G_1 et G_2. De plus, *F* va aussi élaguer certains états, qui, malgré qu'ils correspondent à un isomorphisme entre G_1(s) et G_2(s), ne conduisent pas à une solution d'appariement complète. Afin d'évaluer *F(s,n,m)*, l'algorithme examine tous les nœuds connectés (adjacents) à n et m. Si ces nœuds sont dans le mapping partiel courant (c'est-à-dire dans $M_1(s)$ et $M_2(s)$), l'algorithme vérifie si chaque arc de, ou vers n, a un arc correspondant à un arc de, ou vers m et vice versa. Si les nœuds et les arcs des graphes correspondent à des attributs sémantiques, une autre condition doit également être vérifiée pour que

F(s,n,m) soit vraie, c'est-à-dire que les attributs des nœuds et des arcs à mettre en correspondance doivent être compatibles. Dans le cas le plus strict, l'égalité d'attribut est nécessaire. La fonction de faisabilité *F* est définit par la formule suivante :

$$\boldsymbol{F(s,n,m) = F_{syn}(s,n,m) \wedge F_{sem}(s,n,m)}$$

Où $\boldsymbol{F_{syn}(s,n,m)}$ sont les règles syntaxiques qui dépendent de la structure des graphes et $\boldsymbol{F_{sem}(s,n,m)}$ est la règle sémantique qui dépend des attributs des nœuds et des arcs.

Règles syntaxiques

Pour définir les cinq règles syntaxiques, nous devons introduire les notations supplémentaires suivantes :

- **Pred(G,n)** et **Succ (G,n)** dénotent, respectivement, l'ensemble des prédécesseurs et l'ensemble des successeurs de n.
- **Pred(G,m)** et **Succ (G,m)** dénotent, respectivement, l'ensemble des prédécesseurs et l'ensemble des successeurs de m.
- $\boldsymbol{R_{Pred}}$ indique que chaque prédécesseur de n dans le mapping partiel se correspond à un nœud qui est un prédécesseur de m. L'inverse de cette règle est également vrai, c'est-à-dire que chaque prédécesseur de m dans le mapping partiel correspond au nœud qui est un prédécesseur de n.
- $\boldsymbol{R_{Succ}}$ indique que chaque successeur de n dans le mapping partiel correspond au nœud qui est un successeur de m. L'inverse de cette règle est également vrai.
- $\boldsymbol{R_{In}}$ indique que le nombre de prédécesseurs (successeurs) de n qui se trouvent dans ${T_1}^{in}(s)$ est égal au nombre de prédécesseurs (successeurs) de m présents dans ${T_2}^{in}(s)$.
- $\boldsymbol{R_{Out}}$ indique que le nombre de prédécesseurs (successeurs) de n qui sont dans ${T_1}^{out}(s)$ est égal au nombre de prédécesseurs (successeurs) de m présents dans ${T_2}^{out}(s)$.
- $\boldsymbol{R_{New}}$ indique que le nombre de prédécesseurs (successeurs) de n qui ne sont ni dans ${T_1}^{in}(s)$ ni dans ${T_1}^{out}(s)$ est égal au nombre de prédécesseurs (successeurs) de m qui ne sont ni dans ${T_2}^{in}(s)$ ni dans ${T_2}^{out}(s)$.

La fonction $\boldsymbol{F_{syn}(s,n,m)}$ est définit comme suit :

$$\boldsymbol{F(s,n,m)} = R_{Pred}\ (s,n,m) \wedge R_{Succ}\ (s,n,m) \wedge R_{In}(s,n,m) \wedge R_{Out}(s,n,m) \wedge R_{New}(s,n,m)$$

Où : les cinq règles syntaxiques sont déterminées comme suit :

$\boldsymbol{R_{Pred}(s,n,m)} \Leftrightarrow$

$$\big(\forall n' \in M_1(s) \cap Pred(G_1,n):\ \exists m' \in pred\ (G_2,m)\big|(n',m') \in M(s)\big)$$
$$\wedge\ \big(\forall m' M_2(s) \cap Pred(G_2,m): \exists n' \in pred\ (G_1,n)\big|(n',m')\ \in M(s)\big)$$

$\boldsymbol{R_{Succ}(s,n,m)} \Leftrightarrow$

$$\big(\forall n' \in M_1(s) \cap Succ(G_1,n):\ \exists m' \in Succ\ (G_2,m)\big|(n',m')\ \in M(s)\big)$$
$$\wedge\ (\forall m' M_2(s) \cap Succ(G_2,m): \exists n' \in Succ\ (G_1,n)|(n',m')\ \in M(s))$$

$\boldsymbol{R_{in}(s,n,m)} \Leftrightarrow$

$$\big(\big|Succ(G_1,n) \sqcap {T_1}^{in}(s)\big| \geqslant \big|Succ\ (G_2,m\big) \sqcap {T_2}^{in}(s))$$
$$\wedge\ (|Pred\ (G_1,n) \sqcap {T_1}^{in}(s)| \geqslant |Pred\ (G_2,m) \sqcap {T_2}^{in}(s)|)$$

$\boldsymbol{R_{out}(s,n,m)} \Leftrightarrow$

$$\big(\big|Succ(G_1,n) \sqcap {T_1}^{out}(s)\big| \geqslant \big|Succ\ (G_2,m\big) \sqcap {T_2}^{out}(s))$$
$$\wedge(|Pred\ (G_1,n) \sqcap {T_1}^{out}(s)| \geqslant |Pred\ (G_2,m) \sqcap {T_2}^{out}(s)|)$$

$\boldsymbol{R_{new}(s,n,m)} \Leftrightarrow$

$$\big|\big(N_1(s) \cap Pred(G_1,n)\big)\big| \geqslant \big|\big(N_2(s) \cap Pred(G_2,m)\big)\big|$$
$$\wedge\ |(N_1(s) \cap succ(G_1,n))| \geqslant |(N_2(s) \cap succ(G_2,m)))|$$

Avec :

$$N_2(s) = N_1 - M_1(s) - {T_1}^{in}(s) \cup\ {T_2}^{out}(s)$$

$$N_1(s) = N_2 - M_2(s) - {T_2}^{in}(s) \cup\ {T_2}^{out}(s)$$

Ces deux ensembles servent à voir les nœuds qui restent dans N_1 et N_2 (ensembles des nœuds des deux graphes G_1 et G_2 à mettre en correspondances)

Les règles ci-dessus sont utilisées pour trouver l'isomorphisme de sous-graphes. Pour tester l'isomorphisme des graphes, il faut remplacer l'opérateur (⩾) par l'opérateur (=) dans les 3 règles ($R_{In}(s,n,m)$, $R_{Out}(s,n,m)$, $R_{New}(s,n,m)$).

Règle sémantique

Lorsque les graphes sont attribués, l'inclusion des attributs des nœuds et des arcs dans l'algorithme de correspondance peut être effectué de deux façons, selon les types des attributs (symboliques ou numériques (réels). Pour les attributs symboliques qui, dans certains cas, sont dérivés d'attributs numériques à travers un processus de

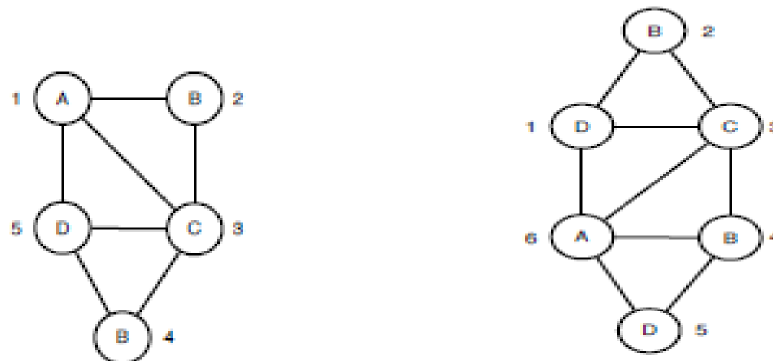
quantification, l'auteur de l'algorithme propose qu'une relation de compatibilité ≈ est définit entre deux attributs de nœud / arc. Pour certaine applications, ≈ peut coïncider avec la relation d'égalité, alors que dans d'autres cas, une définition plus "tolérante" peut être nécessaire. A chaque fois, l'algorithme vérifie la faisabilité d'une nouvelle paire, les attributs des nœuds et des arcs ajoutés sont testés pour la compatibilité sémantique par la règle sémantique $F_{sem}(s, n, m)$, qui est définie formellement par :

$$F_{sem}(s,n,m) \Leftrightarrow n \approx m \ \ (\forall(n',m') \in M(s), ((n,n',)\epsilon E_1 \Rightarrow (n,n') \approx (m,m') \wedge$$
$$(\forall(n',m') \in M(s), ((n',n)\epsilon E_1 \Rightarrow (n',n) \approx (m',m)$$

Pour les attributs numériques, ces informations sont exploitées de deux manières. Tout d'abord, une relation de compatibilité est définie sur la base d'un seuillage sur la différence absolue des attributs qui sont en correspondance, ce qui conduit à une fonction de faisabilité sémantique analogue à $F_{sem}(s, n, m)$ déjà définit auparavant. En outre, une fonction de coût est introduite pour donner une évaluation quantitative de la dissimilarité entre deux nœuds ou arcs. L'algorithme, dans son exploration de l'espace de recherche, enregistre uniquement les correspondances obtenues qui ont un coût minimal. Ce coût est réellement calculé pour chaque état s, comme la somme des coûts des états parents et des coûts dus aux nouveaux nœuds et arcs ajoutés. Puisque ces derniers sont supposés non négatifs, le coût total d'un état sera supérieur ou égal aux coûts de tous ses ancêtres. Nous pouvons utiliser cette information pour élaguer tous les états dont le coût est supérieur au coût du meilleur état de but atteint jusqu'à présent, réduisant encore plus l'espace de recherche. [50]

2.5. Exemple du fonctionnement de l'algorithme VF2

Pour permettre une bonne compréhension du fonctionnement de l'algorithme VF2, nous proposons un exemple qui teste l'isomorphisme des deux graphes G_1, G_2 de la figure 3.10 suivante :



Graphe G_2 Graphe G_2

Figure 3.10 Exemple de deux graphes à apparier avec l'algorithme VF2

Selon la description précédente de l'algorithme VF2, 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 appariement vide. Puis il explore l'espace de recherche selon une stratégie de recherche en profondeur avec retour arrière. A chaque itération, l'algorithme recherche un nouveau couple candidat pour générer un nouvel état. Dans ce but, VF2 utilise un ensemble de règles pour déterminer si ce nouveau couple consistant ou pas pour ne pas explorer ces descendants, chose qui permet de réduire le nombre d'états explorés. Pour l'exemple donné, la figure 3.4 montre tous les états trouvés par VF2 pendant l'exploration. Les états représentés par des lignes pointillées sont ceux générés mais non explorés car ils sont inconsistants. D'autre part, les états représentés par des traits pleins sont ceux qui sont cohérents ou consistants. Notez que la numérotation dépend de l'ordre dans lequel les états sont générés par l'algorithme.

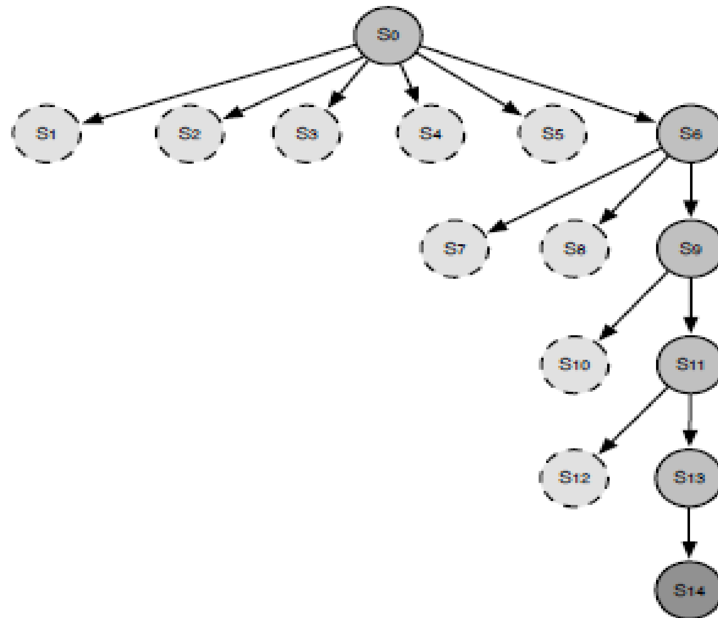


Figure 3.11 Montre tous les états trouvés par VF2

Ci-dessus, on trouve l'ensemble de base pour chaque état et les correspondances trouvées qui sont écrites en gras.

Etats	**Correspondances trouvés**
s_0	$M(s_0) = \varnothing$
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	$\boldsymbol{M(s_6) = \{(1,6)\}}$
s_7	$M(s_7) = \{(1,6),(2,1)\}$
s_8	$M(s_8) = \{(1,6),(2,3)\}$
s_9	$\boldsymbol{M(s_9) = \{(1,6),(2,4)\}}$
s_{10}	$M(s_{10}) = \{(1,6),(2,4),(3,1)\}$
s_{11}	$\boldsymbol{M(s_{11}) = \{(1,6),(2,4),(3,3)\}}$
s_{12}	$M(s_{12}) = \{(1,6),(2,4),(3,3),(4,1)\}$
s_{13}	$\boldsymbol{M(s_{13}) = \{(1,6),(2,4),(3,3),(4,2)\}}$
s_{14}	$\boldsymbol{M(s_{14}) = \{(1,6),(2,4),(3,3),(4,2),(5,1)\}}$

Les ensembles générés par l'algorithme VF2, ainsi que les deux sous-graphes isomorphes induits sont présentés dans la figure 3.13 et la figure 3.14 respectivement.

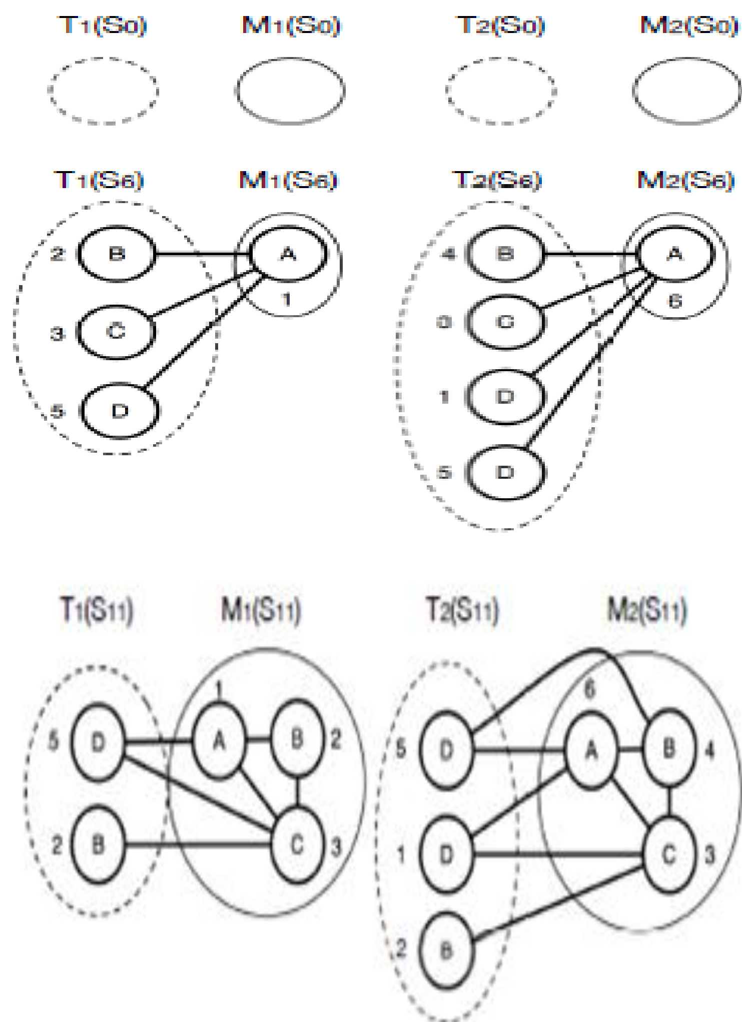


Figure 3.13 : Montre les ensembles de base (lignes continues) et l'ensemble terminal (ligne pointillées) liés au états s_0, s_6 et s_{11}.

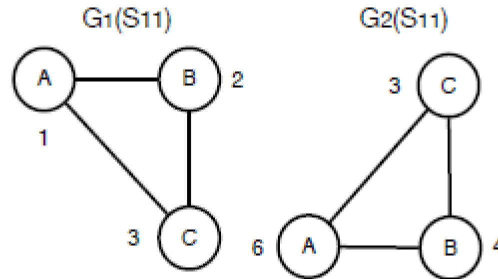


Figure 3.14 : Montre les deux sous-graphes isomorphes induits par les nœuds qui sont à l'intérieur du mapping à l'état s11. Il est clair que s11 est consistant car il satisfait les contraintes imposées.

4 Conclusion

Notre première constatation après avoir décrire et étudier le fonctionnement des deux algorithmes les plus efficaces pour résoudre le problème d'isomorphisme des graphes et des sous-graphes : l'algorithme d'Ullmann et l'algorithme VF2 est que les deux algorithmes se basent sur l'exploration de l'arbre de recherche avec l'intégration des heuristiques pour permettre de réduire le temps de recherche et par conséquent réduire la complexité temporelle et spatiale. Ullmann propose de trouver toutes les permutations possibles (calculer les matrices de permutation) et d'introduire des procédures d'énumération et de raffinement pour rendre l'algorithme plus efficace. Tandis que l'algorithme VF2 s'appuie sur la recherche des mises en en correspondances partielles puis l'intégration des règles de faisabilité dans le processus de matching afin de n'explorer que les nœuds susceptibles d'être dans l'appariement final.

Chapitre 4

Implémentation et résultats

1 Introduction

Dans le présent chapitre, on débute par un petit aperçu sur la programmation « orienté objet » et sur le langage Java qu'on a utilisé pour le développement de notre application ainsi que l'outil Jung qui a été intégré pour faciliter la visualisation des graphes exemples et des résultats obtenus. On présente ensuite notre expérimentation qui vise à implémenter les deux algorithmes qu'on a détaillé dans le chapitre précédent et qui sont considérés parmi les algorithmes de référence pour toute comparaison de performance avec les algorithmes d'isomorphisme de (sous-)graphes. L'algorithme d'Ullmann qui date de 1976 reste toujours parmi les algorithmes les plus efficaces dans le domaine d'appariement de graphe par arbre de recherche à cause des procédures introduites par Ullmann pour réduire l'espace de recherche. L'algorithme VF2 a amélioré l'algorithme d'Ullmann et il a donné un autre envol dans ce domaine par l'intégration des règles de faisabilité dans l'algorithme, chose qui a réduit la complexité de l'algorithme d'Ullmann de $O(N^2)$ à $O(N)$ selon [46]. Notre expérimentation consiste alors à chercher l'isomorphisme de deux (sous-)graphes en appliquant les deux algorithmes et à agrandir à chaque fois la taille de deux graphes pour comparer la complexité temporelle des deux algorithmes cités.

2 Approche orienté objet

La programmation orientée objet (POO) est maintenant au cœur des méthodes modernes de programmation, c'est une nouvelle méthode de programmation qui tend à se rapprocher de notre manière naturelle d'appréhender le monde. Elle propose une méthodologie centrée sur les données c'est-à-dire, elle s'intéresse d'abord aux données, auxquelles elle associe ensuite les traitements.

La programmation orientée objet, cherche à décrire un problème de façon la plus proche possible de la spécification de ce dernier. Apparue dans les années 1970 avec le langage Smalltalk, elle connaît un essor formidable dix ans plus tard avec l'apparition de langages tels C++, puis Java au milieu des années 1990. Cette approche propose une entité logicielle appelée objet, qui, associée à des concepts tels que l'**encapsulation** des données, les messages entre objets, l'**héritage**, etc., confère au langage une expressivité accrue par rapport à la programmation procédurale [59].

L'unité de base d'un programme écrit dans un langage dit « orienté objet », est l'objet. **Un objet** est une entité logicielle autonome modélisant l'aspect statique et dynamique d'une partie du monde réel. Cette entité est constituée par l'ensemble (identité, état, comportement) : l'**identité** est l'identifiant qui permet de distinguer l'objet parmi d'autres objets du même type dans le système, l'**état** est l'ensemble des valeurs instantanées que prennent les **attributs** et il évolue dans le temps. Les attributs de l'objet (on parle aussi de **données membres**) sont les données caractérisant l'état de l'objet, son **comportement** décrit les actions que peut effectuer l'objet ainsi que les réactions qu'il peut avoir en réponses à des **messages** envoyés par d'autres objets. Le comportement est défini par des **méthodes**. Celles-ci permettent à l'objet d'agir sur ses attributs pour modifier son état et ainsi réagir aux sollicitations extérieures. Le concept permettant de classer les objets en catégories est la classe. Une **classe** est un type de données abstrait regroupant les caractéristiques (attributs et méthodes) communes à des objets et permettant de créer des objets possédant ces propriétés [59].

3 Historique sur le langage Java

Java est un langage de programmation proche du langage C++. Il est surtout connu par son utilisation sur Internet car c'est un langage à pouvoir s'adapter à des systèmes informatiques différents. Le développement de Java a commencé en 1990 chez Sun Microsystems Inc. à Mountain View, en Californie. Une équipe dirigée par James Gosling se consacrait au développement d'un nouveau langage de programmation destiné au pilotage des appareils électroménagers. Le projet, qui portait le nom de 'Oak' (chêne en anglais, car on voyait un chêne depuis les fenêtres du bureau où se développait le projet) avait pour but d'éliminer la compilation spécifique au système d'exploitation de l'appareil en question. Le nom 'Oak' était déjà utilisé par un autre langage de programmation. Il fallait donc trouver un nouveau nom. L'équipe de développement ne parvenait pas à trouver un nom adéquat, elle s'accorda une pause café puis, donna au nouveau langage le nom 'Java' qui veut dire 'café' [58].

Java est un langage de programmation à usage général, évolué et orienté objet dont la syntaxe est proche du C. Il existe 2 types de programmes en Java : les applets et les applications. Une application autonome (stand alone program) est une application qui s'exécute sous le contrôle direct du système d'exploitation. Une applet est une application qui est chargée par un navigateur et qui est exécutée sous le contrôle d'un

plug in de ce dernier. Java possède un certain nombre de caractéristiques qui ont largement contribué à son énorme succès : [57]

Java est interprété : le source est compilé en pseudo code ou byte code puis exécuté par un interpréteur Java: la Java Virtual Machine (JVM). Ce concept est à la base du slogan de Sun pour Java : WORA (Write Once, Run Anywhere : écrire une fois, exécuter partout). En effet, le byte code, s'il ne contient pas de code spécifique à une plate−forme particulière peut être exécuté et obtenir quasiment les même résultats sur toutes les machines disposant d'une JVM.

Java est indépendant de toute plate−forme : il n'y a pas de compilation spécifique pour chaque plateforme. Le code reste indépendant de la machine sur laquelle il s'exécute. Il est possible d'exécuter des programmes Java sur tous les environnements qui possèdent une Java Virtual Machine. Cette indépendance est assurée au niveau du code source grâce à Unicode et au niveau du byte code.

Java est orienté objet : comme la plupart des langages récents, Java est orienté objet. Chaque fichier source contient la définition d'une ou plusieurs classes qui sont utilisées les unes avec les autres pour former une application. Java n'est pas complètement objet car il définit des types primitifs (entier, caractère, flottant, booléen,...).

Java est simple : le choix de ses auteurs a été d'abandonner des éléments mal compris ou mal exploités des autres langages tels que la notion de pointeurs (pour éviter les incidents en manipulant directement la mémoire), l'héritage multiple et la surcharge des opérateurs,...etc.

Java est fortement typé : toutes les variables sont typées et il n'existe pas de conversion automatique qui risquerait une perte de données. Si une telle conversion doit être réalisée, le développeur doit obligatoirement utiliser une méthode statique fournie en standard pour la réaliser.

Java assure la gestion de la mémoire : l'allocation de la mémoire pour un objet est automatique à sa création et Java récupère automatiquement la mémoire inutilisée.

Java est sûr : la sécurité fait partie intégrante du système d'exécution et du compilateur. Un programme Java planté ne menace pas le système d'exploitation. Il ne peut pas y avoir d'accès direct à la mémoire. L'accès au disque dur est réglementé dans une applet. Les applets fonctionnant sur le Web sont soumises à des restrictions de sécurité.

Java est économe : le pseudo code a une taille relativement petite car les bibliothèques de classes requises ne sont liées qu'à l'exécution.

Java est multitâche : il permet l'utilisation de threads qui sont des unités d'exécution isolées. La JVM, elle même, utilise plusieurs threads.

4 Environnement Java

Sun fourni gratuitement un ensemble d'outils et d'API pour permettre le développement de programmes avec Java. Ce kit, nommé JDK (Java 2 Software Development Kit), est librement téléchargeable. Le JRE (Java Runtime Environment) contient uniquement l'environnement d'exécution de programmes Java. Le JDK contient lui-même le JRE. Le JRE seul doit être installé sur les machines où des applications Java doivent être exécutées [57]. Plusieurs implantations de Java sont disponibles, dont celle d'origine ORACLE/SUN (createur du langage) et quelques autres. Pour exécuter du code Java : le JRE (Java Runtime Environment) d'ORACLE/SUN suffit (gratuit et largement disponible). Pour créer du code java, plusieurs possibilités (au choix) existent [60] :

- **Java SE** (Standard Edition) dénommé aussi ≪ **JDK** ≫ (Java Development Kit) d'ORACLE/SUN (gratuit) contient compilateur (javac), interpreteur / machine virtuelle (java), divers outils génération doc (javadoc), ...
- la plateforme de développement ≪ **Eclipse** ≫ (gratuite, la plus utilisee)
- la plateforme de développement ≪ **NetBeans** ≫ (gratuite, developpee par
- Oracle).
- ou l'une des nombreuses autres plateformes de développement permettant de développer du code Java...

4.1. Présentation de Java Eclipse

Eclipse est un Environnement de Développement Intégré (IDE), spécialement conçu pour le développement en Java, créé à l'origine par IBM puis cédé à la communauté Open Source [62].

Eclipse est un projet décliné et organisé en un ensemble de sous-projets de développements logiciels, de la fondation Eclipse visant à développer un environnement de production de logiciels libres qui soit extensible, universel et polyvalent, en s'appuyant principalement sur Java [61].

Actuellement, on peut distinguer deux grandes catégories de programmes, en se fondant sur leur *interface* avec l'utilisateur, c'est-à-dire sur la manière dont se font les échanges d'informations entre l'utilisateur et le programme :

• les programmes à interface console,

• les programmes à interface graphique.

- **Les programmes à interface console (ou en ligne de commande)**

Historiquement, ce sont les plus anciens. Dans ces programmes, on fournit des informations à l'écran sous forme de lignes de texte s'affichant séquentiellement, c'est-à-dire les unes à la suite des autres. Pour fournir des informations au programme, l'utilisateur frappe des caractères au clavier (généralement un "écho" apparaît à l'écran). Avec une interface console, c'est le programme qui décide de l'enchaînement des opérations : l'utilisateur est sollicité au moment voulu pour fournir les informations demandées. L'interface console n'utilise qu'une seule fenêtre (dans certains anciens environnement, la fenêtre n'était même pas visible, car elle occupait tout l'écran). Celle-ci ne possède qu'un petit nombre de fonctionnalités : déplacement, fermeture, parfois changement de taille et défilement [61].

- **Les programmes à interface graphique (G.U.I.)**

Dans ces programmes, la communication avec l'utilisateur se fait par l'intermédiaire de *composants* tels que les menus déroulants, les barres d'outils ou les boîtes de dialogue, ces dernières pouvant renfermer des composants aussi variés que les boutons poussoirs, les cases à cocher, les boutons radio, les boîtes de saisie, les listes déroulantes... L'utilisateur a l'impression de piloter le programme, qui semble répondre à n'importe laquelle de ses demandes. On parle souvent dans ce cas de *programmation événementielle*, expression qui traduit bien le fait que le programme réagit à des évènements provoqués par l'utilisateur. Le terme G.U.I. (*Graphical User Interface*) tend à se généraliser pour désigner ce genre d'interface. Manifestement, il met en avant le fait que, pour permettre ce dialogue, on ne peut plus se contenter d'échanger du texte et qu'il faut effectivement être capable de dessiner, donc d'employer une interface graphique. Il n'en reste pas moins que l'aspect le plus caractéristique de ce type de programme est dans l'aspect événementiel. L'interface graphique utilise une fenêtre principale qui s'ouvre au lancement du programme. Il est possible que d'autres fenêtres apparaissent par la suite. L'affichage des informations dans ces fenêtres ne se fait plus séquentiellement. Il est généralement nécessaire de prendre en compte l'aspect

"coordonnées". En contrepartie, on peut afficher du texte en n'importe quel emplacement de la fenêtre, utiliser des polices différentes jouer sur les couleurs, faire des dessins, afficher des images... [61].

4.2 JavaFX

JavaFX est une famille de produits et de technologies de Sun Microsystems qui appartient à Oracle, qui base sur la Machine Virtuel Java pour fonctionner donc la communication avec des applications java standard est très simple. Une application *JavaFX* a accès à toutes les classes fournies par la machine virtuelle java. Les produits *JavaFX* ont pour but de créer des applications internet riches et facilite le développent avec images, graphiques, audio et vidéo. Actuellement *JavaFX* est constitué de *JavaFX* Script et de *JavaFX* Mobile, bien que d'autres produits soient prévus. Il est basé sur Java et utilisable sur tous les écrans : navigateurs, mobile, TV, etc. Il est Open Source.

Le graphe de scène (scene graph) est une notion importante qui représente la structure hiérarchique de l'interface graphique. Techniquement, c'est un graphe acyclique orienté (arbre orienté) représenté par la figure 4.1 :

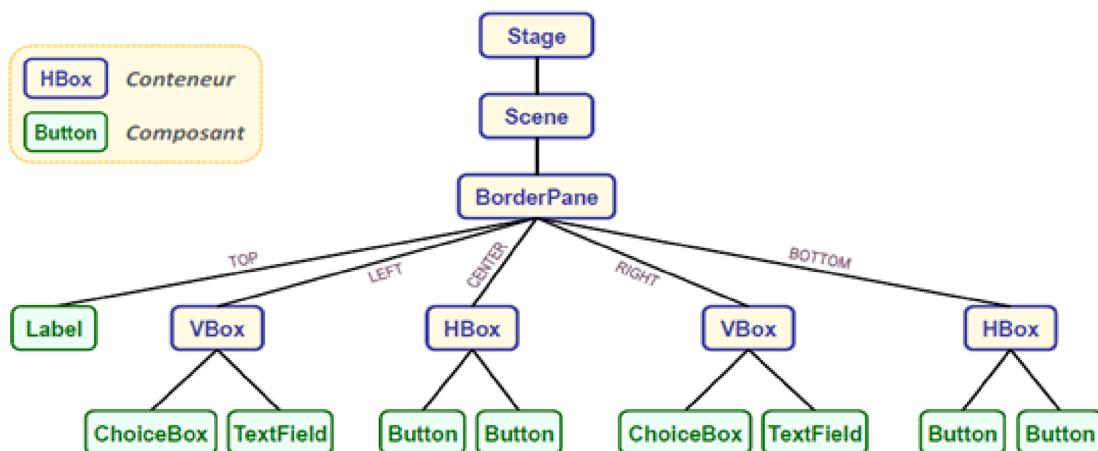


Figure 4.1 : Application *JavaFX* et son graphe de scène.

Tous les éléments contenus dans un graphe de scène sont des objets qui ont pour classe parente la classe Node. Cette classe comporte de nombreuses sous-classes, on peut distinguer différentes familles :

- Les formes primitives (Shape)2D et 3D: Line, Circle, Rectangle, Box, Cylinder, …

- Les conteneurs (Layout-Pane) qui se chargent de la disposition (layout) des composants enfants et qui ont comme classe parente Pane : AnchorPane, BorderPane, GridPane, HBox, VBox, …
- Les composants standard (Controls) qui étendent la classe Control : Label, Button, TextField, ComboBox, …
- 4)- Les composants spécialisés qui sont dédiés à un domaine particulier (par exemple : lecteur multimédia, navigateur web, etc.) : MediaView, WebView, ImageView, Canvas, Chart…

Avec *JavaFX*, les interfaces peuvent être créées de deux manières :

En écrivant du code Java qui fait appel aux API de la plateforme et qui utilise les composants/conteneurs à disposition (classes et interfaces). En décrivant l'interface dans un fichier au format FXML qui sera ensuite chargé dynamiquement dans l'application. Cette deuxième possibilité permet de créer les interfaces (les vues) en utilisant notamment l'outil SceneBuilder qui permet, de manière interactive, de créer les fichiers FXML. le fichier FXML est un fichier au format XML dont la syntaxe est conçue pour décrire l'interface (la vue) avec ses composants, ses conteneurs, sa disposition, … Il décrit le "quoi" mais pas le "comment". Le FXML est un format de fichier propre à JavaFX et sert, entre autres, à définir des interfaces graphiques. Ce format utilise tout simplement une syntaxe XML.

A l'exécution, le fichier FXML sera chargé par l'application (classe FXMLLoader) et un objet Java sera créé (généralement la racine est un conteneur) avec les éléments que le fichier décrit (les composants, conteneurs, graphiques, …). Il est possible de créer les fichiers FXMLs avec un éditeur de texte mais, plus généralement, on utilise l'outil graphique (SceneBuilder) qui permet de concevoir l'interface de manière conviviale et de générer automatiquement le fichier FXML correspondant. Les objets créés par le chargement de fichiers FXML peuvent être assignés à la racine d'un graphe de scène ou représenter un des nœuds dans un graphe de scène.

4.3 SceneBuilder

Le JavaFXSceneBuilder est un outil initialement créé par Oracle et fait partie de l'Open JFX qui permet de créer des fichiers au format FXML via un éditeur graphique. Cet outil est disponible en tant qu'application autosuffisante qui peut être

lancée depuis le bureau windows ou en tant qu'API intégré dans des outils tels qu'Eclipse. Cet outil permet de concevoir l'interface de manière interactive en assemblant les conteneurs et les composants et en définissant leurs propriétés. Le mode de fonctionnement de cet utilitaire est assez classique avec une zone d'édition centrale, entourée d'un certain nombre d'outils: palettes de conteneurs, de composants, de menus, de graphiques, vue de la structure hiérarchique de l'interface, inspecteurs de propriétés, de layout, etc.

4.4 Bibliothèque JUNG

JUNG pour Java Universal Network Graph Framework est une plateforme logicielle contenant une bibliothèque de programmes Java pour générer des graphes de visualisation de réseaux [63].

JUNG est une bibliothèque logicielle open-source qui fournit un langage commun et extensible pour la modélisation, l'analyse et la visualisation de données pouvant être représentées sous forme de graphique ou de réseau. Il est écrit en Java, ce qui permet aux applications basées sur JUNG d'utiliser les capacités étendues intégrées de l'API Java, ainsi que celles des autres bibliothèques Java tierces existantes [net01].

4.5 Matériel utilisé

Le PC qu'on a travaillé sur est un PC de type Intel qui possède comme caractéristique :

- ✓ Un processeur Intel ® core ™ i3-3217U CPU @ 1.8 Ghz 1.8 Ghz.
- ✓ Type de système : Système d'exploitation 32 bits.
- ✓ Mémoire installée (RAM) : 4,00 Go (2.65 Go utilisable).

6. Présentation de l'application

Après avoir présenté, dans le chapitre précédent, les 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 deux algorithmes d'appariement de graphes (l'algorithme d'Ullmann et l'algorithme VF2) puis comparer la complexité des deux algorithmes.

L'architecture du système contient deux module : le module 'construire/importer graphe' et le module matching graphes.

6.1-Module 'Construire / Importer Graphe' :

Ce module permet à l'utilisateur à travers une interface graphique de créer un nouveau graphe ou d'importer un graphe déjà existant. La fonction principale de ce module sert à offrir une interface applicable aux besoins des utilisateurs pour créer un graphe d'une façon relativement simple et inventive ou de l'importer à partir d'un fichier existant. Ce module est schématisé par la figure 4.2.

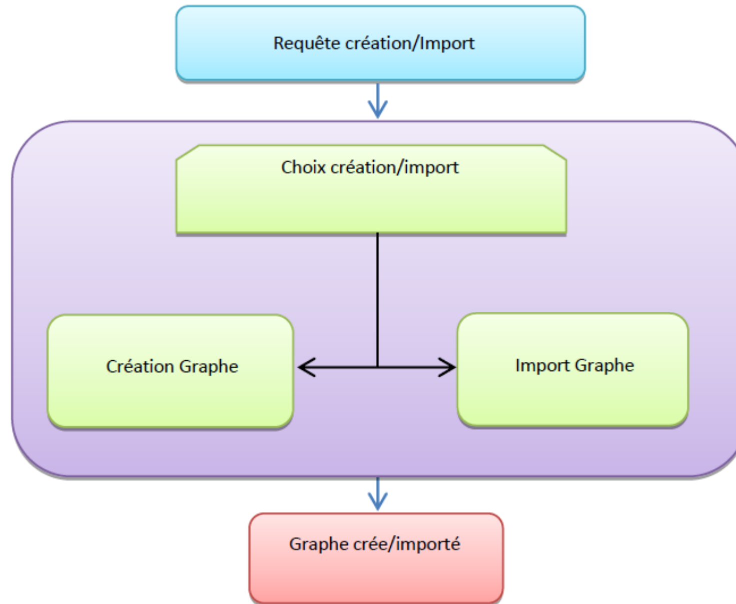


Figure 4.2 Module Création/Import graphe.

6.2-Module 'Matching Graphe' :

Ce module permet d'apparier deux graphes en appliquant l'algorithme d'Ullmann qui est un algorithme d'isomorphisme de (sous-)graphes reconnu, puis l'algorithme VF2 qui est une amélioration de ce dernier. L'architecture de ce module peut être représentée comme suit par la figure 4.3.

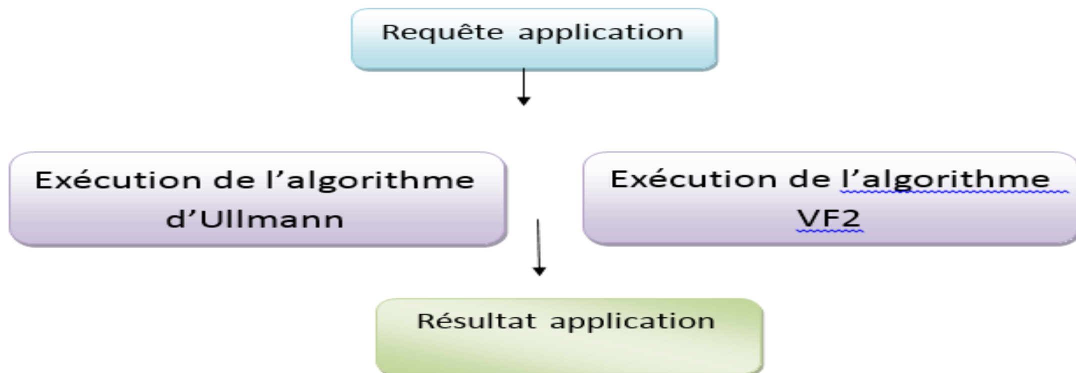


Figure 4.3 : Module Matching graphes

L'organigramme de notre application est représentée par la figure 4.4.

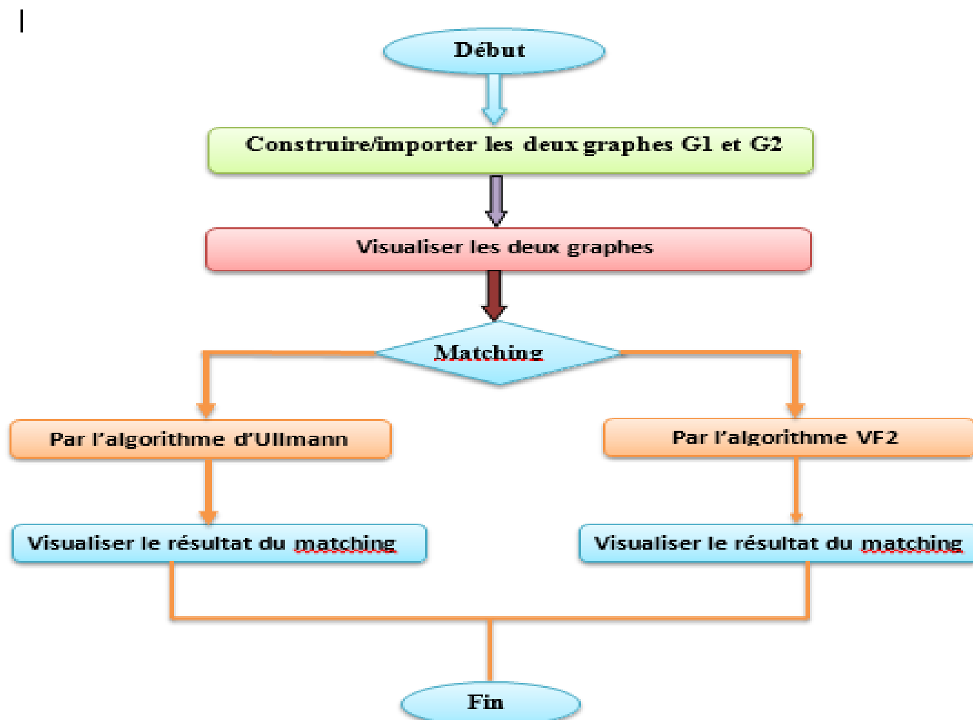


Figure 4.4: Organigramme de l'application.

6.3 Présentation de l'interface principale de l'application

L'interface principale de notre application est présentée par la figure 4.5 comme suit :

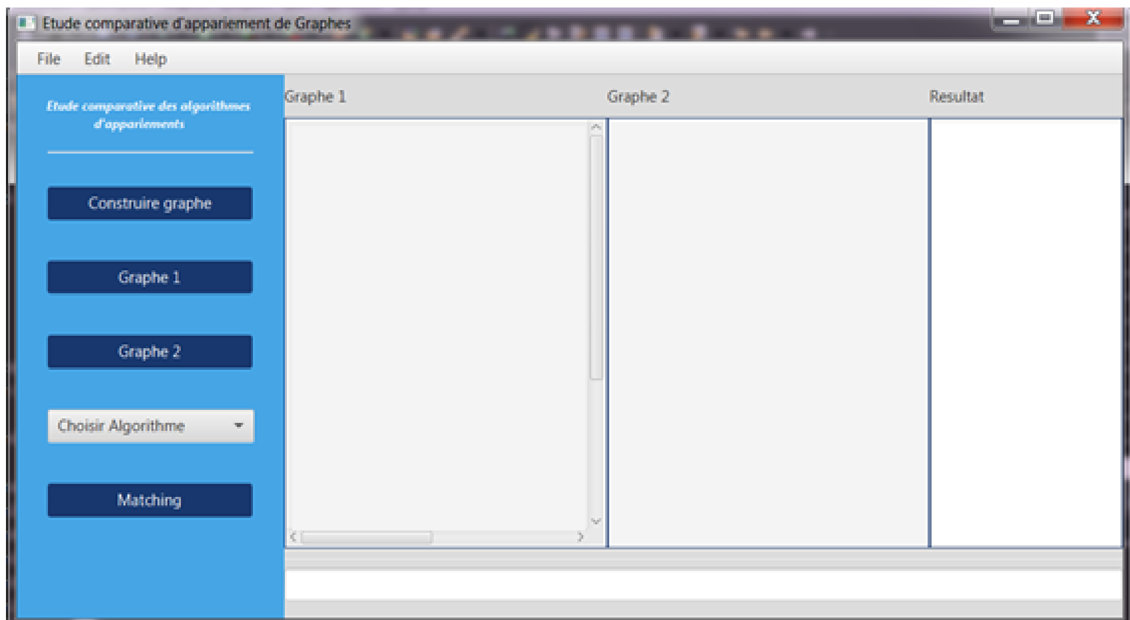


Figure 4.5: Interface principale de l'application

Après avoir exécuté l'onglet 'construire ou importer graphes', puis choisir un des algorithmes, correspondances obtenues sont visualisés à droite et en bas le temps pris par l'algorithme pour trouver le matching (voir l'exemple de la figure 4.6).

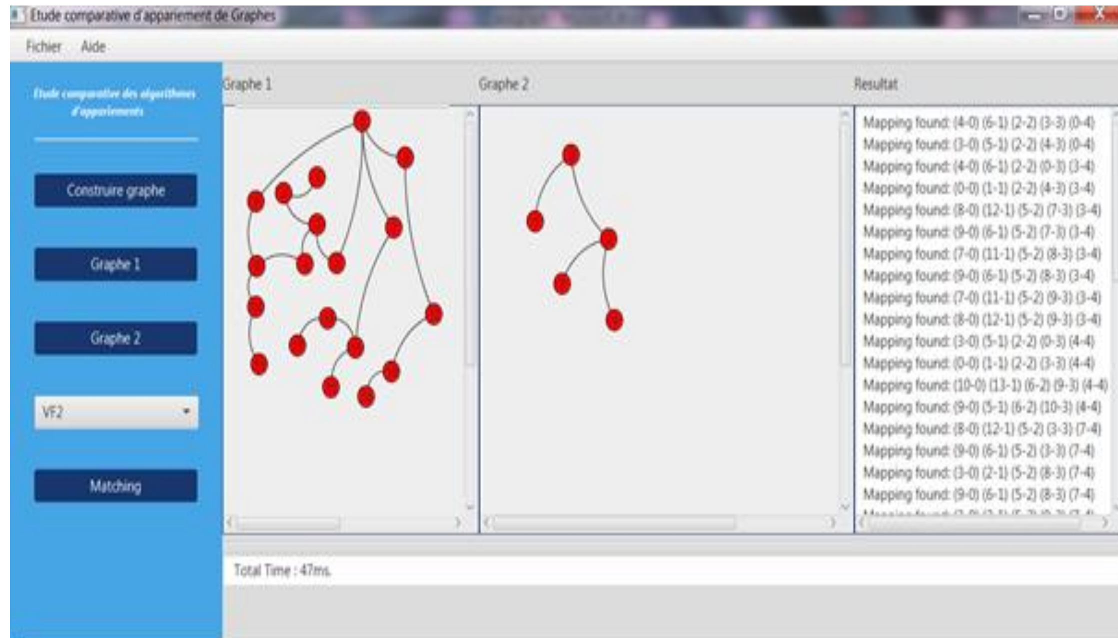


Figure 4.6: Résultats obtenus en appliquant l'algorithme VF2

La figure 4.7 présente le résultat obtenu par l'algorithme d'Ullmann avec le même jeu d'essai.

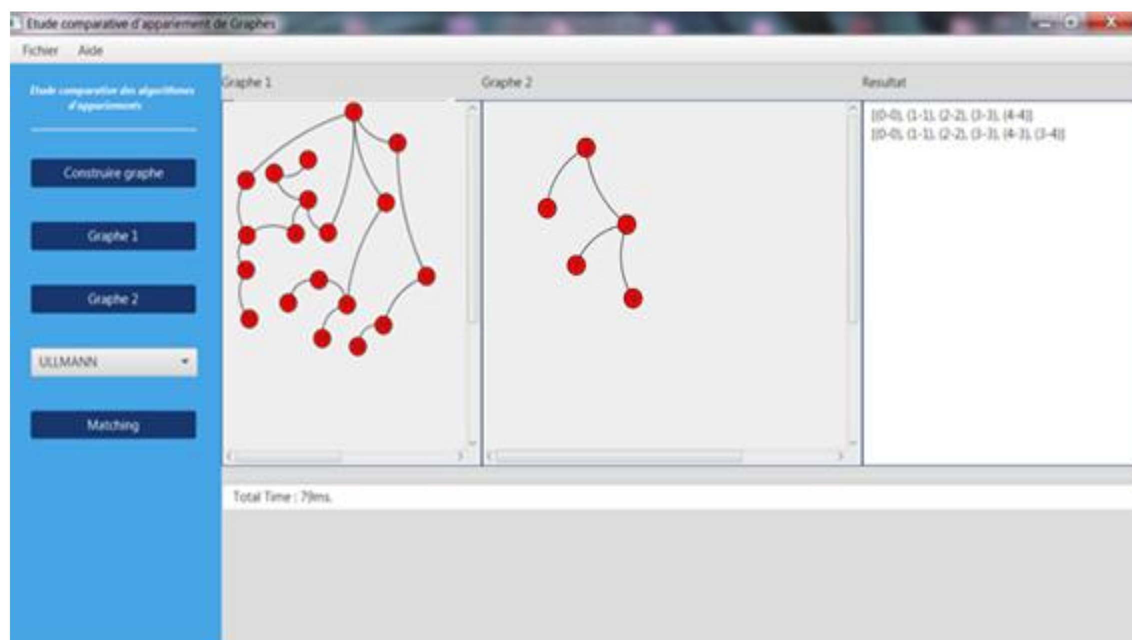


Figure 4.7: Résultats obtenus en appliquant l'algorithme d'Ullmann

6.4 Interprétation des résultats

Le tableau 4.1 ci-après récapitule les résultats obtenus par les deux algorithmes avec quelques essais en choisissant des graphes de taille différente.

Taille du graphe de données	Taille du graphe requête	Temps de matching en ms (milli seconde)	
		Algorithme d'Ullmann	Algorithme VF2
5	3	83 ms	30 ms
10	6	90 ms	75 ms
19	6	79 ms	47 ms
41	9	1172 ms	130 ms

Tableau 1 : Résultats obtenus en appliquant l'algorithme d'Ullmann et VF2 sur quelques graphes.

La courbe de la figure 4.8 indique que : quand le nombre de nœuds des deux graphes sont petits, les résultats obtenus par les deux algorithmes sont plus au moins proches mais avec un graphe de données de 41 nœuds, rechercher sa mise en correspondance avec un graphe requête de 9 nœuds par l'algorithme d'Ullmann a pris un temps de calcul de 46102 ms. Par contre en appliquant l'algorithme VF2, le temps de calcul était de 130 ms. D'où on conclut que VF2 est plus performant pour le jeu d'exemple qu'on a employé pour cette démonstration comparative entre les algorithmes.

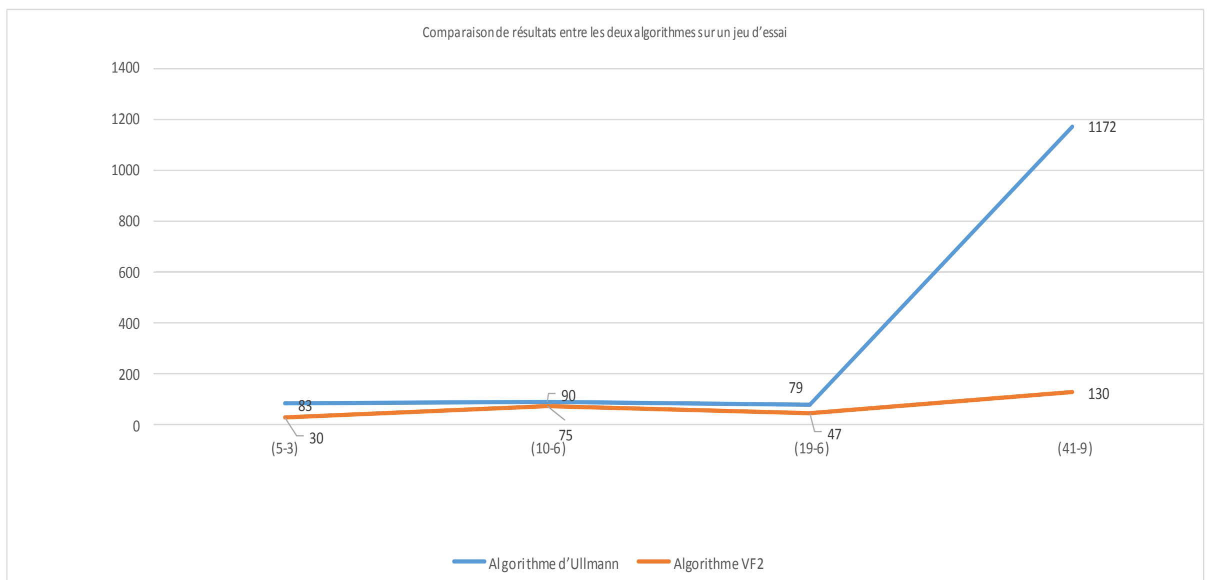


Figure 4.8: Comparaison des résultats entre les deux algorithmes sur un jeu d'essai

7 Conclusion

Les règles de faisabilité introduites dans l'algorithme VF2 servent à élaguer le maximum de nœuds jugés inadéquats. L'algorithme n'explore pas tout l'espace de recherche. Les résultats obtenus après chaque test interprète l'intérêt de cet heuristique. Par ailleurs, l'algorithme d'Ullmann basé sur le calcul des matrices de permutation et aussi sur une procédure de raffinement pour ne pas explorer l'arbre complet a donné des résultats satisfaisants mais après un temps de calcul plus long que celui pris par l'algorithme VF2.

Conclusion Générale

Les graphes consistent des moyens simples pour la modélisation des objets structurés. Le but de cette étude est de présenter l'appariement de graphes qui signifie généralement la comparaison entre graphes. L'appariement de graphe ou 'the graph matching' en anglais, est un processus complexe qui nécessite, dans le cas général, la préservation des structures de graphes. Plusieurs approches ont été abordées dans la littérature pour la résolution de ce problème considéré par plusieurs auteurs comme un problème dans NP-complet. Ces approches sont reparties en deux grandes catégorie : l'appariement exact et l'appariement approximatif ou inexact. La mise en correspondance de deux graphes revient alors à mettre en correspondance tous les nœuds des deux graphes ainsi que les arrêtes pour trouver le matching s'il existe (recherche d'isomorphisme entre les graphes). Cette contrainte a rendu l'appariement exact inapplicable dans des applications complexes du monde réel. Par ailleurs l'appariement inexact vise à remédier à cette contrainte en présentant des algorithmes qui cherchent à trouver une solution au problème avec un temps de calcul acceptable. L'état de l'art qu'on a établi nous a permis de découvrir les différentes techniques pour la résolution des deux types d'appariement.

Les résultats obtenus ont montré que l'algorithme VF2 était plus meilleur que l'algorithme d'Ullmann. Le point de comparaison était le temps de calcul que prenne chaque algorithme (après chargement des deux structures des deux graphes) pour trouver le matching entre le graphe de données et le graphe modèle (requête). On a procédé à incrémenter, à chaque essai, le nombre de nœuds des deux graphes à mettre en correspondance pour tester la complexité temporelle, puis chercher le matching entre eux en appliquant les deux algorithmes. Les résultats ont prouvé à chaque test l'efficacité de l'algorithme VF2.

Bibliographie

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

[2] E.D. Taillard, Informatique orientation logiciels - Éléments de la théorie des graphes Ecole d'ingénieur du Canton de vaud 2003.

[3] Justine Lebrun, Appariement inexact de graphes appliquée à la recherche d'image et d'objet 3 - Université de Cergy Pontoise, 2011. France.

[4] Nicolas Gastineau, Partitionnement, recouvrement et colorabilité dans les graphes Université de Bourgogne – France , 2004.

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

[6] J.A. Bondy et U.S.R. Murty, Théorie des Graphes Traduit de l'anglais par F. Hayet, 2008.

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

[8] Antoine Gournay, Théorie des graphes Institut de Mathématiques, Notes de Cours- Université de Neuchâtel Suisse Septembre 2013.

[9] Michel Rigo, Théorie des graphes, Université de Liège Faculté des sciences Département de mathématiques - Année académique 2009–2010

[10] Katia Abbaci, Allel Hadjali, Ludovic Liétard, Daniel Rocacher, Interrogation de bases de données de graphes : Une approche basée sur un skyline par similarité – Conférence Maghrébine sur l'Extraction et la Gestion des Connaissances (EGC-M), Algeria, Dec 2010.

[11] Sébastien Sorlin et Christine Solnon, Similarité de graphes : une mesure générique et un algorithme tabou réactif - Université de Lyon I France, 2003.

[12] Abdulkader A. M., Parallel Algorithms for Labelled Graph Matching, PhD thesis, Colorado School of Mines, 1998.

[13] Salim Jouili, Indexation de masses de documents graphiques : approches structurelles - Université Nancy 2, Mars 2011.

[14] Anh Phuong TA Inexact graph matching techniques: Application to object detection and human action recognition - Université de Lyon 2010

[15] BARATLI KARIM, Conception et implémentation d'un outil de filtrage de netlist pour un système de prototypage rapide - Université Du Québec En Outaouais Décembre 2012.

[16] Anh Phuong TA, Inexact graph matching techniques: Application to object detection and human action recognition, décembre 2010

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

[18] Kamel MADI, Inexact graph matching: application to 2D and 3D Pattern Recognition Université Claude Bernard Lyon 1 Décembre 2016.

[19] Corentin Ribeyre, Méthodes d'analyse supervisée pour l'interfaces syntaxe-sémantique de la réécriture de graphes à l'analyse par transitions - Université Sorbonne Paris Cité - Université Paris Diderot , Janvier 2016

[20] P. Foggia, G. Percannella, and M. Vento, "Graph matching and learning in pattern recognition in the last 10 years," International Journal of Pattern Recognition and Artificial Intelligence, 2014.

[21] S. Gunter and H. Bunke, "Self-organizing map for clustering in the graph domain, Pattern Recognition Letters , 2002.

[22] Rashid Jalal QURESHI, Reconnaissance de formes et symboles graphiques complexes dans les images de documents - Université François Rabelais , Mars 2008

[23] H. Bunke. On a relation between graph edit distance and maximum common subgraph. Pattern Recognition Letters, 1997.

[24] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. Pattern Recognition Letters, 1998.

[25] W. D.Wallis, P. Shoubridge, M. Kraetzl, and D. Ray. Graph distances using graph union. Pattern Recognition Letters, 2001.

[26] M.-L. Fernandez and G. Valiente. A graph distance metric combining maximum common subgraph and minimum common supergraph. Pattern Recognition Letters, 2001.

[27] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30(30) :47–87, 1981.

[28] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. Journal of the ACM, 1970.

[29] L. Cordella, P. Foggia, C. Sansone and M. Vento, "An improved algorithm for matching large graphs", Workshop on graph-based representations in pattern recognition, Italy 2001.

[30] B. Messmer and H. Bunke, “Recent advances in graph matching”, Journal of pattern recognition and artificial intelligence, 1997.

[31] B. T. Messmer and H. Bunke. A decision tree approach to graph and subgraph isomorphism detection. Pattern Recognition :1979–1998, 1999.

[32] K. Shearer, S. Venkatesh and H. Bunke, “Video sequence matching via decision tree path following”, Journal of pattern recognition letters, 2001.

[33] K. Shearer, H. Bunke, S. Venkatesh and S. Kieronska, “Efficient graph matching for video indexing”, In J. Jolion and W. Kropatsch, “Graph based representations in pattern recognition”, 1997.

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

[35] T. Caelli and S. Kosinov. An eigenspace projection clustering method for inexact graph matching. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2004.

[36] M. Carcassoni and E. R. Hancock. Correspondence matching with modal clusters. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2003.

[37] K. M. Borgwardt. *Graph Kernels*. PhD thesis, Ludwig–Maximilians University, Munich, 2007.

[38] B. Schölkopf, A. Smola, and K. R. Müller. Non linear component analysis as a kernel Eigen value problem. Neural Computation, 1998.

[39] M. Neuhaus and H. Bunke. Edit distance-based kernel functions for structural pattern classification. Pattern Recognition, 2006.

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

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

[42] C. Liu, K. Fan, J. Horng and Y. Wang, “Solving weighted graph matching problem by modified microgenetic algorithm”, Conference of IEEE system, .man and cybernetics, Vancouver, Canada, 1995.

[43] Y. Wang, K. Fan and J. Horng, “Genetic-based search for errorcorrecting graph isomorphism”, Journal of IEEE systems, man and cybernetics, 1997.

[44] K. Khoo and P. Suganthan, “Multiple relational graphs mapping using genetic algorithms”, Conference of evolutionary computation, Seoul, South Korea, 2001.

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

[46] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Evaluating Performance of the VF Graph Matching Algorithm. In Proc. of the 10th International Conference on Image Analysis and Processing, IEEE Computer Society Press, 1999.

[47] Péter Fehér, Methods for Improving and Applying Graph Rewriting-Based Model Transformations – université Budapest, 2017.

[48] Yaohui Lei, Graph and subgraph isomorphism problems - Université de Montréal, Avril 2003.

[49] Vincenzo Carletti, Exacts and Inexacts methods for graph similarity in Structural Pattern Recognition - Université de Caen Normandie, 2016.

[50] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone and Mario Vento, A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs - Octobre 2004

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

[52] J. R. Ullmann. An algorithm for subgraph isomorphism. Journal of the Association of Computing Machinery, Janvier 1976.

[53] Vincent A. Cicirello, Survey of Graph Matching Algorithms - Technical Report Geometric and Intelligent Computing Laboratory Drexel University March 1999

[54] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Fast graph matching for detecting cad image components. In 15th International Conference on Pattern Recognition, 2000. Proceedings, volume 2, 2000.

[55] L. P. 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.

[56] Wolf-Dieter, Graph Matching – Algorithms Pattern Recognition and Image Processing Group (PRIP) Institute of Computer Graphics and Algorithms, 2011

[57] Jean Michel DOUDOUX, Développons en Java - Version 0.85 béta du 22/11/2005

[58] Patrick C´egielski cegielski@u-pec.fr, Initiation au langage Java Mai 2010

[59] BTS IRIS – Cours et Travaux Pratiques Programmation Java (version 0.1) Lebret, TSIRIS, © Lycée Diderot, 1996/2006 en conformité avec le référentiel du BTS IRIS

[60] Norbert Kajler Iean-michel Viovy et l'equipe du CCSI V 1.22 , Programmation avec le langage Java - centre de calcul et des systemes d'information (octobre 2014).

[61] Cay S. Horstmann et Gary Cornell, Au cœur de Java™ volume 1 Notions fondamentales-2008.

[62] Hugues Bersini, Introduction à Java - Code/IRIDIA – Université Libre de Bruxelles, 2013.

[63] Solveig Vidal Ingénieur d'études CNRS, Visualisation de l'information - Un panorama d'outils et de méthodes Dossier De Synthèse Mai 2006.

[64] Hopcroft J. E. et Wong J. K., Linear Time Algorithm for Isomorphism of Planar Graphs, In Sixth ACM Symposium on Theory of Computing, 1974.

[Net01] http://jung.sourceforge.net/manual.html consulté le 02/02/2018.