الجمهورية الجزائرية الديموقراطية الشعبية

République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université de 8 Mai 1945 – Guelma -

Faculté des Mathématiques, d'Informatique et des Sciences de la matière

Département d'Informatique

Mémoire de Fin d'études Master

Filière : Informatique

Option : Systèmes informatique

Thème :

Reconnaissance des motifs dans des graphes EMF

Encadré Par :

Mr. BERREHOUMA Nabil

Présenté par :

GHEMARSSA Imene

Juillet 2019

Remerciement

"الشكر و الحمد لله"

Merci Allah de m'avoir donné la capacité d'écrire et de réfléchir, la force d'y croire, la patience d'aller jusqu'au bout du rêve.

Je tiens à remercier vivement mon encadreur, **Mr BerrehoumaNabil,** avoir encadré pour réaliser ce travail, pour ses précieux conseils et de m'avoir donné le meilleur de son savoir et aide.

Je remercie également les membres de jury de nous faire l'honneur de juger mon travail. Je remercie profondément toutes les **profs de département d'informatique** et toutes les personnes qui ont contribuées à l'élaboration de ce travail.

Finalement, je remercie nos grandes familles, mes amies, mes collègues de l'université **08 mai 1945** Guelma et toute la promotion **2019** de l'informatique.

Imane

Dédicace

Je dédie ce Travail :

A ***mon père*** *Allah yarhmou, qui m'a apporté et m'a appris et me sauver tout.*

A ***ma mère****, pour son soutien et encouragement durant mes études.*

Et mes frères Choukri, karim, Raouf.

Et mes sœurs Razika et son fils Ihssen ,Ilhem, Rahma .

A ma famille maternelle tantes, oncles, cousins et cousines

A mon fiancé S.Haroun.

Mes amies que je n'oublie jamais surtout :

Ahlem, Imene, Oum elkhir, Abir, Wissem, Dounia,

Enfin je le dédie à tous mes amies que je n'ai pas citées et à toute ma promo et collègues d'informatiques E8.

Résumé

Parmi les domaines de recherche et d'application importants de l'extraction de modèles de graphes, notons la recherche d'informations, la découverte de connaissances et l'exploration de données, la théorie mathématique des graphes, l'intelligence artificielle, la vision par ordinateur, la conception assistée par ordinateur, etc.

L'objectif de notre étude est d'établir un état de l'art sur les algorithmes de La reconnaissance des patterns dans les graphes, en s'intéressant sur les graphes qui sont considérés comme des modèles d'abstraction en génie logiciel. L'intérêt de l'utilisation des graphes réside dans leur capacité à représenter les résultats obtenus en théories des graphes. Implémenter un algorithme basé sur la théorie de correspondance des graphes à travers l'isomorphismes des graphes qui nécessite la préservation des structures des graphes.
Mots-clés : Graphe pattern Matching, EMF, MDE, algorithme d'Ullmann, graphe, isomorphisme.

Reconnaissance des Patterns dans les Graphes EMF

Ghemarssa Imene,Berrehouma Nabil

24 juillet 2019

Table des matières

Table des figures

Liste des tableaux

Introduction générale

En génie logiciel, Les modèles occupent une place primordiale tout au long le cycle de vie d'un logiciel. Par définition les modèles sont une abstraction de la réalité où seulement les propos essentiels sont retenus. autrement dit, un modèle n'est qu'un ensemble de concepts muni des relations entre ces concepts et éventuellement des contraintes.

Travailler avec des modèles nécessitent absolument des outils adéquats permettant de manipuler ces derniers depuis leur naissance jusqu'à leur destruction, ceci impliques plusieurs opérations telles que :

- la création des modèles en respectant les règles du domaines modélisé
- la persistance des l'échange des modèles selon des formats standards et interopérables
- la transformation des modèles d'un formalisme vers un autre tout en préservant les équivalences de certaines propriétés
- L'interrogation des modèles pour répondre à certaines questions

Parmi les outils qui sont élaboré pour répondre aux besoins, nous trouvons la plateforme EMF qui fait partie du monde "Eclipse". EMF offre une cadre agréable au modeleurs pour définir ce qu'on appelle un "Meta modèle" qui est considéré comme une grammaire -au sens langage de programmation et qui va servir plus tard à l'instantiation des modèles et leur validation. En se basant sur ce "méta modèle", cet outil permet d'une manière automatique de générer une gamme variée des utilitaires pour gérer les modèles instances.

Dans ce projet , nous nous intéressons à un seul besoin qui est "l'interrogation des modèles". En effet, nous allons considérer un modèle comme étant un graphe dont les sommets sont les concepts et les relations entre ces concepts sont les arcs ou les arrêtes. Puis nous formulant un requête sous formes d'un graphe et la réponse sera le sous graphe induit du modèle qui correspond au graphe requête. Pour atteindre cet objectif, nous avons fait une abstraction des modèles vers des graphes ceci nous a permis d'exploiter les résultats obtenus en théories des graphes et les projeter sur l'interrogation des modèles en génie logiciel.

Dans cette optique, nous nous sommes intéressés dans ce mémoire à l'étude de la théorie de correspondance des graphes connus communément par "La reconnaissance des pattern dans les graphes", et nous implémentons un algorithme très connu dans la littérature qui s'agit de l'algorithme "d'Ullmann" pour l'énumération des isomorphismes existants entre un graphe pattern et un graphes cibles.

Nous présentons notre travail d'une manière progressive en suivant l'organisation suivante :

Chapitre 1 Nous présentons les définitions fondamentaux relatives à la théorie des graphes

qui vont nous servir à présenter dans le chapitre 2 la théories des isomorphismes des graphes.

Chapitre 2 La reconnaissance des pattern est démontré à travers l'isomorphismes des graphes. un intérêt particulier sera porté sur l'algorithme d'Ullmann qui permet de concrétiser les notions théoriques présenté

Chapitre 3 après les deux premiers chapitres théoriques, nous arrivons ici à la présentation des modèles EMF à travers la présentation de leurs cadre globales qui s'agit de l'environnement Ellipse et la plateformes de modélisation EMF

Chapitre 4 le dernier chapitre consiste à associer la théorie et la pratique par l'application de l'algorithme d'Ullmann sur des graphes définis comme des modèles EMF. plusieurs expérimentation ont été effectué pour tester la faisabilité de cet algorithmes sur les graphes EMF.

Enfin, nous concluons notre mémoire par une conclusion générale où nous avons évoqués les problèmes que nous n'avons pas pu surmonter et proposons des perspectives pour la continuité de ce travail.

Chapitre 1

Introduction à la Théorie des Graphes

Sommaire

1.1 Introduction

Les graphes et algorithmes de reconnaissance des patterns constituent des outils puissants dans le processus de localisation et comparaison, en raison de leur efficacité et de leur grande utilité dans diverses domaines.

Un graphe est une représentation abstraite d'un ensemble d'objets (sommets / nœuds)reliées par des liens . Un graphe est un outil puissant pour la modélisation en génie logiciel car la plupart des diagramme utilisé peuvent être vue en tant que des graphes. Par conséquence, il est possible d'exploiter les résultats reconnus sur les graphes pour les projeter sur le domaine formaliser. c'est dans cette perspective que nous allons mener notre travail pour en exploiter les résultat obtenus sur la reconnaissance des pattern dans les graphes sur les modèles EMF discutés dans le chapitre suivant.

1.2 Définitions Fondamentales

Définition 1 (Graphe non orienté) *Un graphe non orienté G est un couple $G = (V, E)$ tel que [?] :*

- *V est un ensemble fini de sommets , on mentionne le non de graphe en cas de confusion $V(G)$*
- *E est un ensemble de couples non ordonnés de sommets $\{v_i, v_j\} \in V^2$, on mentionne le non de graphe en cas de confusion $E(G)$*

*Une paire $\{v_i, v_j\}$ est appelée **une arête**. On dit que les sommets v_i et v_j sont **adjacents**. l'arête $\{v_i, v_j\}$ est appelée arête **incidente** à v_i et v_j.*

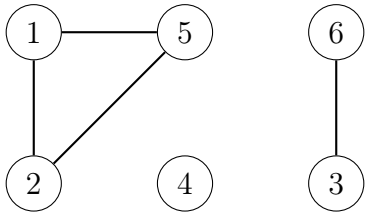


FIGURE 1.1 – Graphe non orienté

Exemple 1 *Le graphe de la figure 1.1 représente un graphe non orienté $G = (V, E)$ avec $V = \{1, 2, 3, 4, 5, 6\}$ et $E = \{\{1, 2\}, \{1, 5\}, \{5, 2\}, \{3, 6\}\}$.*

Définition 2 ***Une boucle** est une arête reliant un sommet à lui-même. Un graphe non-orienté est dit **simple** s'il ne comporte pas de boucle, et s'il ne comporte jamais plus d'une arête entre deux sommets. Un graphe non orienté qui n'est pas simple est **un multi-graphe**. Dans le cas d'un multi-graphe, E n'est plus un ensemble mais un multi-ensemble*[1] *d'arêtes.*

Définition 3

- *On appelle **ordre** d'un graphe le nombre de ses sommets, i.e c'est $|V|$.*

1. ensemble avec répétition

— *On appelle* ***taille*** *d'un graphe le nombre de ses arêtes, i.e c'est* $|E|$.

Définition 4 *Un graphe orienté* G *est un couple* $G = (V, E)$ *tel que :*

— V *est un ensemble fini de sommets*

— E *est un ensemble de couples ordonnés de sommets* $(v_i, v_j) \in V^2$

Un couple (v_i, v_j) *est appelée* ***un arc****, et est représenté graphiquement par* $v_i \to v_j$. v_i *est le* ***sommet initial*** *ou origine, et* v_j ***le sommet terminal*** *ou extrémité. L'arc* $a = (v_i, v_j)$ *est dit sortant en* v_i *et incident en* v_j *, et* v_j *est* ***un successeur*** *de* v_i*, tandis que* v_i *est* ***un prédécesseur*** *de* v_j *. L'ensemble des successeurs d'un sommet* $v_i \in V$ *est noté* $Succ(v_i) = \{v_j \in V | (v_i; v_j) \in E\}$. *L'ensemble des prédécesseurs d'un sommet* $v_i \in V$ *est noté* $Pred(v_i) = \{v_j \in V | (v_j; v_i) \in E\}$.

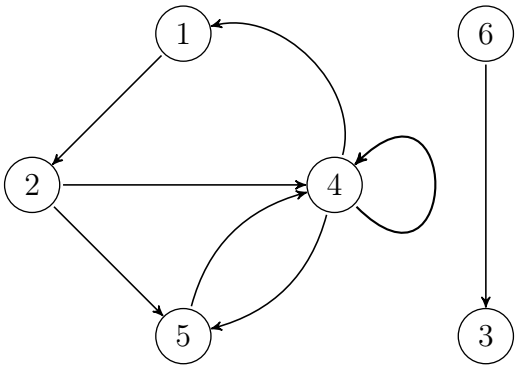


FIGURE 1.2 – Graphe orienté

Exemple 2 *Le graphe de la figure 1.2 représente un graphe orienté* $G = (V, E)$ *avec* $V = \{1, 2, 3, 4, 5, 6\}$ *et* $E = \{(1, 2), (2, 4), (2, 5), (4, 1), (4, 4), (4, 5), (5, 4), (6, 3)\}$.

Définition 5 *Si* G *est un graphe orienté, nous considérerons son graphe non orienté associé* G' *obtenu en enlevant l'orientation de chaque arc de* G*. Nous dirons alors que* G *est une orientation de* G'.

Définition 6 ***Un sous-graphe*** *de* $G = (V, E)$ *est un graphe* $H = (V, E)$ *avec* $V(H) \subseteq V(G)$ *et* $E(H) \subseteq E(G)$*. Nous dirons que* G *contient* H*. Le graphe* H *est un* sous-graphe induit *de* G *si* H *est un sous-graphe de* G *et si* $E(H) = \{(x, y) \in E(G) | x, y \in V(H)\}$. $H = G[V(H)]$ *est le sous-graphe de* G *induit par* $V(H)$*. Un sous-graphe* H *de* G *est appelé* ***couvrant*** *si* $V(H) = V(G)$.

Définition 7 *[?] Soit un graphe* G *et* $X, Y \subseteq V(G)$*. Nous poserons :* $E(X, Y) := \{\{x, y\} \in E(G) | x \in X \setminus Y, y \in Y \setminus X\}$ *si* G *est non orienté et* $E^+(X, Y) := \{(x, y) \in E(G) | x \in X \setminus Y, y \in Y \setminus X\}$ *si* G *est orienté. Si* G *est non orienté et* $X \subseteq V(G)$ *nous poserons* $\delta(X) := |E(X, V(G) \setminus X)|$. ***L'ensemble des voisins*** *de* X *est défini par* $\Gamma(X) := \{v \in V(G) \setminus X | E(X, \{v\}) \neq \emptyset\}$. *Si* G *est orienté et* $X \subseteq V(G)$ *nous poserons :* $\delta^+(X) := |E^+(X, V(G) \setminus X)|$, $\delta^-(X) := |\delta^+(X)(V(G) \setminus X)|$ *et* $\delta(X) := \delta^+(X) + \delta^-(X)$*. Nous utiliserons des indices (par exemple* $\delta_G(X)$*) pour spécifier le graphe* G*, si nécessaire. lorsque l'ensemble* X *est* ***singleton*** *i.e* $X = v$ *, nous écrirons* $\delta(v) := \delta(\{v\}), \Gamma(v) := \Gamma(\{v\}), \delta^+(v) := \delta^+(\{v\})$ *et* $\delta^-(v) := \delta^-(\{v\})$

— *Le degré d'un sommet v est $\delta(v)$, nombre d'arêtes incidentes à v.*
— *$\delta^-(v)$ respectivement $\delta^+(v)$ sont les nombres des arcs entrants respectivement sortants à un sommet v dans un graphe orienté. le degré de v est $\delta(v) = \delta^-(v) + \delta^+(v)$.*
— *Un sommet v de degré 0 est appelé **isolé**.*
— *Un graphe dont tous les sommets ont degré k est appelé **k-régulier**.*
— *Dans un graphe non orienté $\sum_{x\in V} \delta(x) = 2 * |E|$*
— *dans un graphe orienté $\sum_{x\in V} \delta^+(x) = \sum_{x\in V} \delta^-(x)$*

Définition 8 ***Un graphe complet*** *est un graphe simple non orienté tel que toute paire de sommets est adjacente. Le graphe complet à n sommets sera noté K_n.* ***Le complément*** *d'un graphe simple non orienté G est le graphe H tel que $G + H$ [2] est un graphe complet.*

1.3 Chaîne et cycle

1.3.1 Chaine

Une chaîne dans G=(V,E), est une suite ayant pour éléments alternativement des sommets et des arêtes, commençant et se terminant par un sommet, et telle que chaque arête est encadrée par ses extrémités. On note :

$$(x_0, e_1, x_1, ..., e_k, x_k,)$$

où k est un entier $\geqslant 0$, $x_i \in V$ pour $i = 0, ..k$ et $e_j \in E$ pour $j = 1, .., k$, et pour $i = 0, ..., k1$ x_i et x_{i+1} sont les extrémités de e^{i+1}. L'entier K est la longueur de la chaîne.

1.3.2 Cycle

Un cycle est une chaîne de longueur $\geqslant 1$ simple et fermé. C'est donc une suite de la forme :

$$(x_0, e_1, x_1, ..., e_k, x_0,)$$

Où : k est un entier $\geqslant 1$, les x_i et les e_j étant définis comme précédemment. L'entier k est la longueur du cycle. Un cycle est élémentaires si les x_i pour $i = 0, \ldots, k-1$ sont distincts deux à deux. Tout cycle se décompose en cycle élémentaires deux à deux disjoints relativement aux arrêtes.

1.3.3 Distance

Dans un graphe G la distance entre deux sommets x et y est la plus petite longueur des chaînes qui relient x et y dans G. S'il n'y a pas de chaîne reliant x et y la distance est parfois posée égale à l'infini. En note $d(x, y)$ la distance entre les sommets x et y, on a les propriétés classique suivantes : $d(x, x) = 0$ pour tous sommets x, $d(x, y) = d(y, x)$ quels que soient les

2. Si G et H sont deux graphes, $G + H$ est le graphe tel que $V(G + H) = V(G) \cup V(H)$ et tel que $E(G + H)$ est l'union disjointe de $E(G)$ et $E(H)$.

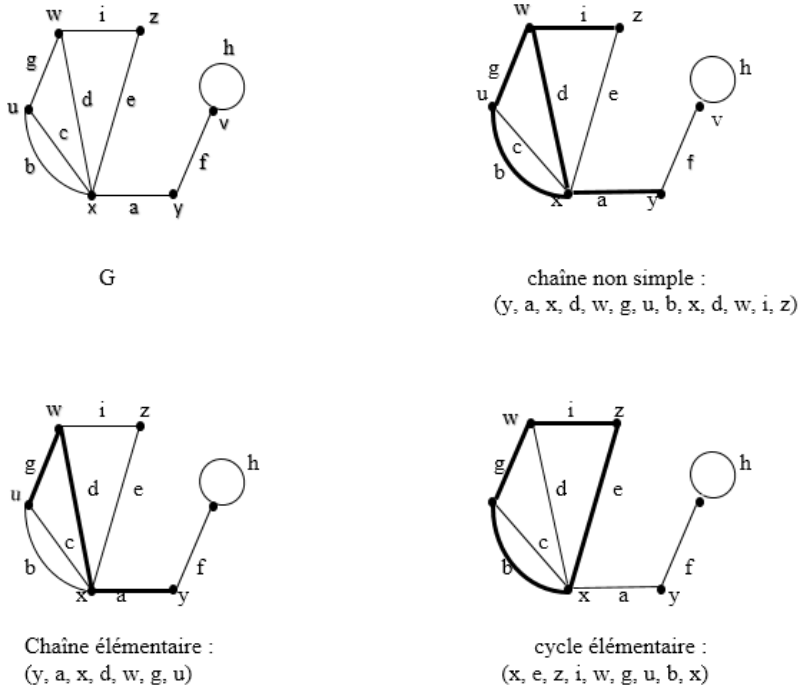


FIGURE 1.3 – Chaines et Cycles

sommets x et y, et également l'inégalité triangulaire : $d(x, y) \leq d(x, z) + d(z, y)$ quels que soient x, y, z.

1.3.4 Diamètre, maille et cocycle

Le diamètre d'un graphe G est la plus grande distance entre deux sommet du graphe. La Maille est la plus petite longueur d'un cycle du graphe.

Cocycle d'un graphe $G = (V, E)$ est l'ensemble des arrêtes de G de la forme xy où $x \in U$ et $y \in X \setminus U$, où $U \subseteq X$ tel que $U \neq \Phi$ et $U \neq X$.

1.4 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 :

1.4.1 Graphes simples

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

1.4.2 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.

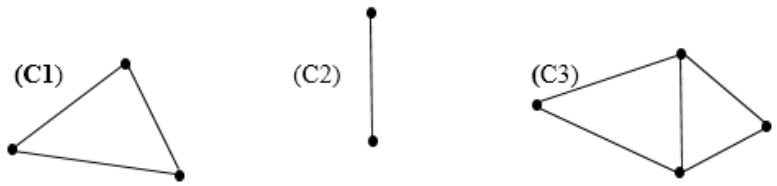


FIGURE 1.4 – Un graphe non connexe et ses trois composantes connexes : C1, C2, C3

FIGURE 1.5 – Le graphe biparti complet k 3,3

1.4.3 Graphes connexes et connexité

Connexité Un graphe G connexe *si* pour toute paire de sommets $x, y \in V$, il existe une chaine entre x et y : on dit alors que les sommets x et y sont connectés. Les composantes connexe d'un graphe G sont les sous-graphe induits.

1.4.4 Graphe planaire

Un graphe est dit planaire s'il existe un plongement dans le plan sans croisement d'arêtes. Dans laquelle deux lignes arêtes ne se rencontrent pas, en dehors éventuellement d'une extrémité commune.

1.4.5 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. On note un graphe biparti $G = (X, Y, E)$ ou X et Y sont les deux classes (et par conséquent $X \cup Y$ est l'ensemble des sommets), E est l'ensemble des arêtes.

- Un graphe biparti est un graphe dont l'ensemble de sommets peut être partagé en deux stable.
- Un graphe k-parti, est un graphe dont l'ensemble de sommets peut partager en k stables.
- Un graphe biparti $G = (X, Y, E)$ est dit complet si $E = xy | x \in X, y \in Y$.
- Un graphe biparti n'a de boucles car une boucle contredirait l'hypothèse qu'une arête a ses extrémités dans classes différentes.

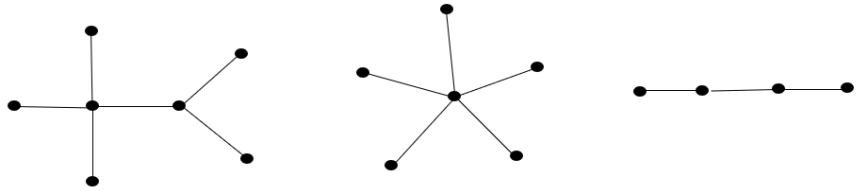


FIGURE 1.6 – Exemples d'arbres

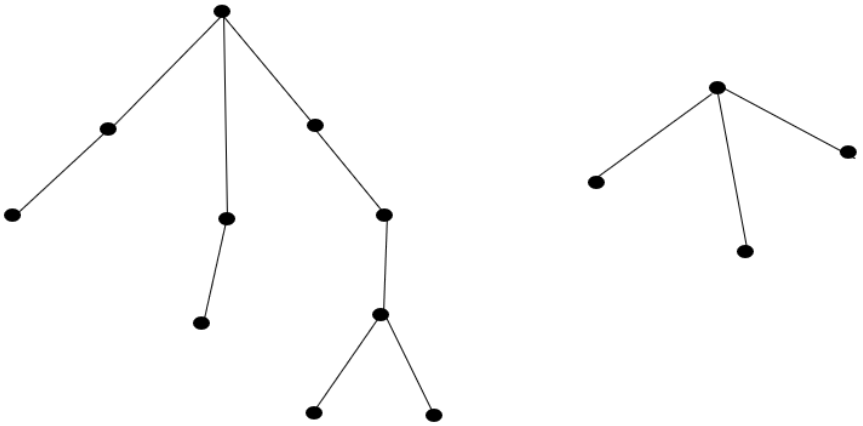


FIGURE 1.7 – Foret comportant deux arbres

1.4.6 Graphe étiqueté

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

1.4.7 Graphe partiel

Soit $G = (E, V)$ un graphe. Le graphe $G1 = (E1, V1)$ est un graphe partiel de G, si $V1$ est inclus dans V. Autrement dit, on obtient $G1$ en enlevant une ou plusieurs arêtes du graphe G.

1.5 Arbres et forêts

Un arbre est un graphe connexe et acyclique[3] , Ainsi un arbre est nécessairement simple, une chaîne élémentaire est en particulier un arbre. Les arbres ont des propriétés, on distingue toujours par nG et mG ou n le nombre de sommets d'un arbre et m son nombre d'arrêtes.

Forêts Une forêt est un graphe acyclique (donc nécessairement simple). Un arbre est donc une forêt ainsi les composantes connexes d'une forêt sont des arbres. Une feuille dans une forêt est un sommet de degré égal à 1.

3. Acyclique : c'est-à-dire un graphe sans cycle.

FIGURE 1.8 – un arbre couvrant d'un graphe (en gras)

1.5.1 Caractérisation des arbres

soit $G = (V, E)$ un arbre, alors :

1. G est acyclique et on a $|V| - 1$ arêtes
2. G est un graphe connexe minimal (chaque arête est un isthme [4])
3. G est un graphe maximal sans cycles (l'addition d'une arête quelconque crée un cycle),
4. Toute paire de sommets de G est connectée par une chaine unique.

1.5.2 Arbres couvrants d'un graphe

un arbre couvrant d'un graphe $G = (V, E)$ est un graphe $A(V', E')$ partiel de G qui est un arbre et dont $V' = V$.

1.6 Parcours de graphes

en vue de leurs traitement, les graphes sont représenté par 3 structures de données différents, chaque représentation est adapté à un type de traitement particulier

1. La matrice d'adjacence,
2. Les listes des voisins ou successeurs,
3. La liste des arêtes.

1.6.1 Matrice d'adjacence

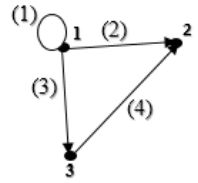
Dans un graphe $G = (V, E)$ d'ordre n soit $V = v_0, v_1, .., v_{n-1}$. La matrice d'adjacence de $M(G) = (m_{ij}$ où $i, j = 0..n - 1$ consiste en une matrice booléenne carrée de taille $n \times n$ telle que

$$m_{ij} = \begin{cases} 1 \; si \; v_j \; est \; un \; voisin \; de \; v_i \\ 0 \; sinon \end{cases}$$

Dans le cas de graphes non orientés, la matrice est symétrique par rapport à sa diagonale descendante. Dans ce cas, on peut ne mémoriser que la composante triangulaire supérieure de la matrice d'adjacence.

une matrice d'adjacence possède les propriétés suivantes :

4. un isthme est une arrête dont la surpression rend le graphe non connexe



Graphe orienté
(Numéros d'arcs entre parenthèse)

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

Matrice d'adjacence

Arc	Couple des extrémités	
1	1	1
2	1	2
3	1	3
4	3	2

Liste des arcs

FIGURE 1.9 – divers représentation de graphes

1. C'est une matrice carré
2. Si il existe une relation d'ordre sur les sommets, alors la matrice d'adjacence est unique
3. Les éléments de diagonale $m_{ii} = 0$ sauf s'il y a une boucle alors $m_{ii} = 1$
4. Elle est symétrique : $m_{ij} = m_{ji}$ pour les graphes non orienté

1.7 Compléxité d'algorithmes de graphes

L'exécution des algorithmes nécessite deux ressources importantes en l'occurrence le temps et l'espace mémoire. Dans le cadre de ce chapitre, nous nous intéresserons par la complexité des problèmes décidables. Les problèmes indécidables n'ont pas des algorithmes. La complexité temporelle d'un algorithme est le nombre d'étapes nécessaires pour recoudre un problème de taille n. La complexité est défini souvent au pire cas. Nous cherchons souvent les bornes asymptotiques du nombres d'étapes de réalisation mais pas le nombre exact. Nous utilisons pour ce propos la notation *Big-O* [**?**].

Définition 9 *Un algorithme à une complexité $f(n) = O(g(n))$ si il existe des constantes n_0 et c tel que $\forall n > n_0, f(n) \leqslant c \cdot g(n)$*

Nous disons que la fonction $f(n)$ est majorée par la fonction $g(n)$.

Définition 10 *Un algorithme est dit de **complexité polynomiale** ou simplement polynomial si sa complexité est $O(p(n))$, où $p(n)$ est un polynôme de la variable n.*

Définition 11 *Un algorithme est dit de **complexité exponentielle** ou simplement exponentiel si sa complexité est $O(c_n)$, où c est une constante réel strictement supérieur à 1.*

La table 1.1 montre l'évolution du temps d'exécution d'un algorithmes pour différentes complexités en fonction de la taille d'entrée n.

Complexity	Size = 10	Size = 20	Size = 30	Size = 40	Size = 50
$O(n)$	0.00001 s	0.00002 s	0.00003 s	0.00004 s	0.00005 s
$O(n^2)$	0.00001 s	0.00004 s	0.00009 s	0.000016 s	0.000025 s
$O(n^5)$	0.01 s	0.32 s	24.3 s	1.7 mn	5.2 mn
$O(2^n)$	0.001s	1.0 s	17.9 mn	12.7 jours	35.7 ans
$O(3^n)$	0.059 s	58.0 mn	5.6 ans	3855 siècles	$2 \cdot 10^8$ siècles

TABLE 1.1 – Temps d'exécution en fonction du taille n du problème pour les différentes complexité

1.8 Complexité d'un problème

La complexité d'un problème est équivalente à la complexité du meilleur algorithme pouvant résoudre ce problème. Un problème est dit **calculable** si il existe un algorithme polynomiale qui peut la résoudre. un problème est dit *non calculable ou difficile* si il n'existe pas un algorithme polynomial qui le résout.

La théorie de la calculabilité est un domaine de la logique mathématique et de l'informatique théorique. La calculabilité cherche d'une part à identifier **les classes de complexité des problèmes** calculables/non calculables. d'autre part à appliquer ces concepts à des questions fondamentales des mathématiques. Une bonne appréhension de ce qui est calculable et de ce qui ne l'est pas permet de voir les limites des problèmes que peuvent résoudre les ordinateurs. Une classe de complexité est un ensemble de problème ayant besoin de même quantité de ressources. il existe 3 grandes classes de complexité

1. La classe P des problèmes polynomiaux.
2. La classe NP des problèmes non déterministement polynomiaux.
3. La classe des problèmes NP-complets, dits encore difficiles.

1.9 Conclusion

Ce chapitre était pour objectif de préparer le lecteur à se familiariser avec les différentes notions et définitions relatives à la théorie des graphes. Dans le prochain chapitre, Nous nous concentrerons sur le problème d'isomorphismes des graphes qui est l'objet de notre étude.

Chapitre 2

Isomorphismes des Graphes

Sommaire

2.1 Introduction

Dans le monde de la modélisation , il y a toujours un problème pour trouver un pattern dans un modèle de taille importante. Tout modèle peut être considéré comme un graphe dont les sommets sont les concepts et les arrêtes sont les relations entre ces concepts. La solution de ce problème revient à trouver un algorithme efficace permettant d'identifier les isomorphisme entre le graphe pattern et le graphe cible qui est le modèle même. Dans ce chapitre, nous allons s'intéresser à la formulation du problème d'isomorphisme de graphe et en analysant un algorithme reconnu dans la littérature qui donne des solutions satisfaisantes de ce problème.

2.2 Graph Matching

un graph matching exact est une correspondance entre les sommets de deux graphes G_1 et G_2 de telle manière que s'il existe une arête/ arc entre deux sommets dans le premier graphe alors les deux sommets correspondants dans le deuxième graphs sont aussi reliés par une arête/arc. Plusieurs variante du graph matching existent (isomorphism, isomorphism de sou-graphes, monomorphism, homomorphism, sougraphe isomorphe maximal)

Définition 12 *soit deux graphes $G_1 = (V_1, E_1)$ et $G_2 = (V_2, E_2)$.* ***Un Mapping*** *est une fonction $\mu : V_1 \to V_2$. Un Mapping est dit* ***arête/arc préservant*** *ou* ***Homomorphism*** *si et seulement si :*

$$\forall \upsilon, \omega \in V_1, (\upsilon, \omega) \in E_1 \Rightarrow (\mu(\upsilon), \mu(\omega)) \in V_2 \vee \mu(\upsilon) = \mu(\omega)$$

Définition 13 ***Un Monomorphisme*** *est un homomorphisme μ qui est injective*

$$\forall \upsilon \neq \omega \in V_1, \mu(\upsilon) \neq \mu(\omega)$$

Définition 14 ***Un isomorphisme*** *de graphe est un monomorphisme μ bijective avec son mapping inverse μ^{-1} est un monomorphisme*

$$\begin{cases} \forall \nu_2 \in V_2, \exists \nu_1 \in V_1 : \nu_1 = \mu^{-1}(\nu_2) \\ \mu_1 \; est \, un \, monomorphism \end{cases}$$

autrement dit deux graphes $G_1 = (V_1, E_1)$ et $G_2 = (V_2, E_2)$ sont **isomorphes** s'il existe une bijection μ entre V_1 et V_2 tel que pour chaque pair de sommets $i, j \in V_1, (i, j) \in E_1$ si et seulement si $(\mu(i), \mu(j)) \in E_2$. nous allons dénoter $G_1 \equiv G_2$ lorsque G_1 et G_2 sont isomorphes. $Iso(G_1, G_2)$ désigne l'ensemble de tous les isomorphismes entres G_1 G_2.

Exemple 3 *Les graphes des figures 2.1 et 2.2 sont isomorphes. Un isomorphisme entre G_1 et*

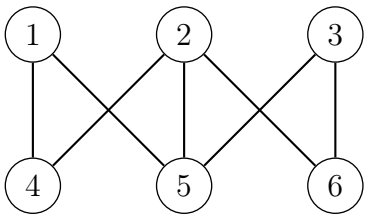


FIGURE 2.1 – G_1

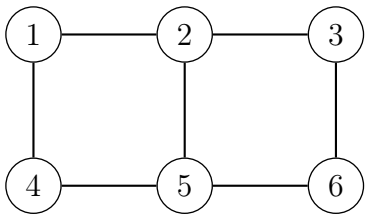


FIGURE 2.2 – G_2

G_2 est peut être présenté par la matrice suivante :

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 5 & 3 & 4 & 2 & 6 \end{bmatrix}$$

où la première ligne consiste aux sommets de G_1 et la deuxième ligne aux sommets de G_2 . nous allons voir dans la suite une autre représentation matricielle d'un isomorphisme.

Définition 15 (Graph automorphism) *Un automorphisme d'un graphe $G = (V, E)$ est un isomorphisme entre le graphe G et le graphe $G^{'} = (\varphi(V(G)), E(G))$ où φ est une permutation des sommets de G*

2.3 Problème d'Isomorphismes de Graphes

Étant donné deux graphes g_1 et g_2. Le problème d'isomorphisme de graphes peut être recomposés en deux sous problèmes :

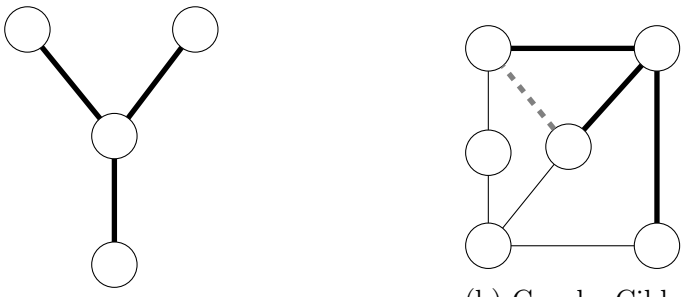
1. est ce que les deux graphes sont isomorphes
2. énumérer tous les isomorphismes entre ses deux graphes

2.4 Isomorphisme de sous graphes

Le problème d'isomorphisme de sous-graphes consiste à trouver un isomorphisme entre un graphe qu'on appelle *graphe pattern* $G_\alpha = (V_\alpha, E_\alpha)$ et un sous-graphe d'un graphe qu'on appelle *graphe cible* $G_\beta = (V_\beta, E_\beta)$ [**?**]

2.5 Algorithme d'Ullmann

L'un des algorithmes les plus reconnu pour le problème d'isomorphismes de sous graphe est celui *d'Ullmann* [**?**]. L'algorithme est basée sur l'exploration de l'arbre représentant l'espace des solutions possibles et utilise des heuristiques pour réduire sa taille en supprimant certaines branches de l'arbre qui ne mènent pas vers des solutions. Cette technique est connu dans literature par *The Branch and Bound Heursitic*. 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



(a) Graphe Pattern

(b) Graphe Cible

FIGURE 2.3 – un isomorphisme de graphe non induit, le graphe pattern est un sous graphe du graphe cible sauf que l'arrête en pointillé n'existe pas dans le graphe pattern. [**?**]

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. Dans ce qui suit nous présentons progressivement cet algorithme.

2.5.1 Préliminaires

L'entrée de l'alogorithme d'Ullmann consiste à deux graphes [1]

1. $G_\alpha = (V_\alpha, E_\alpha)$: qui s'appelle graphe pattern
2. $G_\beta = (V_\beta, E_\beta)$: qui s'appelle graphe cible

Sans perte de généralisation, nous donnons des labels aux sommets des deux graphes : $V_\alpha = \{\alpha_1, \alpha_2, .., \alpha_n\}$, et $V_\beta = \{\beta_1, \beta_2, .., \beta_m\}$, où $n = |G_\alpha|$ et $m = |G_\beta|$ et ils vérifient que $n \leq m$.

L'isomorphisme entre G_α et G_β est une fonction injective $\mu : V_\alpha \to V_\beta$ vérifiant la condition suivante :

$$\forall \alpha_i, \alpha_j \in V_\alpha, (\alpha_i, \alpha_j) \in E\alpha \Rightarrow (\mu(\alpha_i), \mu(\alpha_j)) \in E_\beta \tag{2.1}$$

d'une manière similaire, un isomorphisme de sous graphes induit entre G_α et G_β est une fonction injective $\mu : V_\alpha V_\beta$ vérifiant la condition suivante :

$$\forall \alpha_i, \alpha_j \in V_\alpha, (\alpha_i, \alpha_j) \in E\alpha \Longleftrightarrow (\mu(\alpha_i), \mu(\alpha_j)) \in E_\beta \tag{2.2}$$

La figure 2.3 illustre la différence entre l'isomorphisme de graphe et de sous graphe induit l'output de l'algorithme sera l'un des 3 variantes :

1. l'existence ou l'absence d'un isomorphisme content.
2. le premier sous graphe isomorphe au graphe pattern.
3. l'énumération de tous les sous graphes isomorphes au graphe pattern.

1. Dans ce mémoire , nous avons considéré que des graphes non orienté ; la généralisation vers les graphes orienté est tout à fait possible

2.5.2 Structures de Données et Matrice Candidate

Dans cet algorithme, un isomorphisme de sous graphe $\mu : V_\alpha \to V_\beta$ est représentée par une matrice binaire M d'une taille $n \times m$. Les lignes de la matrices M correspondent aux sommets du graphe pattern $\alpha_i \in V_\alpha$ et les colonnes correspondent aux sommets du graphe cible $\beta_j \in V_\beta$. Les éléments de la matrice M sont définis comme suite :

$$m_{ij} = \begin{cases} 1 \; si \; \mu(i) = j \\ 0 \; sinon \end{cases}$$

La matrice M représente un isomorphisme de graphe si dans chaque ligne il y a un et un seul 1 et dans chaque colonne il y au plus au 1. c'est à dire

$$\sum_{i=1}^{n} m_{ij} = 1 \; and \; \sum_{j=1}^{m} m_{ij} \leq 1 \tag{2.3}$$

Une matrice candidate ou matrice de compatibilité M^0 est une matrice binaire qui définit les mappings possibles pour un sommet α_i du graph pattern dans les sommets β_j du graph cible. Dans cette matrice , une ligne peut contenir un ou plusieurs 1 .

$$m^0_{ij} = \begin{cases} 1 \; si \; \delta_{G_\alpha}(i) \leq \delta_{G_\beta}(j) \\ 0 \; sinon \end{cases}$$

2.5.3 Déroulement de l'Algorithme

L'algorithme d'Ullmann génère la matrice M représentant un isomorphisme de sous-graphe à partir de la matrice M^0 qui est la racine de l'arbre représentant l'esapce de recherche , en utilsant une méthode de *Backtracking* ou retour en arrière. l'exploration de l'arbre de recherche se fait en profondeur. Un pas en avant représente une transition de M^d à M^{d+1}, et un pas en arrière représente une restauration de M^{d-1}. A chaque étape, l'algorithme sélectionne une paire compatible non encore traitée $(i; j)$ tel que $i \in V_\alpha \; and \; j \in C_d(i)$ où

$$C_d(i) = \{j \in V_\beta | m^d_{ij} = 1\}\}$$

Est l'ensemble des candidats de i. En utilisant i et j, l'algorithme génère une nouvelle matrice M^{d+1} en utilisant des opérations de *filtrage* et de *raffinement* (décrites ci-dessous). Les deux opérations réduisent le nombre de paires compatibles, plus exactement, les matrices M^d et M^{d+1} sont conformes à la condition

$$m^d_{ij} = 0 \Longrightarrow m^{d+1}_{ij} = 0$$

Nous constatons qu'une matrice candidate en profondeur d M^d ne peut pas conduire à un isomorphisme de sous graphe si elle contient une ligne i ne contenant aucun 1 (aucun sommet

cible candidat existe, c'est-à-dire, $C_d(i) = \Phi$). Lorsque tous les candidats sont traités, l'algorithme effectue un retour arrière. De cette façon, l'algorithme produit une suite de matrice $M^0 M^1, ..M^n = M$ où la dernière matrice M représente un isomorphisme de sous-graphe de G_α à G_β.

2.5.4 Algorithme d'énumération triviale pour l'isomorphisme de sous-graphes

Dans cette section , nous présentons la version initiale sans optimisation de l'algorithme d'Ullmann consistant à énumérer toutes les mapping possibles et faire dégager parmi lesquelles les isomorphismes éventuelles. Évidemment , la complexité de cet algorithme est de l'ordre $O(c^{n \times m})$ est le temps d'exécution va s'exploser rapidement.(comme nous allons montrée dans la suite). mais il s'agit d'un point de départ pour les autres version optimisées.

Les nombres de noeuds et des arêtes de G_α etG_β sont p_α, q_α et p_β, q_β, respectivement. Les matrices d'adjacence de G_α et G_β sont $A = [a_{ij}]$ et $B = [b_{ij}]$, respectivement.

Soit $M^{'} = [m^{'}_{ij}]$, une matrice de $p_\alpha \times p_\beta$ 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 lignes et les colonnes de B pour produire une matrice supplémentaire C.

$$C = [c_{ij}] = M^{'}(M^{'}B)^{\top}$$

où $\top$ indique la transposée.

La matrice $M^{'}$ présente une isomorphisme si la condition suivante est vérifiée :

$$\underset{1 \leq i \leq p_\alpha}{\forall i} \quad \underset{1 \leq j \leq p_\beta}{\forall j} \quad a_{ij} = 1 \Longrightarrow c_{ij} = 1 \tag{2.4}$$

Dans ce cas là, si $m^{'}_{ij} = 1$ alors le sommet i du graphe pattern correspond au sommet j du graphe cible pour l'isomorphisme représenté par la matrice $M^{'}$.
Au début de l'alogrithme d'énumération triviale, nous initialisons la matrice $M^{'} = M^0$. Une fois l'initialisation est faite , l'algorithme générera toutes les matrices $M^{'}$ possibles tel que pour chaque $m^{'}_{ij}$ de $M^{'}$,$m^{'}_{ij} = 1 \Longrightarrow m^0_{ij} = 1$. Pour chaque matrice $M^{'}$ généré , l'algorithme test si elle représente un isomorphisme en appliquant la condition 2.4 Le mécanisme de génération des matrices $M^{'}$ consiste à changer pour chaque ligne tous les 1 à 0 sauf un seul pour en vérifier la condition 2.3.

Dans l'arbre de recherche, la racine est bien la matrice M^0 et les feuilles sont les matrices $M^{'}$. la profondeur de l'arbre est $d = p_\alpha$. Chaque nœud intermédiaires au profondeur $d < p_\alpha$ dans l'arbre correspond à une matrice $M \neq M^0$ la différence est dans les premiers d lignes. Dans ce qui suit nous présentons une version itérative et une autre récursive

Énumération triviale itérative

La version itérative utilise les variables suivants :

— $F = [F_1, F_2, .., F_{p_\beta}]$ est un tableau de p_β élément boolean pour recenser les colonnes utilsés, $F_i = 1$ signifie que la colonne i est utilisé.

— $H = [H_1, H_2, .., H_{p_\alpha}]$ de p_α élement de type entier pour enregistre le colonne retenu au profondeur d c'est à dire $H_d = k$ signifie qu'au profondeur d la colonne k est retenu.

Algorithm 1 Trivial Iso Enumeration

1: $M \leftarrow M^0$, $d \leftarrow 1$, $H_1 \leftarrow 1$ ▷ Step 1
2: **for** $i = 1.., p_\alpha$ **do**
3: $F[i] \leftarrow 0$
4: **end for**
5: **if** $\nexists j \mid m_{dj} = 1 \wedge F_j = 0$ **then** ▷ Step 2
6: **go to** 34
7: **end if**
8: $M^d = M$
9: **if** $d = 1$ **then**
10: $k \leftarrow H_1$
11: **else**
12: $k \leftarrow 0$
13: **end if**
14: $k \leftarrow k + 1$ ▷ Step 3
15: **if** $m_{dk} = 0 \vee F_k = 1$ **then**
16: **go to** 14
17: **end if**
18: **for all** $j \neq k$ **do**
19: $m_{dj} = 0$
20: **end for**
21: **if** $d < p_\alpha$ **then** ▷ Step 4
22: **go to** 32
23: **else**
24: Vérifier si M est un isomorphisme en appliquant la condition 2.4
25: Sortir si isomorphisme est trouvé
26: **end if**
27: **if** $\nexists j > k \mid m_{dj} = 1 \wedge F_j = 0$ **then** ▷ Step 5
28: **go to** 34
29: **end if**
30: $M = M_d$
31: **go to** 14
32: $H_d \leftarrow k$, $F_k \leftarrow 1$, $d \leftarrow d + 1$ ▷ step6
33: **go to** 5
34: **if** d=1 **then** ▷ Step 7
35: Terminer l'algorithme
36: **end if**
37: $F_k \leftarrow 0$, $d \leftarrow d - 1$, $M = M_d$, $k \leftarrow H_d$

Énumération Triviale Récursive

La version origiale de l'algorithme est itérative comme présenté dans la section précédente. Tandis que une version récursive équivalente et plus simple pour la compréhension existe , et

mérite notre attention.

Algorithm 2 Enumeration Trivial Récursive

```
1: procedure TROUVER ISOMORPHIMSE(M, d)
2:     if d = p_α then
3:         Vérifier si M est un isomorphisme en appliquant la condition 2.4
4:         Sortir si isomorphisme est trouvé
5:     else
6:         M^d = M
7:         for all j = 1.., p_β do
8:             if m_dj = 1 then
9:                 for all k = 1.., p_β ∧ k ≠ j do
10:                    m_dk = 0
11:                    Trouver Isomorphimse(M,d+1)
12:                    M ← M^d
13:                end for
14:            end if
15:        end for
16:    end if
```

2.6 Filtrage ou Construction de la Matrice Candidate

Il est très clair que l'algorithme d'Ullman doit être initialisé par la matrice M^0.L'algorithme suivant construit cett matrice à partir des deux graphes pattern G_α et cible G_β.

Algorithm 3 Construction de la Matrice Candidate

```
1: procedure CONSTRUIRE M^0(G_α, G_β)
2:     for all i = 1..p_α do
3:         for all j = 1..p_β do
4:             if δ_{G_α}(α_i) ≤ δ_{G_β}(β_j) then
5:                 m^0_ij = 1
6:             else
7:                 m^0_ij = 0
8:             end if
9:         end for
10:    end for
```

2.7 Algorithme d'Ullmann avec Raffinement

Pour réduire le temps d'exécution nécessaire pour trouver les isomorphisme de sous graphes, le recours à des heuristique est inévitable tenant compte que la complexité de cet algorithme est exponentielle. Cet heuristique consiste à utiliser une procédure qui consiste à éliminer certains 1 dans les matrices M et par conséquence supprimer certaines branches de l'arbre de recherche.

Afin d'expliquer cette procédure, nous supposons $\beta_j = \mu(\alpha_i)$ et considérant les ensembles $\Gamma(\alpha_i) = \{\alpha_{i_1}, \alpha_{i_2}..\alpha_{i_x}\}$ et $\Gamma(\beta_j) = \{\beta_{j_1}, \beta_{j_2}..\beta_{i_y}\}$ les voisins respectivement de α_i et β_i dans

respectivement le graphe pattern G_α et le graphe cible G_β . D'après la notion d'isomorphisme de sous graphe, il est nécessaire que pour chaque élément de $\Gamma(\alpha_i)$ il existe un élément dans $\Gamma(\beta_j)$ tel que $\mu(\alpha_{i_x}) = \beta_{j_y}$. Si c'est le cas, alors l'élément de la matrice M qui correspond à ce mapping $\alpha_{i_x} \to \beta_{j_y}$ doit être égale à 1. par conséquence , nous pouvons formuler la condition de raffinement comme suite :

$$\underset{1\leq x\leq p_\alpha}{(\forall x)}\ ((a_{ix} = 1) \Longrightarrow \underset{1\leq y\leq p_\beta}{(\exists y)}\ (m_{xy} \cdot b_{yj} = 1)) \tag{2.5}$$

La procédure de raffinement teste simplement pour chaque 1 dans la matrice M si la condition 2.5 est vérifiée. Pour chaque $m_{ij} = 1$ où la condition 2.5 est non vérifiée, m_{ij} est changé à 0. Une telle changement peut entrainer le changement de la condition pour d'autre 1 dans la matrice. alors, il faut appliquer la vérification encore une autre fois. La vérification continu jusqu'à ce que aucune changement ne se produit.

Durant la procédure du raffinement, nous testons dans chaque itération si une ligne i de la matrice M est entièrement nulle. c'est à dire

$$\sum_{1\leq j\leq p_\beta} m_{ij} = 0$$

Ceci signifie que le sommet α_i du graphe cible n'a aucun candidat dans le graphe cible et par conséquence , il est impossible de trouver un isomorphisme. Donc il faut arrêter la recherche.

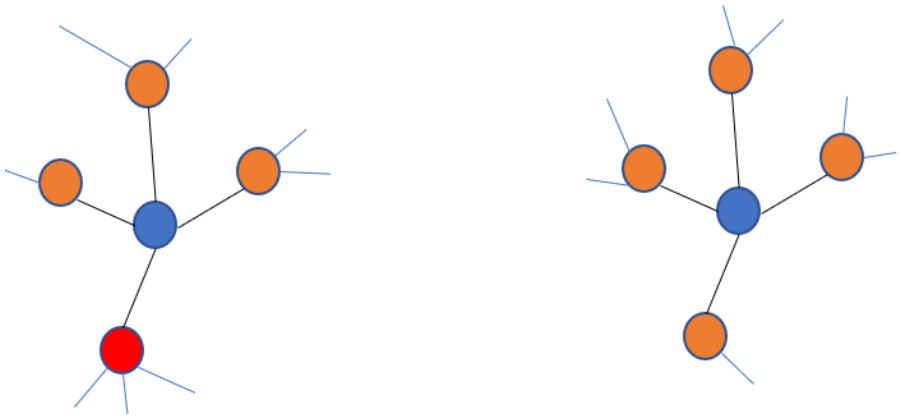


FIGURE 2.4 – Le sommets en bleu ne sont pas compatibles, parce que l'un des voisins (le sommet rouge) du graphe pattern n'a aucun sommet compatible dans les voisin dans le graphe cible

En faisons appel à la procédure de raffinement , l'algorithme d'isomorphisme d'Ullmann devient comme suite :

Algorithm 4 Raffinment

```
1: procedure RAFFINER(M, A, B)
2:     while vrai do
3:         for all m_ij = 1 do
4:             for all α_x ∈ Γ(α_i) | a_ix = 1 do
5:                 if ∄β_y ∈ Γ(β_j) | m_xy • b_yj = 1 then
6:                     Paire incompatible
7:                     m_ij = 0
8:                 end if
9:             end for
10:        end for
11:        if ∃i | Σ_{1≤j≤p_β} m_ij = 0 then
12:            La matrice M ne conduit à aucun Isomorphisme
13:        end if
14:        if pas de modification dans l'itération en cours then
15:            Terminer
16:        end if
17:    end while
```

2.8 Conclusion

Durant ce chapitre, nous avons focalisé notre attention sur le problème d'isomorphisme des sous graphe. Nous avons commencer par l'introduction du problème puis nous avons présenté un algorithme très connu dans la littérature s'agissant de celui d'Ullmann. plusieurs version sont discuté et des amélioration ont été proposé pour cet algorithme. Dans la suite de travail, nous allons implémenter cet Algorithme pour l'utiliser à la reconnaissance des pattern dans nos modèle EMF.

Chapitre 3

La Plateforme de Modélisation EMF et ses Outils

Sommaire

3.1 Introduction

Après la présentation de la théorie des graphes et la discussion du problème d'isomorphismes des sous graphes dans les deux premiers chapitres . Nous commençons à partir de ce chapitre de présenter leurs concrétisation dans le monde du génie logiciel.

3.2 Eclipse Modeling Framework

Eclipse Modeling Framework EMF est une plateforme de modélisation et de génération automatique du code utilisé pour concevoir et implémenter des outils logiciels et des applications suivant une approche pragmatique favorisant la manipulation des modèles à travers des transformation ou raffinement successives jusqu'au l'aboutissement au code final. En effet, à partir d'un modèles décrit en XMI, EMF produit un ensemble de classes java pour chaque concept défini dans le modèle avec une suite d'autres classe utilitaire permettant d'éditer et de visualiser le modèle. en plus de ses capacité de génération du code EMF est la base des plusieurs autres outils de modélisation.

3.3 Modélisation avec EMF

Dans ce qui suit nous détailleront EMF [?] à travers le parcours des étapes conduisant à la réalisation d'un outil de modélisation des graphes.
La première étape consiste à créer un Projet de modélisation EMF dans le Workspace du IDE Eclipse comme montrer dans la figure 3.1

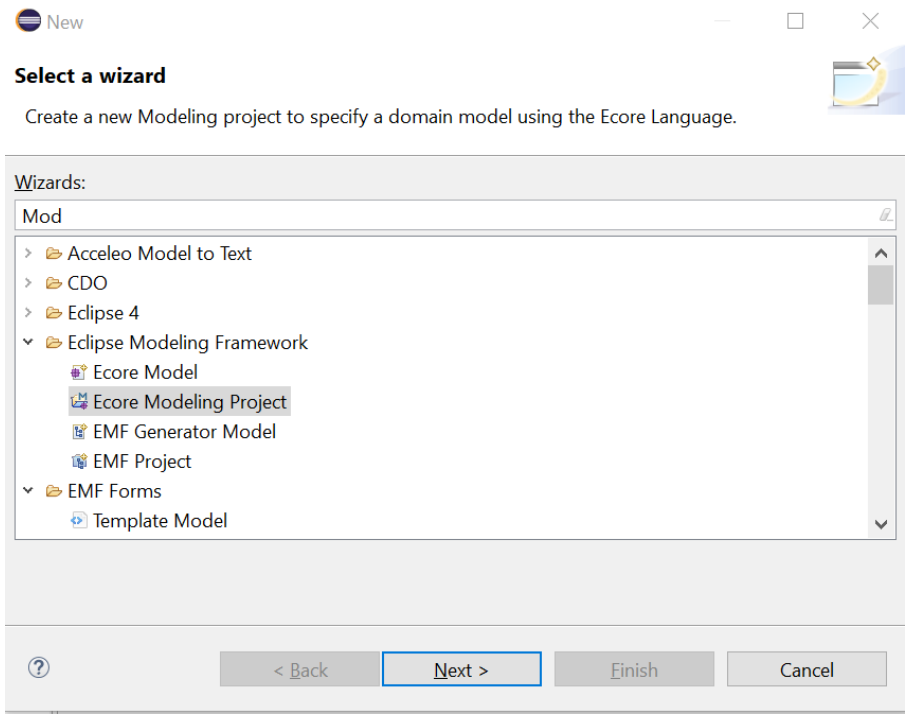


FIGURE 3.1 – Nouveau Projet de modélisation EMF

Ensuite, cliquer sur "Next" et choisir un nom pour le projet "dz.univ-guelma.dep-inf.graph" [1]

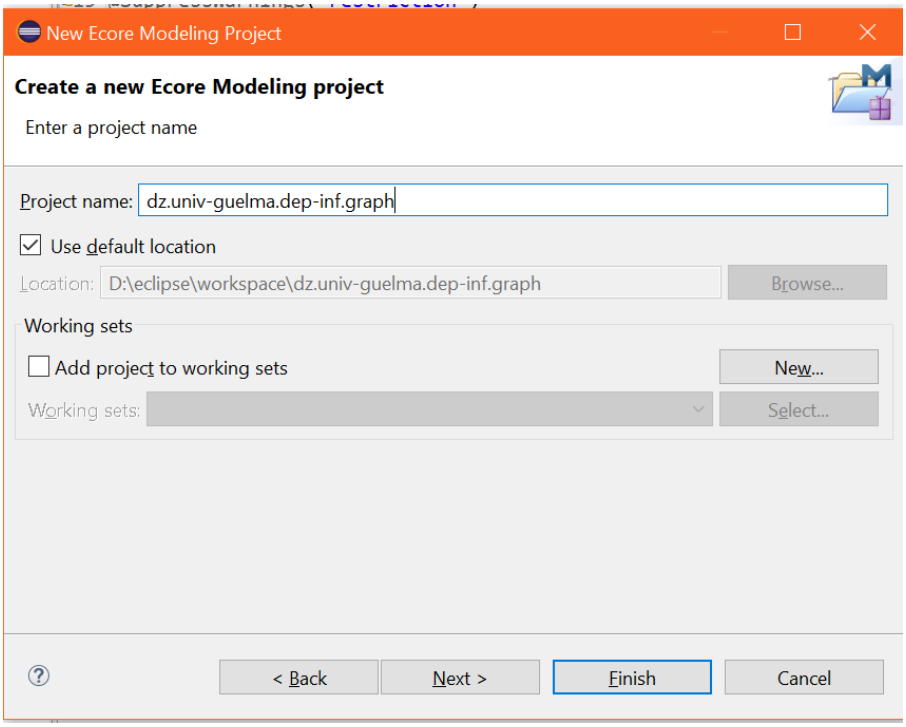


FIGURE 3.2 – Etape 2 Creation d'un projet de modélisation EMF

cliquer sur "Next" pour et remplir les champs comme indiqué dans la figure 3.3.

1. **Main Package Name** donne un nom au conteneur global qui va inclure tous les concept définis dans le modèle.
2. **NS URI** est un identifiant du modèle, toutes référence à ce modèle fait usage de ce identifiant.
3. **NS Prefix** c'est le qualificatif d'espace des nom, utilisé comme extension aux fichier contenant la représentation textuelle des instance du modèle en question.

1. l'utilisation du nom du domaine inversé et fortement conseillé comme nom des projet et fait l'unanimité dans les communautés Java et Eclipse

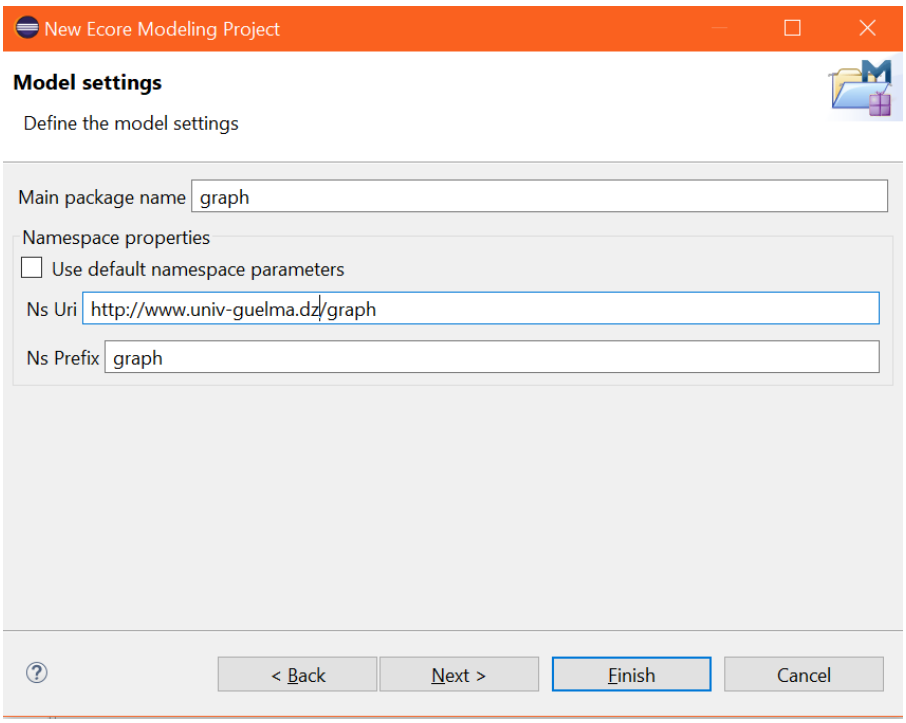


FIGURE 3.3 – étape 3 : choix du nom du modèle ainsi qui ses URI et prefixe

Choisir le point de vue "Design" (figure 3.4) qui convient à notre situation où nous allons définir les concepts du modèle "Graph" et les relation entre eux, puis appuyer sur "Finish".

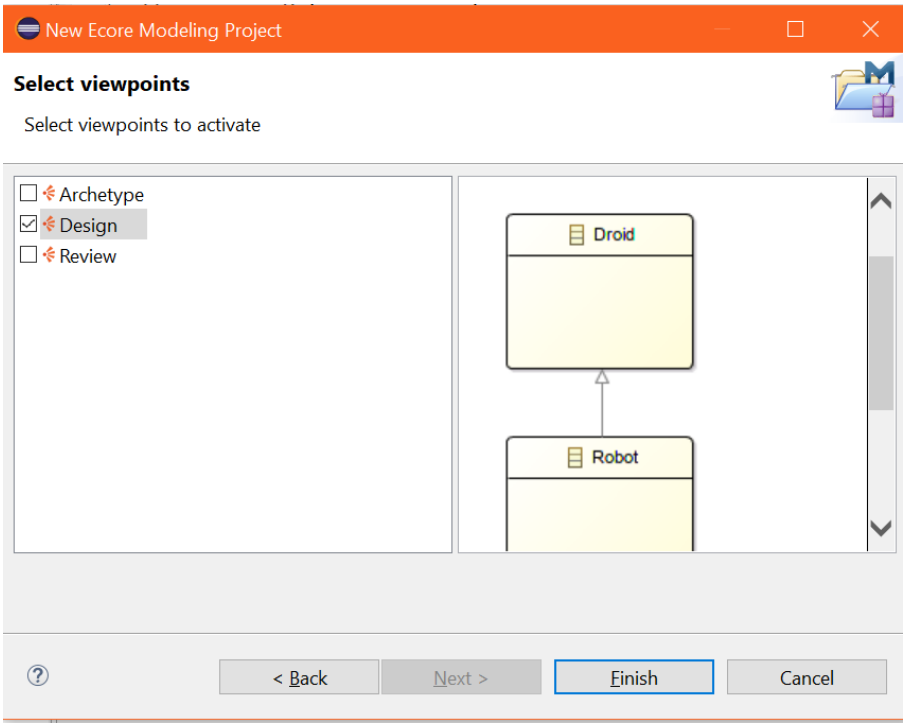


FIGURE 3.4 – étape 4 : choix de point de Vue "Design"

Nous obtenons un environnement nous permettant d'éditer notre modèle de graphe comme montré dans la figure 3.5.

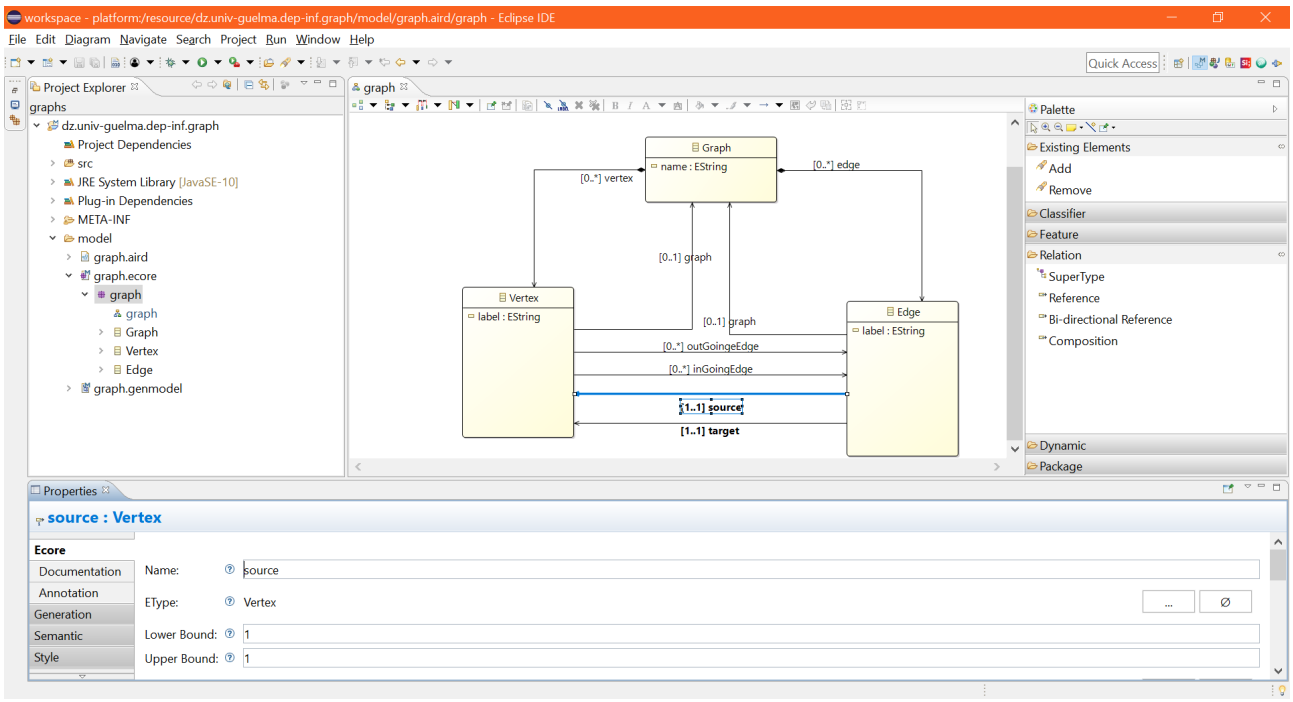


FIGURE 3.5 – Édition du modèle graphe

L'environnement est divisé en trois zones principales :

1. A gauche , un explorateur du projet nous permet d'avoir une vision structuré des éléments constituant notre projet. Nous mentionnant notamment les fichiers suivants :
   — Le fichier "graph.ecore" : contenant la définition du modèle en format XMI
   — Le fichier "graph.genmodel" : contient les directives de génération du code pour notre modèle
   — le fichier "graph.aird" contient la définition de la représentation graphique du modèle "graphe"
2. Au centre l'éditeur principal où le diagramme décrivant le modèle est défini en utilisant la palette associé qui offre plusieurs instrument comme les classe, les énumérations , les relations d'héritage de référence et de composition et bien d'autre chose.
3. en bas, il y a une vue qui permet d'éditer les propriétés des éléments sélectionné dans l'éditeur comme montré dans la figure 3.5 qui montre les propriétés de la relation "Source " entre un "Edge" et un "Vertex".

3.4 Génération du code Java

dans cette étape, nous générerons les entités à partir du modèle "graph" que nous avons créé dans la section précédente . Notez que si nous avons besoin de modifier notre modèle, if faut régénérer les entités à nouveau. EMF peut faire des changements simples comme l'ajout d'éléments de modèle ou des attributs à un élément [?]. pour des modifications complexes, comme le déplacement d'un attribut vers une autre classe, il faut migrer les instances existantes du modèle. Cela est pris en charge par la plateforme EDAPT. (Voir https ://www.eclipse.org/edapt/)

Pour générer des entités, nous devons utiliser un modèle de générateur (voir figure 3.6). Cela nous permet de configurer des propriétés pour la génération de code qui ne font pas partie du modèle lui-même. Par exemple, le code source est également généré pour le plugin et le sous-dossier. Le modèle de générateur se trouve dans le fichier "graph.genmodel". Donc , il suffit d'ouvrir ce fichier et cliquer avec le bouton droit de la souris sur la racine de l'arbre visualisé dans l document puis de générer les codes suivants :

1. Le code Java du modèle
2. le code de adaptateur permettant de visualiser les éléments du modèle dans des vue "Éclipse"
3. le code de l'éditeur des modèles instances de graphe

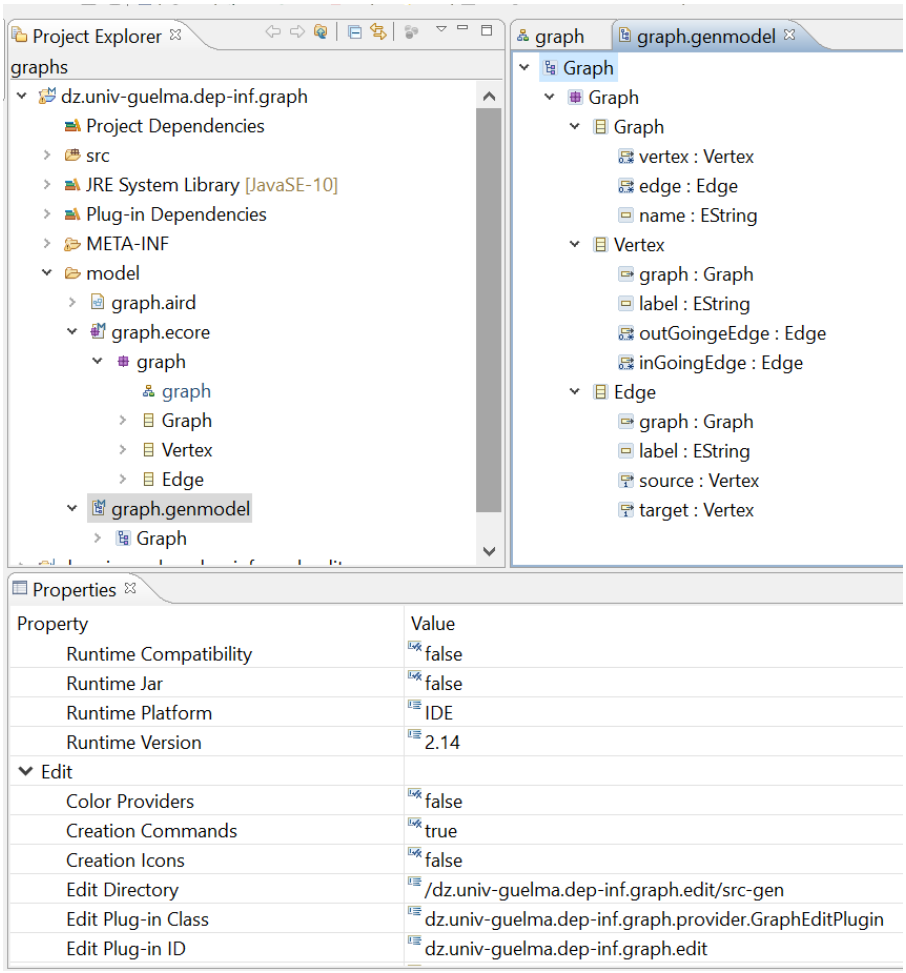


FIGURE 3.6 – Un modèle de générateur pour un modèle EMF

3.5 Visualisation des modèles avec Siruis

la dimension clé dans tout outil de modélisation, c'est la capacité de cet outils à présenter ses modèles d'une manière facile , lisible et agréable. Un développeur d'un outil de modélisation à deux possibilité pour visualiser les modèles, soit :

1. Il implémente intégralement l'interface de visualisation.

2. il fait recours à des outils de génération automatique des interfaces de visualisation.

Dans notre cas, nous avons utilisé la deuxième technique en employant l'outil *Siruis* [**?**]. Sirius est un projet *Eclipse* qui permet de créer, visualiser et éditer vos modèles. Sirius a été créé par *Obeo et Thales* pour fournir un atelier générique d'ingénierie d'architecture basée sur des modèles. Dans le reste de ce chapitre, nous essayerons d'illustrer cet plateforme à travers la présentation des étapes conduisant à un éditeur de graphes décri selon le méta modèle présenté dans les sections précédentes.

3.6 Caractéristiques de Siruis

La puissance de Siruis réside dans les points suivants :

— Il est basé sur le standard EMF.
— il sépare entre la sémantique des modèles et de leurs représentation.
— il permet de définir plusieurs vue pour le même modèle sémantique
— il est facile à utiliser et à promouvoir
— il est extensible et intégrable dans l'environnement *Éclipse* sous forme des plugin

3.7 Concept de Siruis

D'une manière plus précise. Siruis offre deux choses :

1. Une syntaxe graphique [2] pour le méta modèle EMF pour lequel on veut développer l'outil visuel.
2. Un environnement complet (Éditeur visuel puissant, des, persistance en XML , possibilité de retour en arrière en cas d'erreur , intégration dans l'environnement Éclipse en ajoutant des menu principaux et contextuels)

Cela se fait en créant et en configurant *un modèle de spécification de point de vue* qui décrit la structure, l'apparence et le comportement de l'outil de visuel.

3.8 Étapes de Réalisation d'Un Éditeur Visuel avec Siruis

3.8.1 Initialisation de l'éditeur

Pour réaliser un éditeur avec Siruis, il faut créer un nouveau projet de type *Viewpoint Specification* 3.7 .

2. C'est l'équivalent d'une grammaire ou syntaxe textuelle dans le cas d'un langage de programmation. La syntaxe graphique consiste à associer des représentations visuelles à des concepts sémantiques

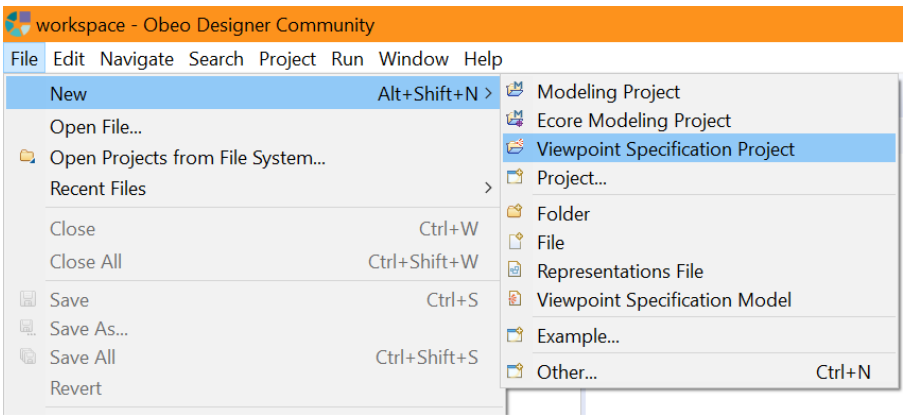


FIGURE 3.7 – Nouveau Projet Siruis

Sur la deuxième page de l'assistant, nous donnons le nom au fichier « *graph.odesign*. Ce dernier contient la spécification de notre éditeur et c'est l'unique fichier que nous allons modifier 3.8.

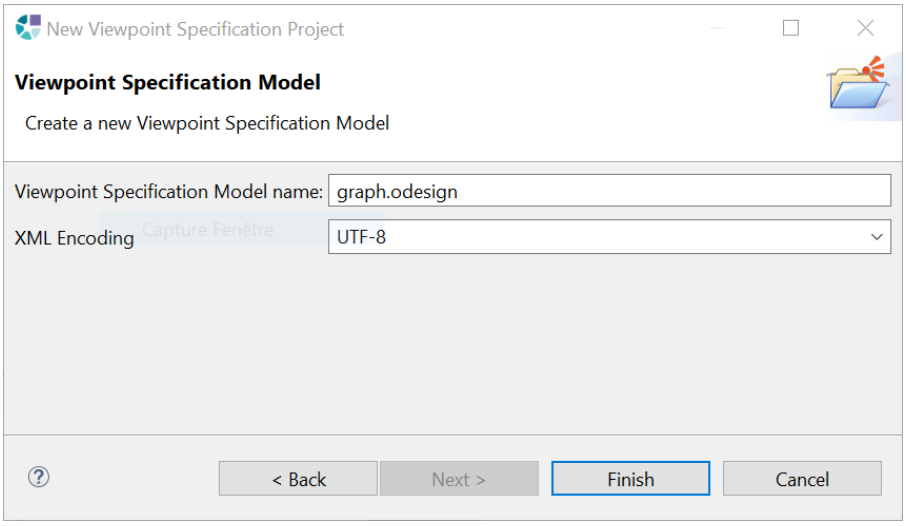


FIGURE 3.8 – création d'un modèle de spécification d'un ViewPoint

En Validant, nous allons obtenir un *Viewpoint* 3.9. Comme son nom l'indique, un viewpoint sert à définir une manière de représenter un modèle. Cela permet de proposer différents points de vue, selon par exemple le niveau de détail souhaité ou l'angle de vision qui veut l'utilisateur.

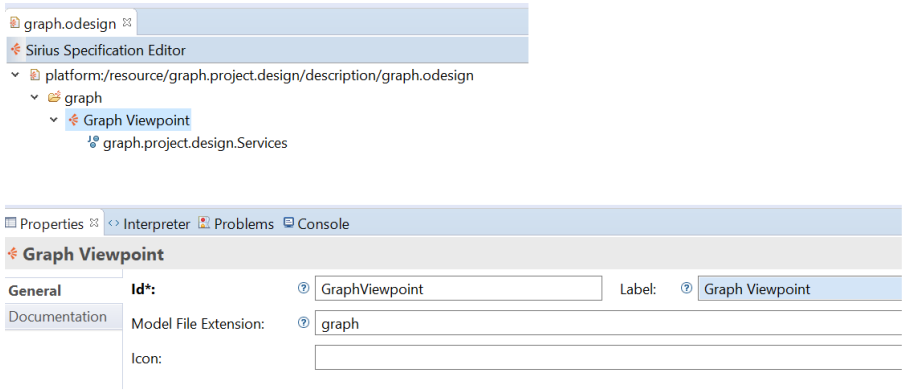


FIGURE 3.9 – Interface de configration d'un ViewPoint

Dans la vue « Properties », trois champs sont à remplir :

1. un champ « ID » : présent sur tous les éléments qui seront définis dans le fichier « odesign » et est obligatoire.
2. un champ « Label » : est là pour donner un nom plus compréhensible aux éléments
3. un champ « Model File Extension » : contient l'extension des fichiers de modèle sémantique que nous voulons associer à notre viewpoint.

Nous devons associer ensuite une représentation à notre viewpoint afin d'indiquer à Sirius quel type d'éditeur nous voulons créer. Dans notre cas, nous allons créer un diagramme comme montrée dans la Figure 3.10

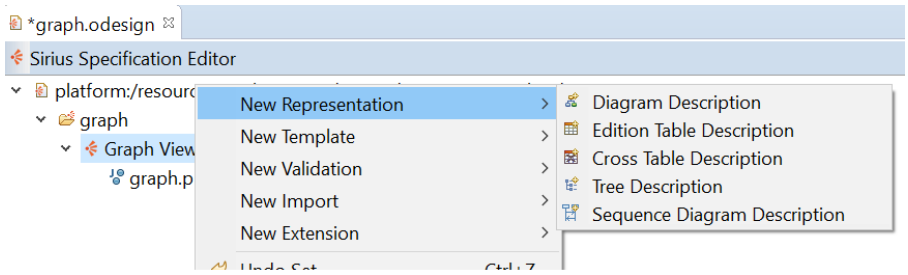


FIGURE 3.10 – Nouvelle Représentation pour le *ViewPoint* qui sera de Type *diagram*. Il est possible aussi de faire des représentations de type arbre ou tableau

Les propriétés d'une description sont montrées dans la figure 3.11.

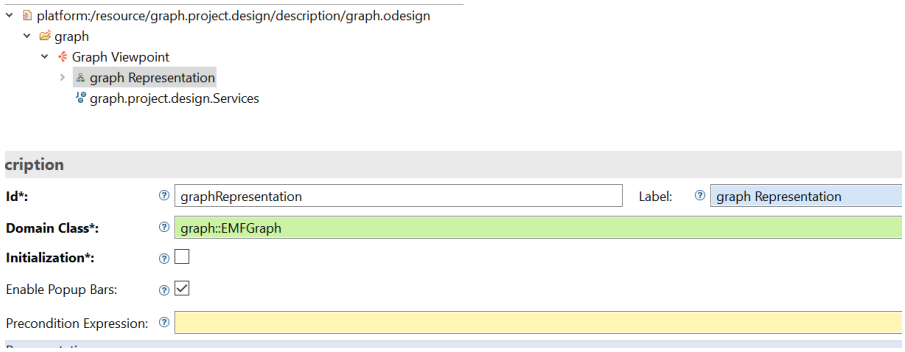


FIGURE 3.11 – Les propriétés d'un représentation, nous mentionnons notamment la Classe qui représente le domaine modélisé qui est un Graphe dans notre cas

3.8.2 Spécification des Nœuds

Une fois nous avons entre nos mains le canevas qui représente tout le graphe et qui est réalisée par l'élément *graph Representation*. Nous commençons à définir (Figure 3.12) les nœuds de notre modèle et qui sont tout simplement les sommets du graphe.

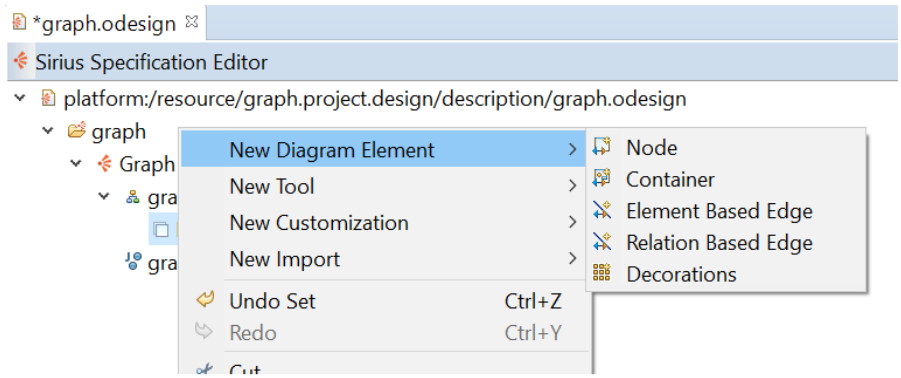


FIGURE 3.12 – Création d'un nouvelle mapping pour un Noeud

La configuration complète d'un nœud nécessite :

1. Création d'un mapping avec une classe du modèle sémantique.
2. configuration de l'apparence du nœud (Forme, couleur, label affiché, .. etc.)
3. définition des règles empêchant éventuellement la visualisation des nœuds qui ne répondent pas à des pré-conditions

3.8.3 Spécification des Relations

En plus des nœuds qui représentent les objets de notre modèle, Sirius permet de définir la représentation des relations entre les éléments. Pour cela, deux possibilités s'offrent à nous :

1. soit un lien basé sur une relation entre deux objets (association, Composition, référence , etc.)
2. soit une relation basée sur un objet explicite

Dans notre cas, nous utilisons la première possibilité car les relations représentent les arc/arrêtes du graphe qui sont modélisé par une classe dans le méta modèle EMF des graphes (Figure 3.5).

FIGURE 3.13 – Les propriétés d'une relation

3.9 Conclusion

EMF et Siruis sont deux plateformes de grandes importance dans le monde de modélisation graphique en génie logiciel. Après l'étude approfondie de ces deux plateformes que nous avons effectuées durant ce chapitre. Nous sommes arrivé au point où nous devons faire la liaison entre la théorie des graphes et la modélisation avec ces plateformes. ceci est l'objet du prochain chapitre. La relation est créé avec un style défini ses propriétés permet de définir les critères d'affichage. Le champ « Source Mapping » donne le type d'objet source, le champ « Target Mapping » donne le type d'objet cible de la relation et le champ « Target finder expression » donne l'expression utilisée pour trouver l'objet cible depuis l'objet source.

Chapitre 4

Conception et Implémentation

Sommaire

4.1 Introduction

Ce chapitre est dédié à la conception et la reconnaissance des motifs de graphes en EMF. Dans ce qui suit, nous détaillerons les différentes étapes de la conception et de la réalisation de notre projet, ainsi que les différents résultats obtenus. Le but de notre projet de fin d'études est de réaliser une application qui permet de visualiser des graphes exemples, nous nous sommes intéressés aux méthodes basées sur la recherche d'isomorphisme de deux (sous-)graphes, en appliquant l'algorithme d'Ullmann qui est considérer comme un algorithme plus efficace dans le domaine d'appariement de graphe par arbre de recherche. Nous sommes expérimentés à :

1. Chercher l'isomorphisme de deux (sous-)graphes
2. Agrandir la taille des graphes.
3. Ensuite Comparer la complexité d'algorithme appliqué avec et sans raffinement.

4.2 Vue Globale de notre Travail

Afin d'atteindre nos objectifs consistant à trouver des pattern dans des graphes EMF. nous devons mettre en œuvre une plateforme qui nous permet

1. de définir des graphes EMF et de les manipuler
2. de visualiser les graphes
3. d'établir un lien entre la représentation visuel du graphe et le modèle EMF correspondant
4. d'exécuter les différents algorithmes de graphes en particulier l'algorithme de détection d'isomorphismes.
5. de répercuter les résultat obtenus sur la représentation visuel.

Ceci nous conduit à adopter une démarche comme celle décrit dans la figure En effet, La

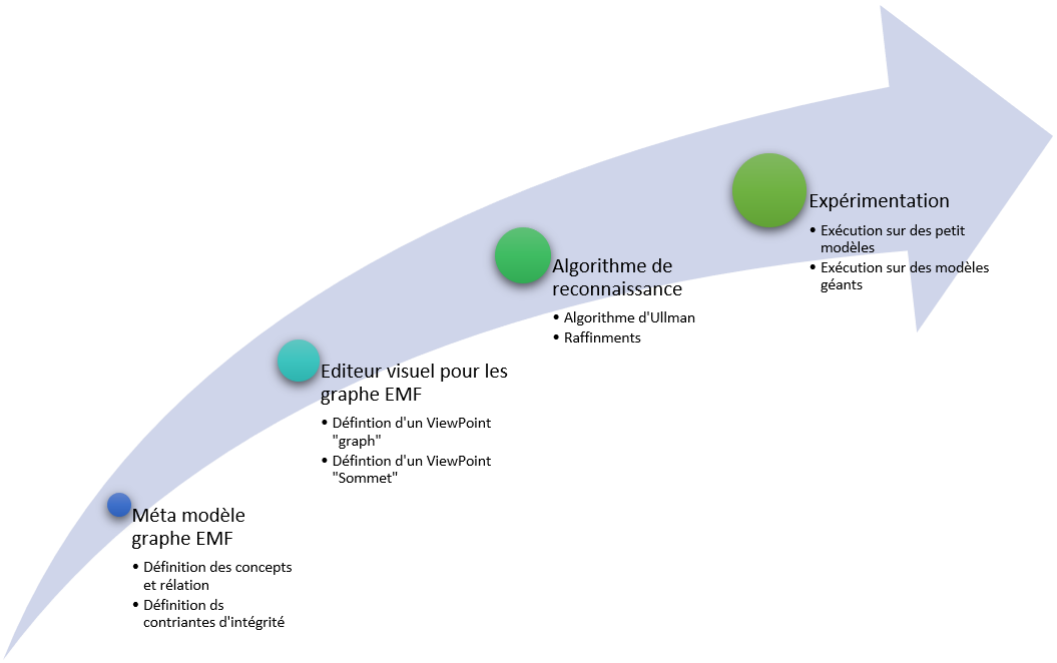


FIGURE 4.1 – Vue globale de notre travail

première étape primordiale est de définir le concept d'un graphe au sens EMF. ceci passe obligatoirement par la définition d'un modèle Ecore pour les graphes. cette étape est la base de tout les autres qui la succèdent.

Sur la base du méta-modèle graphe. Nous développement un éditeur visuel moyennant des facilités offertes par la plateforme "Sirius" pour la construction des "View Point" (Formalisme graphique) pour des DSL (Domain secific Language) qui est dans notre cas les graphes.

Une fois nous avons entre les nos mains les graphes et leurs visualisation, il arrive le temps d'implémenter notre algorithme de reconnaissance de pattern qui n'est plus qu'une implémentation de l'algorithme d'Ullmann très réputé en littérature. Nous nominalisons par l'analyse des résultat obtenu t la discussion de la faisabilité de notre implémentation.

4.3 Le Méta Modèle "graphe"

En suivant les étapes expliquées dans le chapitre 3 (voir **??**). Nous Présentons dans la figure 4.2 Notre méta modèle de graph. Dans le tableau nous résumons la signification des différent concepts, leurs relation ainsi que leurs contraintes.

4.4 Éditeur Visuel pour les graphes

Nous avons utilisé l'outil "Sirius" pour la mise en œuvre de notre éditeur des graphes. En se basant sur la description de sirius présenté dans le chapitre 3. nous avons défini deux point de vue :

1. Un point de Vue graphe : permettant de visualiser et éditer un graphe entier
2. un point de vue Sommet : permet de visualiser et éditer un sommet particulier

La figure donne un aperçu sur la définition des point de vue La figure nous montre une capture d'écran de notre éditeur visuel des graphes

1. La zone 1 résume un graphe d'une manière structurée
2. La zone 2 permet de visualiser et manipuler un graphe, plusieurs graphes peuvent être affiche en juxtaposé.
3. La zone 3 offre un palette d'outils permettant d'insérer des sommets ,des arcs/arrêtes ou des "Entries"
4. La zone 4 permet d'éditer les propriétés d'un graphe, sommet , arc/arrête, entré
5. La zone 5 donne un aperçu, c'est très pratique pour des graphes larges.

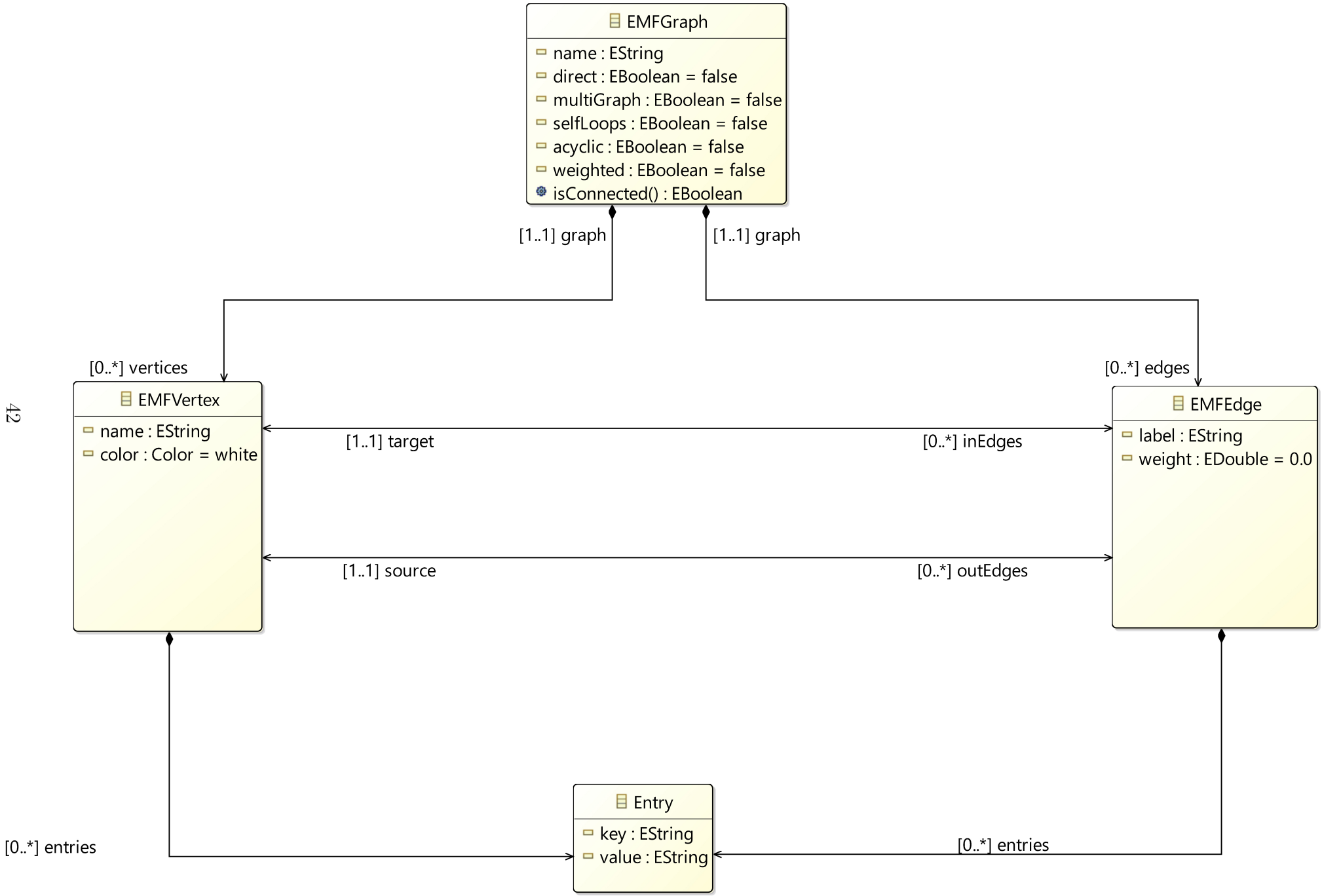
4.5 Algorithme utilitaires

Afin de pouvoir implémenter convenablement notre algorithme de reconnaissance de pattern. Ils nous faut utiliser un ensemble d'algorithme sur les graphes dans les sous section suivante nous présentons quelques algorithme que nous avons implémenter. Il est judicieux de notre à que nous avons fait recours à un API de graphe qui s'appelle "JGraphT" pour représenter la structure de donnée graphe. En effet, nous commençons par un Graphe EMF édité par l'utilisateur pour le transformer en un graphe "JGraphT" puis nous extrayons les matrices d'adjacence pour qu'on puisse appliquer l'algorithme d'Ullmann qi utilise la représentation matricielle d'un graphe. suite à l'exécution de l'algorithme d'Ullmann nous interprétons les résultats sous forme d'un graph EMF pour les afficher à l'utilisateur.

4.5.1 Génération des graphes aléatoires

Listing 4.1 – une méthode java pour la génération aléatoire d'un graphe

```
1 private static int randomGraph(int n, int m, long seed, boolean simple, boolean directed, boolean acyclic,
```


EMFGraph
name : EString
direct : EBoolean = false
multiGraph : EBoolean = false
selfLoops : EBoolean = false
acyclic : EBoolean = false
weighted : EBoolean = false
isConnected() : EBoolean
[1..1] graph
[1..1] graph
[0..*] vertices
[0..*] edges
EMFVertex
name : EString
color : Color = white
EMFEdge
label : EString
weight : EDouble = 0.0
[1..1] target
[0..*] inEdges
[1..1] source
[0..*] outEdges
Entry
key : EString
value : EString
[0..*] entries
[0..*] entries

Concepts	sémantique	contraintes
EMFGraph	Graphe comme conteneur des Sommets et des Arrêtes/arcs	1. Un graphe ne doit contenir deux sommets portant le même nom 2. Un graphe ne doit pas contenir deux Arrêtes/arcs avec le même nom. 3. Si le graphe est simple alors il doit pas contenir aucun boucle ni arrêtes/arcs multiples
EMFVertex	Représente les sommets du graphes	1. chaque sommet doit appartenir à au moins un graphe 2. chaque sommet doit porter un nom distinct 3. chaque sommet possède une couleur 4. chaque sommet peut définir un ensemble de pairs(clé, valeur)
EMFEdge	une arrête dans le cas d'un graphe non orienté et un arc dans le cas d'un graphe orienté	1. chaque arc/arrête doit appartenir à au moins un graphe 2. chaque arc/arrête doit porter un nom distinct 3. chaque arc/arrête peut définir un ensemble de pairs(clé, valeur) 4. chaque arc/arrête peut avoir un poids 5. dans le cas d'un graphe simple, la source et la destination d'un arc/arrête doivent être différentes
Entry	un ensemble de pairs (clé, valeur) permettant de donner une sémantique précise aux sommets et arcs/arrêtes	1. une "Entry" appartient exclusivement à un Sommet ou arc/arrête unique 2. Les clé d'une entry sont disjoint un à un

TABLE 4.1 – Sémantique du méta modèle Graphe

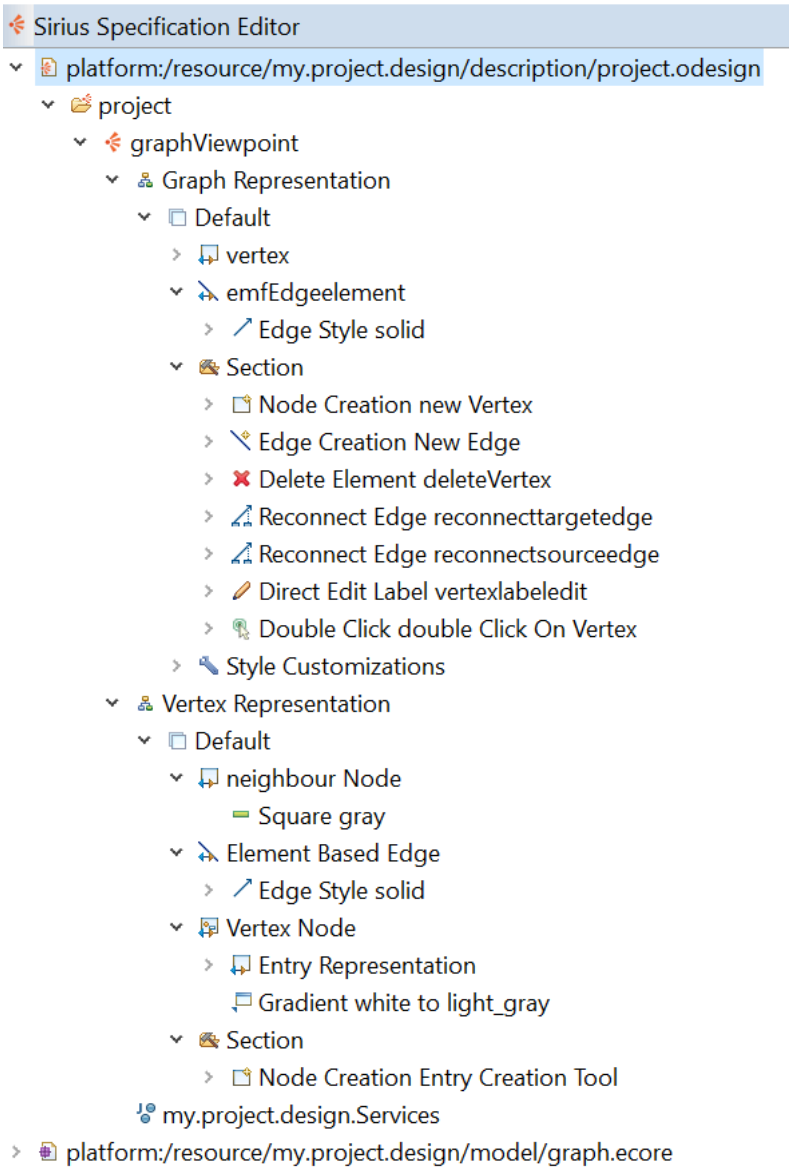


FIGURE 4.3 – Définition des points de vue graphe et vertex

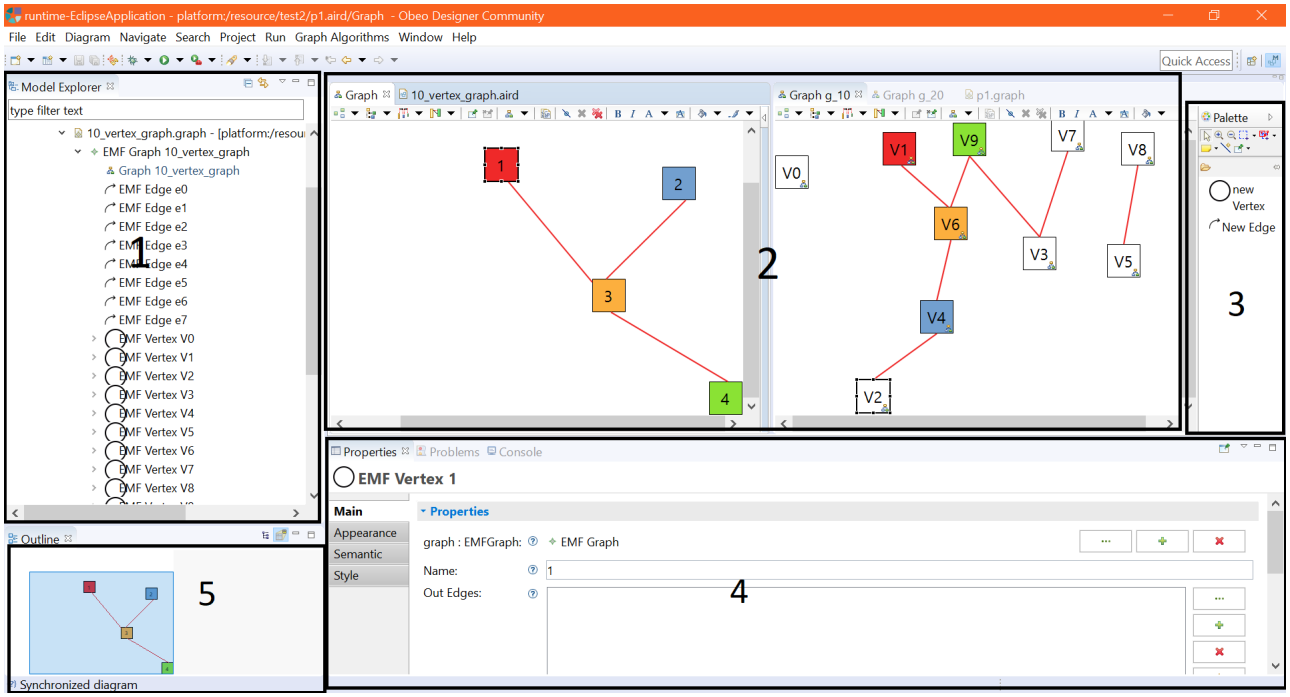


FIGURE 4.4 – Les Zones de notre éditeur visuel

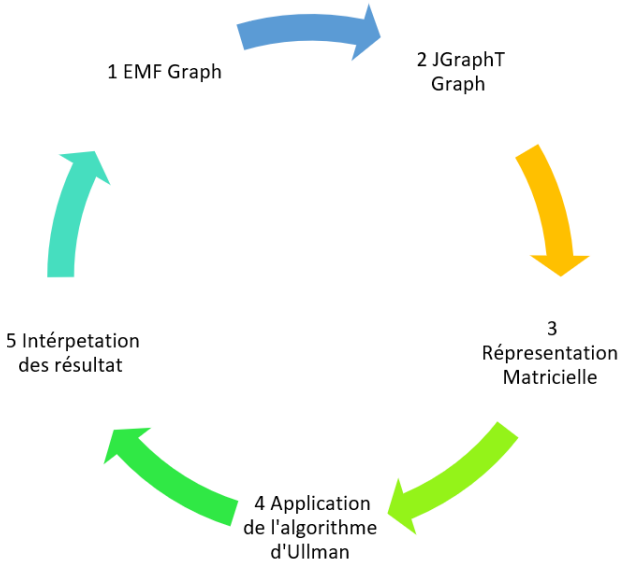


FIGURE 4.5 – Cycle de transformation faisant appelle à des algorithmes utilitaires

```
2                       boolean weighted, int minweight, int maxweight, int nodei[], int nodej[], int weight[]) {
3               int maxedges, nodea, nodeb, numedges, temp;
4               int dagpermute[] = new int[n + 1];
5               boolean adj[][] = new boolean[n + 1][n + 1];
6               Random ran = new Random(seed);
7               // initialize the adjacency matrix
8               for (nodea = 1; nodea <= n; nodea++)
9                       for (nodeb = 1; nodeb <= n; nodeb++)
10                              adj[nodea][nodeb] = false;
11              numedges = 0;
12              // check for valid input data
13              if (simple) {
14                      maxedges = n * (n - 1);
15                      if (!directed)
16                              maxedges /= 2;
17                      if (m > maxedges)
18                              return 1;
19              }
20              if (acyclic) {
21                      maxedges = (n * (n - 1)) / 2;
22                      if (m > maxedges)
23                              return 1;
24                      randomPermutation(n, ran, dagpermute);
25              }
26              while (numedges < m) {
27                      nodea = ran.nextInt(n) + 1;
28                      nodeb = ran.nextInt(n) + 1;
29                      if (simple || acyclic)
30                              if (nodea == nodeb)
31                                      continue;
32                      if ((simple && (!directed)) || acyclic)
33                              if (nodea > nodeb) {
34                                      temp = nodea;
35                                      nodea = nodeb;
36                                      nodeb = temp;
37                              }
38                      if (acyclic) {
39                              nodea = dagpermute[nodea];
40                              nodeb = dagpermute[nodeb];
41                      }
42                      if ((!simple) || (simple && (!adj[nodea][nodeb]))) {
43                              numedges++;
44                              nodei[numedges] = nodea;
45                              nodej[numedges] = nodeb;
46                              adj[nodea][nodeb] = true;
47                              if (weighted)
48                                      weight[numedges] = (int) (minweight + ran.nextDouble() *
49                                      (maxweight + 1 - minweight));
50                      }
51              }
52              return 0;
53      }
```

4.5.2 Analyse de graphe

degrés d'un Sommet

Listing 4.2 – La méthode de calcul du degré d'un sommet d'un graphe

```
1   public static int degreeOut(EMFGraph graph, EMFVertex vertex) {
2               EList<EMFVertex> vertices = graph.getVertices();
3               int size = graph.getVertices().size();
4               if (size == 0 || !vertices.contains(vertex))
5                       return -1;
6               return vertex.getOutEdges().size();
7       }
8
9       public static int degreeIn(EMFGraph graph, EMFVertex vertex) {
10              EList<EMFVertex> vertices = graph.getVertices();
11              int size = graph.getVertices().size();
12              if (size == 0 || !vertices.contains(vertex))
13                      return -1;
14              return vertex.getInEdges().size();
15      }
16
17      public static int degree(EMFGraph graph, EMFVertex vertex) {
18              EList<EMFVertex> vertices = graph.getVertices();
```

```
19                 List<EMFVertex> neihbours = neighbour(graph, vertex);
20                 int size = graph.getVertices().size();
21                 if (size == 0 || !vertices.contains(vertex))
22                         return -1;
23
24                 return degreeOut(graph, vertex) + degreeIn(graph, vertex);
25         }
```

Matrice d'adjacence d'un graphe

Listing 4.3 – La méthode de calcul de la matrice d'adjascance d'un EMF Graph

```
1  public static List<EMFEdge> incidentEdges(EMFGraph graph, EMFVertex vertex) {
2                  EList<EMFVertex> vertices = graph.getVertices();
3                  int size = graph.getVertices().size();
4                  if (size == 0 || !vertices.contains(vertex))
5                          return null;
6                  List<EMFEdge> incidentEdges = new ArrayList<EMFEdge>();
7                  incidentEdges.addAll(vertex.getOutEdges());
8                  for (EMFEdge e : vertex.getInEdges()) {
9                          if (!incidentEdges.contains(e))
10                                 incidentEdges.add(e);
11                 }
12                 return incidentEdges;
13         }
```

Conversion entre EMFGraph et JGraphT

EMF graph vers JGraphT

Listing 4.4 – La méthode de conversion d'un EMF graph vers JGraphT graph

```
1  public static Graph<EMFVertex, EMFEdgeWrapper> EMFtoTgraphT(EMFGraph graph) {
2
3                  Graph<EMFVertex, EMFEdgeWrapper> g = graph.isDirect()
4                                  ? GraphTypeBuilder.<EMFVertex, DefaultEdge>directed().
5                                    allowingMultipleEdges(graph.isMultiGraph())
6                                  .allowingSelfLoops(graph.isSelfLoops()).edgeClass(EMFEdgeWrapper.class)
7                                  .weighted(graph.isWeighted()).buildGraph()
8                                  : GraphTypeBuilder.<EMFVertex, DefaultEdge>undirected().
9                                  allowingMultipleEdges(graph.isMultiGraph())
10                                 .allowingSelfLoops(graph.isSelfLoops()).edgeClass(EMFEdgeWrapper.class)
11                                 .weighted(graph.isWeighted()).buildGraph();
12
13                 EList<EMFVertex> vertices = graph.getVertices();
14                 Iterator<EMFVertex> verticesIt = vertices.iterator();
15                 while (verticesIt.hasNext()) {
16                         EMFVertex vertex = verticesIt.next();
17                         g.addVertex(vertex);
18                 }
19
20                 EList<EMFEdge> edges = graph.getEdges();
21                 Iterator<EMFEdge> edgeIt = edges.iterator();
22                 while (edgeIt.hasNext()) {
23                         EMFEdge edge = edgeIt.next();
24                         g.addEdge(edge.getSource(), edge.getTarget(), new EMFEdgeWrapper(edge));
25
26                 }
27
28                 return g;
29         }
```

JGraphT vers EMF graph

Listing 4.5 – La méthode de conversion d'un JGraphT graph vers EMF graph

```
1  public static EMFGraph tgraphToEMF(Graph<String, DefaultEdge> graph) {
2                  EMFGraph g = GraphFactory.eINSTANCE.createEMFGraph();
```

```
3
4                 Set<String> vertices = graph.vertexSet();
5                 for (Iterator<String> iterator = vertices.iterator(); iterator.hasNext();) {
6                         String name = iterator.next();
7                         EMFVertex emfVertex = GraphFactory.eINSTANCE.createEMFVertex();
8                         emfVertex.setName(name);
9                         g.getVertices().add(emfVertex);
10                }
11                Set<DefaultEdge> edges = graph.edgeSet();
12                int count = 0;
13                for (Iterator<DefaultEdge> iterator = edges.iterator(); iterator.hasNext();) {
14                        DefaultEdge edge = iterator.next();
15                        String sourceVertex = graph.getEdgeSource(edge);
16                        String targetVertex = graph.getEdgeTarget(edge);
17                        double weight = graph.getEdgeWeight(edge);
18
19                        EMFEdge emfEdge = GraphFactory.eINSTANCE.createEMFEdge();
20                        emfEdge.setWeight(weight);
21                        emfEdge.setLabel("e" + count++);
22                        EMFVertex source = getEmfVertexbyName(g, sourceVertex);
23                        emfEdge.setSource(source);
24                        EMFVertex target = getEmfVertexbyName(g, targetVertex);
25                        emfEdge.setTarget(target);
26                        g.getEdges().add(emfEdge);
27
28                }
29                return g;
30        }
```

4.6 Exemple Exécution de l'Algorithme d'Ullman

Soit deux graphe $G_1(V_A, E_A)$, $G_2(V_B, E_B)$ avec leurs matrice d'adjacence A et B comme présenté dans la figure 4.6.

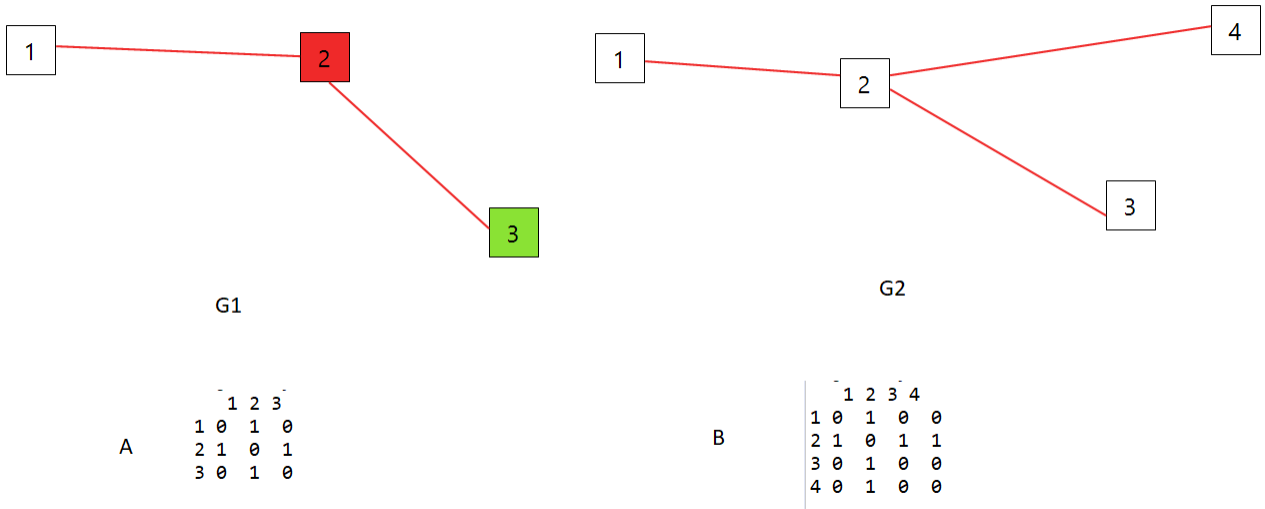


FIGURE 4.6 – Exemple de Déroulement de l'Algorithme d'Ullmann

Le listing de la figure 4.7 présente les étapes de déroulement

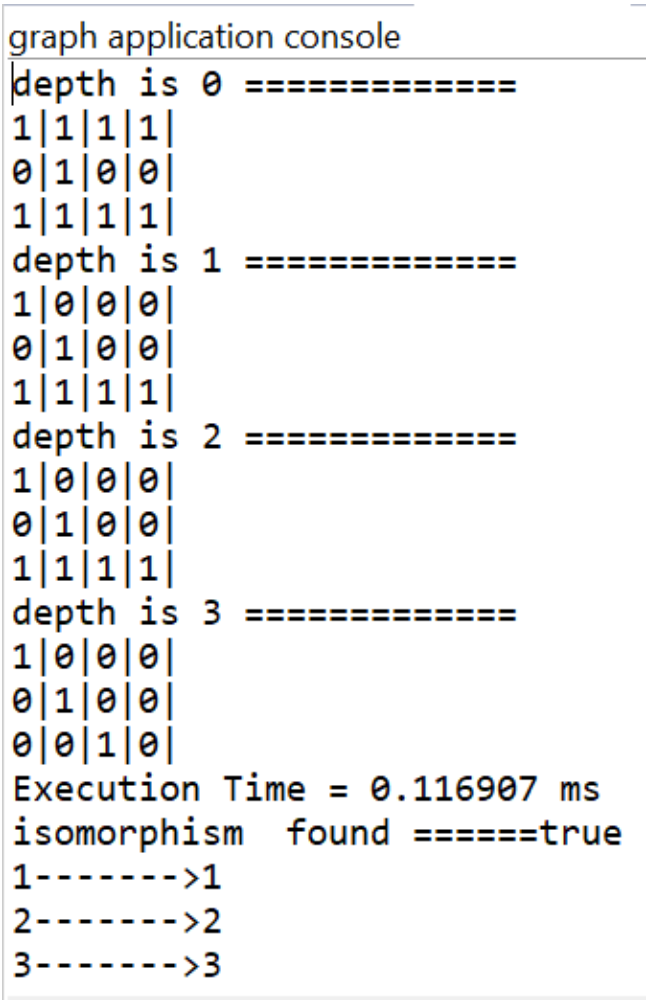


FIGURE 4.7 – Étapes de déroulement de l'algorithme d'Ullmann sur les graphes G1 et G2

4.7 Expérimentation et discussion

Nous avons effectués nos expérimentation sur une machine de type Intel i5 avec une mémoire vive de 6 Go. Notre objectif est d'analyser le comportement de notre implémentation dans des différentes situations :

1. des pattern petit, moyen , grands
2. des target petit , moyen ,grands

Nous résumons les résultats de notre exécution dans le tableau 4.2

Nombre sommets cible	nombre arrêtes cible	Nombre sommets cible	Nombre sommets cible	durée d'exécution sans raffinement	durée d'exécution avec raffinement
5	8	4	4	5.91 ms	0.20 ms
10	15	4	4	0.91 ms	0.70 ms
15	17	4	4	4.71 ms	1.35 ms
20	26	4	4	1.45 ms	2.83 ms
30	36	4	4	10.04 ms	3.40 ms
40	51	4	4	9.89 ms	2.81 ms
50	59	4	4	13.24 ms	5.93 ms
70	81	4	4	40.92 ms	11.60 ms
80	99	4	4	50.30 ms	19.18 ms
100	112	4	4	67.68 ms	35.90 ms

TABLE 4.2 – Temps d'exécution de l'algorithme d'Ullmann avec et sans raffinement
Les résultat sont mis en évidence dans les courbes de la figure 4.8

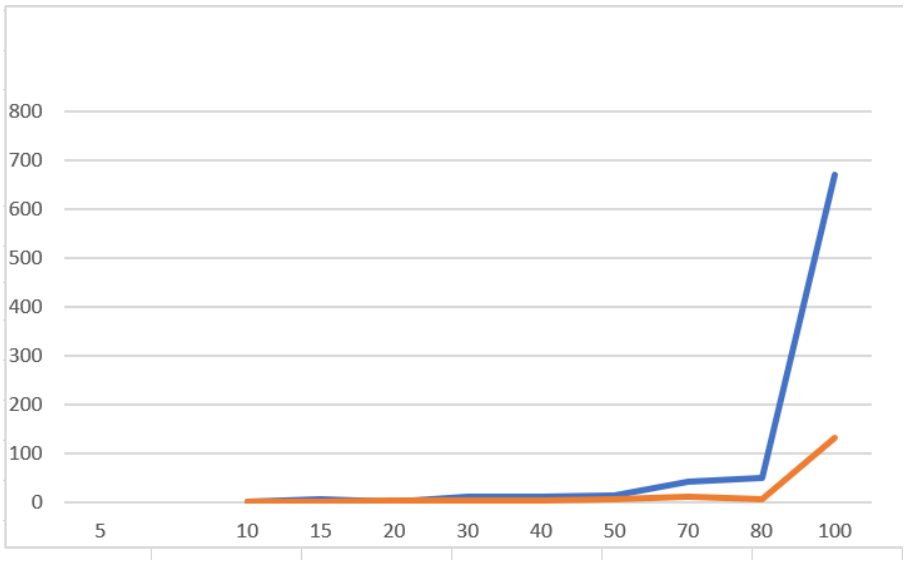


FIGURE 4.8 – Courbe relatif à l'exécution de l'algorithme d'Ullmann : le courbe en bleu représente le temps d'exécution sans raffinement en fonction du nombre des sommets dans le graphes cible. la courbe en rouge représente le temps d'exécution avec raffinement

4.8 Conclusion

Le but de ce chapitre est d'implémenter l'algorithme d'Ullmann afin de détecter les pattern dans des graphes réalisé avec la plateformes EMF. Nous avons défini dans un premier temps le méta-modèle *Graph* puis nous avons élaboré un éditeur visuel avec Siruis permettant de manipuler graphiquement des graphes. Enfin, nous avons ajouter des menus à l'environnement Éclipse pour pouvoir exécuter les différentes commandes tel que les statistiques sur les graphes , la génération des graphes aléatoires et finalement l'exécution de algorithme d'Ullman.

Conclusion générale

Les graphes ont une forme essentielle d'organisation des données et font leur apparition dans les applications les plus surprenantes de la chimie, de la neurologie, des sciences sociales, de la linguistique, et de bien d'autres. L'une des tâches les plus fréquentes lors de l'analyse de graphes est la recherche de différents modèles. Le problème de la mise en correspondance de deux graphes (reconnaissance des patterns dans les graphes) appelé problème d'isomorphisme de sous-graphe. C'est dans ce cadre que nous avons élaborés notre travaille dans lequel nous avons implémentés une méthode qui cherche l'isomorphisme de sous-graphe présenter par un algorithme pour trouver une solution au problème avec un temps de calcul acceptable.

Sur cette base il existe des algorithmes efficaces pour faire correspondre de grands graphes, à la fois exactement et inexactement, avec différents types d'isomorphisme sont maintenant disponibles, nous avons appliqué l'algorithme « d'Ullmann » et testé les résultats obtenus avec et sans raffinement ce qui permet de résoudre le problème du graphe. À partir des approches existantes, il peut être conclu que l'algorithme Ullmann est utilisé pour les graphes étiquetés. Ullmann a utilisé une procédure récursive de backtracking et de raffinement pour réduire l'espace de recherche et l'incorpore dans l'algorithme. Et aussi l'approche de retour en arrière et il correspond exactement au type d'algorithme qui réduit la taille de l'espace de recherche. Notre sujet est très important et présente des champs d'application très vastes, nous avons essayé de réunir les informations pour une étude approfondie en manipulant différentes méthodes pour trouver le matching entre le graphe de données et le graphe modèle. En général, les résultats sont très encourageants et ont permis la réalisation physique d'un système de reconnaissance de motifs dans de graphes EMF. Enfin, le programme implémenté montre que l'algorithme d'Ullmann est efficace pour chercher à trouver tous les isomorphismes de (sous-) graphes possibles avec un temps de calcul, à chaque exemple que nous avons essayé (après chargement des deux structures des deux graphes), nous avons trouvés que le temps de calcule était moins quand le processus de raffinement est appliqué, Les résultats ont prouvé à chaque test l'efficacité de l'algorithme pour tester la complexité temporelle. Cependant, et certainement, des améliorations restent toujours nécessaires à apporter.

Bibliographie

[1] U. Čibej and J. Mihelič, "Improvements to ullmann's algorithm for the subgraph isomorphism problem," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 29, no. 07, p. 1550025, 2015.

[2] J.-C. Fournier, "Graphes et applications," *Hermes science publications-Lavoisier*, 2007.

[3] A. Bretto, A. Faisant, and F. Hennecart, *Éléments de théorie des graphes*. Springer Science & Business Media, 2012.

[4] T. Bärecke, *Isomorphisme inexact de graphes par optimisation évolutionnaire*. PhD thesis, 2009.

[5] S. Sorlin and C. Solnon, "Similarité de graphes : une mesure générique et un algorithme tabou réactif," 2005.

[6] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[7] 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*, vol. 28, no. 01, p. 1450001, 2014.

[8] J. E. Hopcroft and J.-K. Wong, "Linear time algorithm for isomorphism of planar graphs (preliminary report)," in *Proceedings of the sixth annual ACM symposium on Theory of computing*, pp. 172–184, ACM, 1974.

[9] J. Kobler, U. Schöning, and J. Torán, *The graph isomorphism problem : its structural complexity*. Springer Science & Business Media, 2012.

[10] T. Cormen, "Introduction à l'algorithmique," 1997.

[11] B. Gallagher, "Matching structure and semantics : A survey on graph-based pattern matching.," in *AAAI Fall Symposium : Capturing and Using Patterns for Evidence Detection*, pp. 45–53, 2006.

[12] J. Cheng, J. X. Yu, and S. Y. Philip, "Graph pattern matching : A join/semijoin approach," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 7, pp. 1006–1021, 2010.

[13] S. Jouili, *Indexation de masses de documents graphiques : approches structurelles*. PhD thesis, 2011.

[14] A. M. Abdulkader, "Parallel algorithms for labelled graph matching," *Colorado School of Mines*, 1998.

[15] F. Budinsky, D. Steinberg, R. Ellersick, T. J. Grose, and E. Merks, *Eclipse modeling framework : a developer's guide*. Addison-Wesley Professional, 2004.

[16] C. Brun and A. Pierantonio, "Model differences in the eclipse modeling framework," *UPGRADE, The European Journal for the Informatics Professional*, vol. 9, no. 2, pp. 29–34, 2008.

[17] V. Viyović, M. Maksimović, and B. Perisić, "Sirius : A rapid development of dsm graphical editor," in *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, pp. 233–238, IEEE, 2014.