Ecole Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 18
www.polytech.univ-tours.fr

Département Informatique
5ème année
2012-2013

Rapport de Projet de Fin d'Etudes

GraphLib : une librairie C# pour l'exploitation de graphes en reconnaissance des formes

Encadrants :
Jean-Yves Ramel
jean-yves.ramel@univ-tours.fr
Romain Raveaux
romain.raveaux@univ-tours.fr

Etudiant :
Sébastien Schaal
sebastien.schaal@etu.univ-tours.fr

Année 2012-2013
DI5 – Promotion 2013

Table des matières

Table des figures

Remerciements

Je voudrais tout d’abord commencer par remercier messieurs Jean-Yves Ramel et Romain Raveaux, mes encadrants, pour les conseils et l’aide précieuse qu’ils ont su me procurer tout au long de ce projet.

Introduction

Au cours de ma formation d'informatique à l'école d'ingénieur Polytech'Tours, j'ai été amené à faire un projet de fin d'étude durant ma 5ème année. Ce dernier s'est déroulé du 27 septembre 2012 au 30 avril 2013.

Ce document a pour but de présenter le travail effectué et de dresser un bilan, à propos du sujet intitulé « GraphLib : une librairie C# pour l'exploitation de graphes en reconnaissance des formes ». Ce projet fait suite à un projet commencé par un ancien étudiant Geoffrey Chopin qui a eu l'occasion de commencer la librairie au cours de son stage de fin d'étude. Cette librairie a pour but d'être utilisée plus tard par des développeurs C# souhaitant utiliser des algorithmes de graphes.

Le document est séparé en deux parties principales. La première partie permettra de faire une présentation du projet, des objectifs et de l'existant. La seconde partie du document est consacrée au travail réalisé pendant ce PFE. Cette partie permettra de décrire les différentes modifications apportées à la librairie, mais aussi de décrire le logiciel de test développé à partir de cette librairie, pour finir par les résultats de la phase de test.

Ce projet a été encadré par Mr Jean-Yves Ramel et Mr Romain Raveaux.

Présentation du projet

1 Contexte du projet

Le graphe est un bon moyen de représentation d'objet en informatique. Le principe du graphe est de pouvoir représenter un objet de façon simple, pour résoudre un problème par une méthode de résolution liée aux graphes. Dans notre cas, nous allons nous intéresser plus précisément aux graphes représentant des images. Le but recherché étant de pouvoir effectuer de la reconnaissance de forme à partir de ces graphes.

Le principe est ensuite d'utiliser des algorithmes d'isomorphisme ou de similarité sur ces graphes pour les comparer. Les algorithmes d'isomorphisme sont séparés en deux sous parties : les isomorphismes exacts et les isomorphismes approchés. Les isomorphismes exacts considèrent que deux graphes ne sont pas isomorphes dès qu'une différence est constatée, tandis que les isomorphismes approchés considèrent que deux graphes peuvent être isomorphes en dessous d'une certaine marge d'erreur. Un isomorphisme peut être de graphe (graphe équivalent à un autre graphe) ou de sous-graphe (graphe sous-graphe d'un autre graphe).

La distance d'édition est un algorithme de similarité permettant lui aussi de comparer deux graphes mais ne retourne pas la même information. Il permet de renvoyer une valeur correspondante au niveau de ressemblance entre deux graphes. La distance d'édition peut retourner une valeur exacte ou une valeur approchée selon l'algorithme utilisé.

Ce projet s'inscrit à la suite d'un projet déjà commencé par un ancien élève de l'école et le but est de continuer ce qui a été entrepris par ce dernier.

2 Objectifs

Le projet porte sur la création d'une librairie open source en C# permettant d'utiliser des graphes et des algorithmes de calcul d'isomorphisme et de dissimilarité associés à ces derniers. La première version de la librairie propose déjà une structure de graphe, ainsi que l'implémentation de 2 algorithmes de recherche d'isomorphismes (Corneil & Gotlieb et VF2).

Les objectifs de mon projet sont les suivants :

- Une finalisation de l'algorithme VF2
- Une modification de certaines parties de la modélisation objet des graphes
- L'ajout de l'algorithme de calcul de distance d'édition (une ou plusieurs implémentations)
- Création d'une application permettant de passer d'une image à un graphe
- Création d'une application de test permettant de gérer les différentes possibilités de la librairie
- Rendre la librairie open source

3 Description de l'existant

La librairie existante est séparée en deux parties : une librairie **Graphs** et une librairie **Matching**.

3.1 La librairie Graphs

La librairie Graphs permet de gérer les graphes. Un graphe est composé de Node et d'Edge. Chaque Node et Edge hérite d'un GraphComponent qui permet de les relier à un Label. Le label est le composant clé permettant de définir les différents attributs du graphe. Pour comparer deux attributs, il faudra comparer deux labels. La classe label étant une classe abstraite, il faut créer un label héritant de cette classe pour chaque type de graphe. Cependant une classe GenericLabel permet de passer outre et de créer un label générique.

La librairie permet aussi de gérer les différentes entrées / sorties liées aux graphes et aux attributs GXL. Elle dispose aussi de 4 interfaces permettant de faire le lien avec la distance entre 2 graphes, le matching entre 2 graphes, l'isomorphisme de graphe et de sous-graphe. La librairie existante sera modifiée pour ajouter une nouvelle méthode de comparaison de nœuds, ainsi qu'un nouvel accesseur lié aux nœuds du graphe.

Vous trouverez le diagramme de classe correspondant à la librairie actuelle en Annexes (cf. Figure 18).

3.2 La librairie Matching

La librairie Matching quant à elle permet d'utiliser la librairie Graphs dans les algorithmes de matching entre graphe. Elle est actuellement utilisée avec les algorithmes de Corneil & Gotlieb (associé à un arbre de recherche) ainsi que VF2. Une autre classe permet de gérer un chronomètre pour évaluer les performances des algorithmes. Deux autres classes permettent de gérer différents labels. C'est dans cette librairie que l'algorithme de distance d'édition sera par la suite implémenté.

Vous trouverez le diagramme de classe correspondant à la librairie actuelle en Annexes (cf. Figure 19).

3.3 L'exportation avec dot et touchGraph

L'existant permet d'écrire le graphe au format dot puis d'utiliser dot.exe pour transformer un graphe en image au format jpeg ou png.

L'utilisation de l'exportation en version touchGraph permet de créer un graphe dynamique avec des nœuds cliquables et déplaçables à l'écran par l'utilisateur.

4 Représentation des graphes

4.1 Définitions

Commençons d'abord par définir un graphe.

Un graphe G=[X,U] est défini par :
- un ensemble X de sommets
- un ensemble U de paires de sommets. Si les paires sont ordonnées, ce sont des axes, sinon ce sont des arêtes.

On note u=(i,j) ϵ U, un arc avec (i,j) ϵ X, deux sommets. On appelle i le nœud source et j le nœud destination.

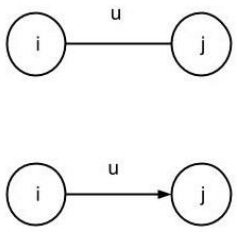
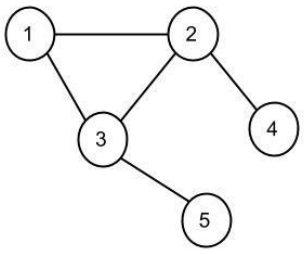


Figure 1. Arête (en haut) et arc (en bas)

Un graphe composé d'arcs est dit dirigé, tandis qu'un graphe composé d'arêtes est dit non dirigé. Les graphes peuvent être attribués, c'est-à-dire qu'un nœud ou un arc peut être associé à des attributs qui lui sont propre, qui le caractérise. Ces attributs permettent de faire le lien avec l'objet que le graphe représente. Par exemple, des coordonnées x et y pour les sommets du graphe d'une image. Si un graphe possède beaucoup d'attributs, on dit qu'il est fortement attribué.

La matrice d'adjacence associée à un graphe est une matrice de degré n avec n le nombre de sommets du graphe. La valeur (i,j) = 1 si les sommets i et j sont reliés et (i,j) = 0 s'ils ne le sont pas.



	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	0
3	1	1	0	0	1
4	0	1	0	0	0
5	0	0	1	0	0

Figure 2. Exemple de matrice d'adjacence

4.2 Définitions dans les librairies Graphs et Matching

Dans la librairie Graphs, un graphe est relié à une liste de Node et une liste d'Edge correspondants respectivement à ses sommets et à ses edges. De plus, un booléen permet de savoir si le graphe est dirigé ou non.

Les Edge sont reliés à un nœud source et un nœud cible. Les Node sont reliés à une liste d'Edge entrants et une liste d'Edge sortants. Lorsque le graphe est non dirigé, tous les Edge sont des Edge sortants. En plus de cela, les Edge et Node sont reliés à des Label qui permettent d'enregistrer les attributs liées à ces derniers. Toutes ces classes permettent de gérer un graphe attribué dynamiquement.

Dans la librairie Matching, l'arbre de recherche est représenté par la classe SearchTree. Cette classe permet à partir d'un graphe de construire toutes les matrices de permutations des Label de ce graphe, et ensuite générer l'arbre de recherche utilisé pour Corneil and Gotlieb sous forme d'un graphe. Un arbre peut être assimilé à un graphe car il possède le même fonctionnement de sommets et d'edges.

4.3 Les fichiers GXL

Le format GXL (Graph eXchange Language) est un format adapté au stockage de graphe. Les fonctions de chargement et de sauvegarde de graphes de la librairie se font à l'aide de fichier GXL. Ce format permet en effet de stocker un graphe avec un identifiant, une liste de nœuds et une liste d'edge avec leurs attributs. Ce format ressemble au format XML et fonctionne avec un système de balises prédéfinies. Vous trouverez un exemple de fichier au format GXL en Annexes

4.4 La classe Label

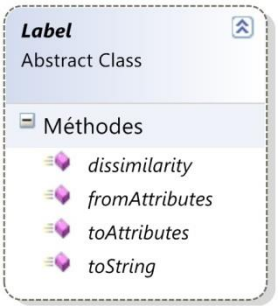


Figure 3. La classe label

La classe Label est une classe abstraite. Une classe héritant de la classe Label permet de modéliser une liste d'attributs pour les nœuds ou les arcs. Pour pouvoir comparer deux graphes, il faut créer les Label associés à leurs nœuds et arcs. Toutefois, la classe GenericLabel héritant elle-même de la classe Label s'adaptera à tout type de liste d'attributs, sans avoir à redéfinir un nouveau Label. Cette classe permet aussi grâce à la redéfinition de dissimilarity de transformer un modèle exact d'isomorphisme en modèle tolérant aux erreurs.

5 Description des algorithmes existants

L'existant possède deux algorithmes d'isomorphismes implémentés : Corneil and Gotlieb [CG70], et VF2 inspiré de l'algorithme VF de Cordella [Cor98] . Toutefois, certains tests ont montré que l'algorithme VF2 possédait des bugs à corriger.

5.1 Corneil and Gotlieb

L'algorithme de Corneil et Gotlieb est un précurseur dans l'isomorphisme de graphe. Le principe de cet algorithme est de retrouver le motif d'un graphe 1 dans un graphe 2. Pour cela, on utilise un certain type d'arbre de recherche.

Cet arbre de recherche se calcule à partir de toutes les matrices de permutations de la matrice d'adjacence de ce graphe. Pour obtenir une branche de l'arbre, il suffit de parcourir la matrice de permutations en diagonale en partant du haut à gauche de la matrice. Voici un exemple d'arbre de recherche pour un graphe à 3 nœuds :

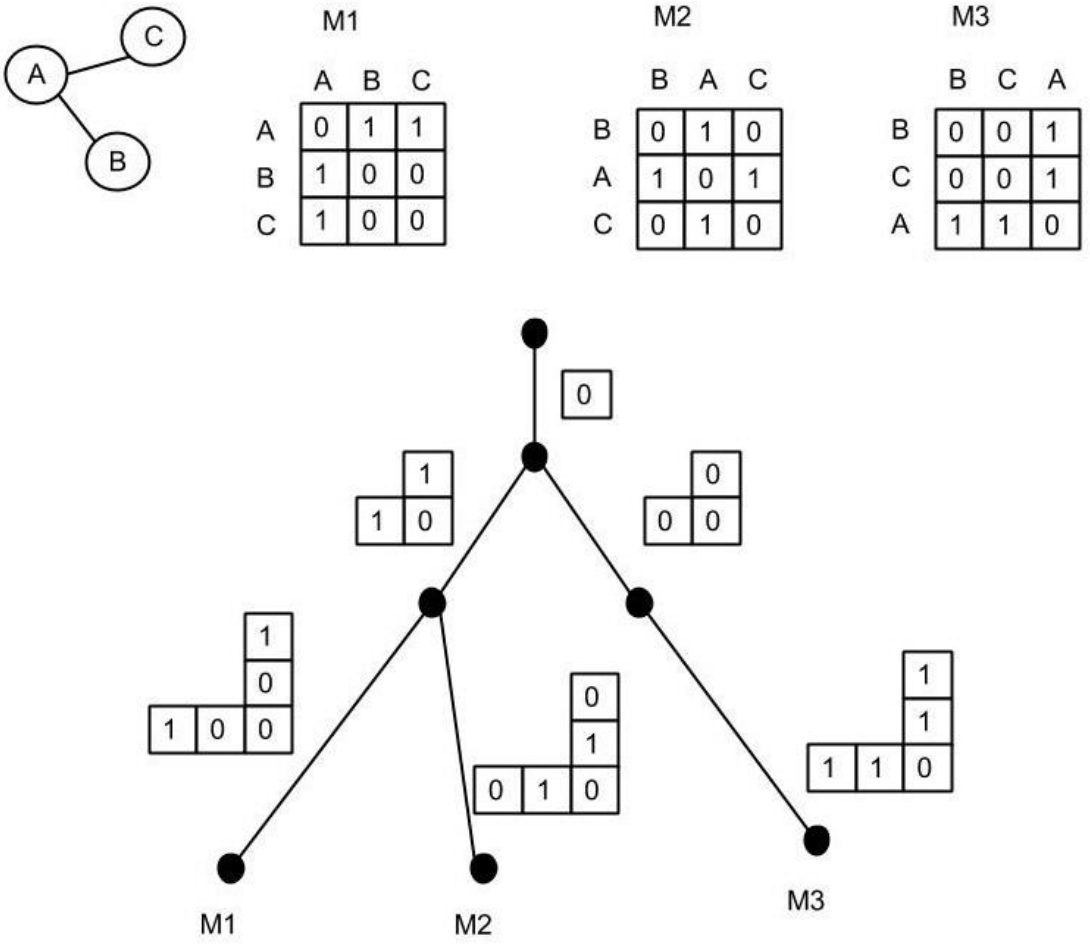


Figure 4. Exemple d'arbre de recherche

Chaque matrice de permutations rajoute une nouvelle feuille à l'arbre de recherche, ce qui donne au maximum N ! feuilles pour l'arbre de recherche d'un graphe à N nœuds (il existe N ! matrices de permutations, mais il peut y avoir redondance).

Le principe est ensuite de retrouver le motif de la matrice d'adjacence du graphe 1 dans l'arbre de recherche du graphe 2. L'inconvénient de cet algorithme est sa complexité car l'arbre de recherche se calcul en O(N !N ²), avec N le nombre de nœuds du graphe.

5.2 L'algorithme VF2

L'algorithme VF2 est une des méthodes les plus utilisées dans l'isomorphisme de graphe et de sous-graphe. Ceci est dû à sa complexité qui est de O(N !N) dans le pire des cas et de $O(N^3)$ dans le meilleur des cas. Le principe de cet algorithme est plus complexe. Il sépare les deux graphes en 4 catégories (In Mapping, To Mapping, From Mapping et Disconnected), en triant les nœuds par degré (nombre d'arrête relié au nœud) décroissant. En partant du nœud possédant le plus grand degré, l'algorithme va continuellement chercher des correspondances entre le graphe 1 et le graphe 2. Si tous les nœuds du graphe 1 trouvent un nœud correspondant dans le graphe 2, alors le graphe 1 est un sous graphe du graphe 2. De plus, si tous les nœuds du graphe 2 ont eux aussi trouvé un correspondant, alors le graphe 1 et le graphe 2 sont isomorphes.

6 Bilan sur l'existant

La librairie Graphs existante propose un moyen de représenter toutes sortes de graphes attribués. Elle permet de créer et de gérer les différents nœuds et edges du graphe facilement. En plus de cela, ces différents graphes peuvent être exportés sous différents formats.

La librairie Matching implémente déjà deux algorithmes d'isomorphismes que sont Corneil and Gotlieb et VF2. Toutefois ce dernier algorithme possède quelques erreurs qu'il faudra corriger.

Pour plus d'informations sur l'existant, n'hésitez pas à parcourir le rapport sur l'existant.

7 Modification des classes existantes

7.1 Implémentation des interfaces

La librairie Graphs possédait des interfaces correspondantes à différents types d'algorithmes utilisables sur les graphes. Ces interfaces étaient les suivantes :

- IIsomorphism (Isomorphisme de graphe)
- ISubGraphIsomorphism (Isomorphisme de sous-graphe)
- IGraphMatching (Matching de graphe)
- IGraphDistance (Distance entre graphe)

Après mon travail, les classes IsomorphismCandG et Vf implémentent les interfaces IIsomorphism, ISubGraphIsomorphism et IGraphMatching. Les valeurs de retour des fonctions présentes dans ces classes ont été modifiées pour s'adapter aux fonctions de ces interfaces. Ces classes possèdent maintenant deux attributs : un attribut de type ResultIsomorphism qui était anciennement la valeur de retour de l'algorithme principal et un attribut de type List<String[]> permettant de sauvegarder l'appariement des graphes correspondant à l'interface IGraphMatching.

7.2 Appariement de graphes

Les classes IsomorphismCandG et Vf ont été modifiées pour réaliser l'appariement entre les graphes. L'appariement représente la correspondance entre les nœuds du premier graphe avec ceux du second graphe. Voici un exemple d'appariement (les traits rouges correspondent à l'appariement) :

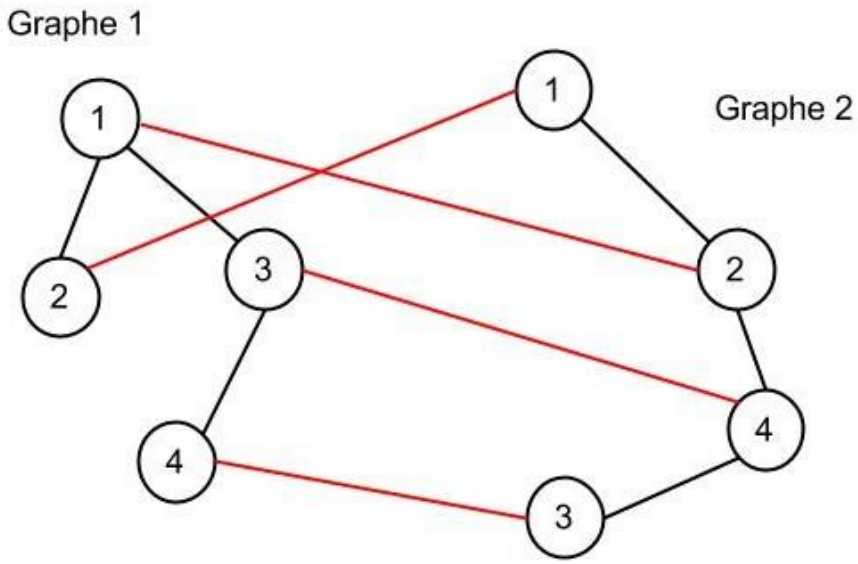


Figure 5. Appariement de graphes

L'appariement entre les graphes n'avaient pas été fait dans la version précédente de la librairie pour les classes précédemment citées. Il a donc fallu le réaliser pour implémenter l'interface GraphMatching.

Pour l'algorithme de Corneil & Gotlieb, il a fallu modifier la classe SearchTree. En effet les identifiants des nœuds de l'arbre de recherche étaient générés automatiquement et nous perdions alors l'information les reliant aux nœuds du graphe. L'identifiant des nœuds de l'arbre a donc été modifié de la manière suivante : _NumPermutation-IdNoeudGraphe. Le numéro de la permutation permet de ne pas perdre l'unicité de l'identifiant d'un nœud, tandis que l'identifiant du nœud d'un graphe permet de garder l'information concernant le nœud du graphe actuel.

Dans les deux algorithmes d'isomorphismes, la liste d'appariement est mise à jour à chaque fois qu'un nœud du premier graphe trouve sa correspondance avec un nœud du second graphe.

8 La distance d'édition et le BeamSearch

8.1 Les algorithmes

L'étude des différents algorithmes a été effectuée à l'aide de plusieurs articles présents dans la Bibliographie ([Neu06], [Rie09], [Fer10], [Gao10]). Ces différents articles expliquent le fonctionnement de la distance d'édition et de plusieurs dérivés de l'algorithme qui en découle. Toutefois le seul algorithme dérivé qu'il a été choisi de développer est le BeamSearch.

8.1.1 Description générale

La distance d'édition est un algorithme qui va évaluer la ressemblance entre deux graphes. En effet cet algorithme a pour but de renvoyer un coût correspondant au coût minimum de passage d'un graphe à l'autre. Cette évaluation du coût minimum de passage se fait en évaluant toutes les séquences d'opérations transformant un graphe G1 en un graphe G2.

Les opérations classiques sur les graphes sont les suivantes:

- l'insertion
- la suppression
- la substitution

Ces trois opérations peuvent être effectuées sur les nœuds et sur les arcs, ce qui fait un total de 6 opérations à prendre en compte. Chaque opération est associée à un coût variable qui permettra de calculer le coût total d'une suite d'opération.

L'algorithme de distance d'édition peut être assimilé à un arbre où chaque branche correspondrait à un chemin possible pour passer du graphe G1 au graphe G2. Chaque nœud de cet arbre possède un coût et un état permettant de connaître les nœuds ayant déjà subis des opérations et ceux restants. Les arcs reliant ces nœuds correspondent aux opérations effectués pour les obtenir. Ainsi pour obtenir la suite d'opération, il suffit de regarder la suite d'arcs

utilisés pour passer du nœud source à la feuille souhaitée. L'algorithme peut être résumé en étape de la façon suivante :

- On part d'un nœud du graphe G1 et on effectue toutes les opérations reliant ce nœud à ceux du graphe G2 (substitution avec des nœuds de G2 ou suppression de ce nœud). On obtient ainsi tous les premiers arcs partant du nœud source qui était l'état initial, et tous les nouveaux nœuds qui en découlent sont les nouveaux états après opérations.
- On se place sur le nœud feuille possédant le coût le plus bas et on recommence l'opération précédente à partir d'un nouveau nœud du graphe G1
- S'il ne reste plus de nœuds du graphe G1 à traiter, on teste la suppression des nœuds de G2
- On s'arrête dès qu'on arrive à un état où tous les nœuds ont été traités

8.1.2 Détail de l'algorithme

L'algorithme en version algorithmique se présente sous la forme suivante (tiré de l'article de Kaspar Riesen [Rie09]) :

Algorithm 1. Graph edit distance algorithm

Input: Non-empty graphs $g_1 = (V_1, E_1, \mu_1, \nu_1)$ and $g_2 = (V_2, E_2, \mu_2 \nu_2)$, where $V_1 = \{u_1, \ldots, u_{|V_1|}\}$ and $V_2 = \{v_1, \ldots, v_{|V_2|}\}$

Output: A minimum cost edit path from g_1 to g_2 e.g. $p_{\min} = \{u_1 \to v_3, u_2 \to \varepsilon, \ldots, \varepsilon \to v_2\}$

1: initialize *OPEN* to the empty set $\{\}$
2: For each node $w \in V_2$, insert the substitution $\{u_1 \to w\}$ into *OPEN*
3: Insert the deletion $\{u_1 \to \varepsilon\}$ into *OPEN*
4: **loop**
5: Remove $p_{\min} = \text{argmin}_{p \in OPEN}\{g(p) + h(p)\}$ from *OPEN*
6: **if** $p_{\min}$ is a complete edit path **then**
7: Return $p_{\min}$ as the solution
8: **else**
9: Let $p_{\min} = \{u_1 \to v_{i1}, \cdots, u_k \to v_{ik}\}$
10: **if** $k < |V_1|$ **then**
11: For each $w \in V_2 \setminus \{v_{i1}, \cdots, v_{ik}\}$, insert $p_{\min} \cup \{u_{k+1} \to w\}$ into *OPEN*
12: Insert $p_{\min} \cup \{u_{k+1} \to \varepsilon\}$ into *OPEN*
13: **else**
14: Insert $p_{\min} \cup \bigcup_{w \in V_2 \setminus \{v_{i1}, \cdots, v_{ik}\}} \{\varepsilon \to w\}$ into *OPEN*
15: **end if**
16: **end if**
17: **end loop**

Figure 6. Algorithme de distance d'édition

Description des variables et fonctions utilisés :

- La liste OPEN est une liste de chemin d'édition, c'est-à-dire une suite d'opération.
- p_{min} est la variable correspondant au chemin optimal tout au long de l'algorithme.
- g1 et g2 correspondent aux deux graphes
- V1 et V2 correspondent à la liste des nœuds de g1 et g2
- les fonctions g() et h() correspondent aux fonctions de coût utilisées avec h une heuristique appliquée pour le coût en plus de la fonction de base g
- k : nombre d'opérations de p_{min} à un moment donné

Déroulement de l'algorithme :

- La liste OPEN est initialisée à vide.
- On ajoute toutes les substitutions du nœud u_1 de g1 avec les nœuds w de g2, ainsi que la suppression du nœud u_1 dans OPEN.
- On rentre dans la boucle globale.
- On récupère le chemin optimal de OPEN que l'on associe à p_{min} et que l'on extrait de OPEN.
- Si p_{min} est un chemin complet (passage de g1 à g2 avec les opérations de p_{min}), alors on retourne p_{min}. C'est l'unique condition d'arrêt de la boucle, une condition qui sera toujours atteinte.
- Sinon, si k < Nombre de nœuds de g1 (on n'a pas encore traité tous les nœuds de g1), on ajoute les substitutions du nœud u_{k+1} avec les nœuds w de g2 qui n'ont pas encore été utilisés ainsi que la suppression de u_{k+1} dans OPEN.
- Sinon, si k > Nombre de nœuds de g2 (on a déjà traité tous les nœuds de g1), on ajoute l'insertion des nœuds de g2 qui n'ont pas encore été utilisés.
- On revient au début de la boucle globale

Opérations induites :

Ceci représente la base de l'algorithme mais beaucoup d'opérations sont cachées derrière les insertions, les suppressions et les substitutions des nœuds. Il faut notamment prendre en compte les opérations sur les arcs associées aux nœuds traités. Ces opérations induites sont les suivantes :

- insertion d'arcs :

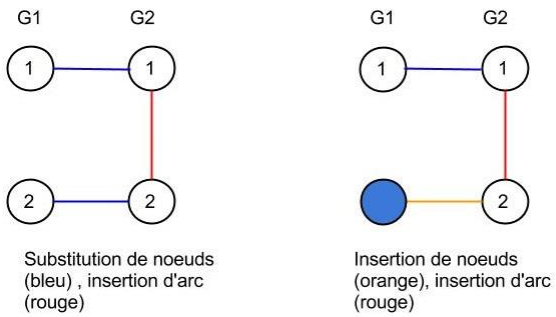


Figure 7. Insertion d'arc

Deux cas de figures peuvent se présenter. Lors d'une substitution de nœuds, le nœud de g2 substitué est relié avec un autre nœud de g2 déjà substitué à un nœud de g1 qui ne possède pas d'arc avec le nœud de g1 de la substitution courante. Lors de l'insertion d'un nœud de g2, ce nœud est relié à un nœud de g2 déjà traité.

- substitution d'arc :

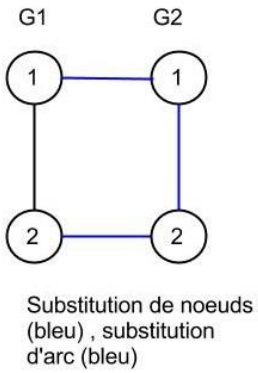


Figure 8. Substitution d'arc

Un seul cas de figure, lorsque les nœuds substitués de g1 et g2 possèdent tous deux un arc les reliant à deux autres nœuds substitués.

- suppression d'arc :

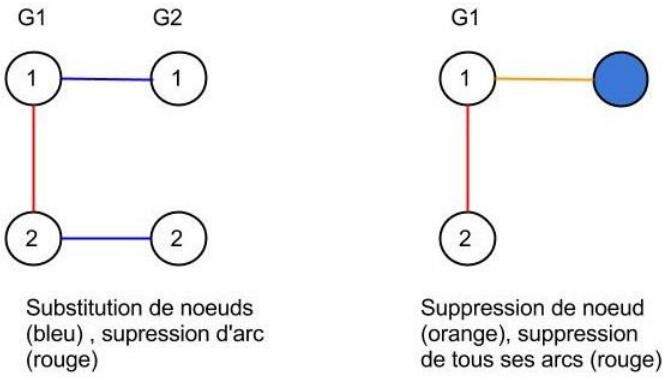


Figure 9. Suppression d'arc

Deux cas de figure peuvent se présenter. Lors d'une substitution de nœuds, le nœud de g1 est relié à un autre nœud de g1 déjà substitué à un nœud de g2 qui ne possède pas d'arc relié au nœud de g2 de la substitution courante. Lors de la suppression d'un nœud de g1, tous les arcs reliés à ce nœud sont supprimés.

8.1.3 Le BeamSearch

Le BeamSearch est une évolution de la distance d'édition qui permet d'obtenir une valeur approchée du coût minimum. Cette méthode a pour avantage d'être bien meilleure au niveau du temps et d'avoir des résultats se rapprochant assez bien du coût minimum.

Le principe est de ne pas garder tous les éléments de la liste OPEN mais de ne garder que les K meilleurs. Avant le calcul de p_{min} il faut trier la liste OPEN par éléments de coûts croissants. Ensuite on ne garde que les K premiers éléments de la liste OPEN triée.

8.2 La classe EditPath

Pour utiliser l'algorithme de distance d'édition, il a fallu définir une nouvelle classe EditPath correspondant à un chemin (une suite d'opérations).

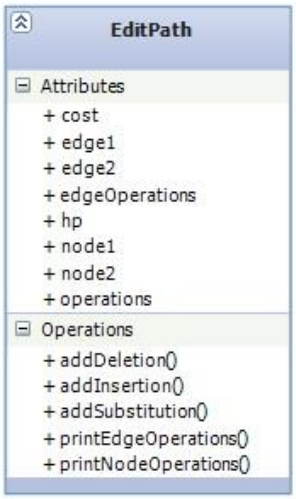


Figure 10. La classe EditPath

Les attributs :

- cost : correspond au cout de base sans heuristique. Lors de l'ajout d'une opération, on ajoute le coût de l'opération au cost actuel.
- hp : correspond au coût calculé avec heuristique. Dans notre cas, on choisit une heuristique minorante correspondant à la différence de taille entre les listes de nœuds node1 et node2 et les listes d'edge edge1 et edge2(éléments qui devront être insérés ou supprimés)
- les listes operations et edgeOperations sont respectivement des list<Node[]> des list<Edge[]> correspondant à les listes d'opérations sur les nœuds et d'opérations sur les edges.
- les listes d'Edge edge1 et edge2 , ainsi que les listes de Node node1 et node2 correspondent aux listes d'edges de g1 et g2 ainsi qu'aux listes de nœuds de g1 et g2 non utilisés.

Les méthodes :

- addDeletion, addInsertion et addSubstitution correspondent respectivement aux fonctions de suppression, d'insertion et de substitution de nœud.
- printEdgeOperations et printNodeOperations permettent de transformer les listes d'opérations sur les edges et sur les nœuds en String pour être afficher plus facilement par la suite.

Représentation des opérations :

- la substitution est représentée par un tableau {node1,node2}
- l'insertion est représentée par un tableau {null,node2}
- la suppression est représentée par un tableau {node1,null}

8.3 La classe GraphEditDistance

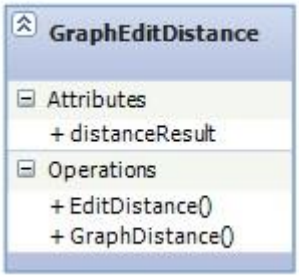


Figure 11. La classe GraphEditDistance

La classe GraphEditDistance est la classe qui correspond à la distance d'édition. Cette classe implémente l'interface IGraphDistance. La fonction EditDistance() permet de réaliser l'algorithme et distanceResult enregistre l'EditPath associé. Vous trouverez en Annexes l'implémentation de cette fonction. Elle ressemble beaucoup à l'algorithme vu en début de partie, avec openPath et bestPath correspondant respectivement à la liste OPEN et au p_{min}.

8.4 La classe BeamSearch

La classe BeamSearch est faite sur le même modèle que GraphEditDistance. La différence se trouve dans la fonction EditDistance(), au niveau du calcul de bestPath :

```
//Tri de la liste
   openPath.Sort(comparePath);

   //On ne garde que les k premiers
   if(openPath.Count>nbPathMax)
   {
       openPath.RemoveRange(nbPathMax, openPath.Count - nbPathMax);
   }

   //La liste est triée par coût croissant
   bestPath = openPath[0];
   openPath.RemoveAt(0);
```

Le tri de la liste se fait à l'aide de la fonction comparePath suivante :

```
private int comparePath(EditPath Path1, EditPath Path2)
     {
         double Cost1 = Path1.Cost + Path1.Hp;
         double Cost2 = Path2.Cost + Path2.Hp;
         return Cost1.CompareTo(Cost2);
     }
```

9 Le logiciel

Le logiciel de test créé est un logiciel permettant de tester tous les algorithmes de la librairie Matching avec un affichage des graphes testés à l'écran.

9.1 Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation du logiciel est le suivant :

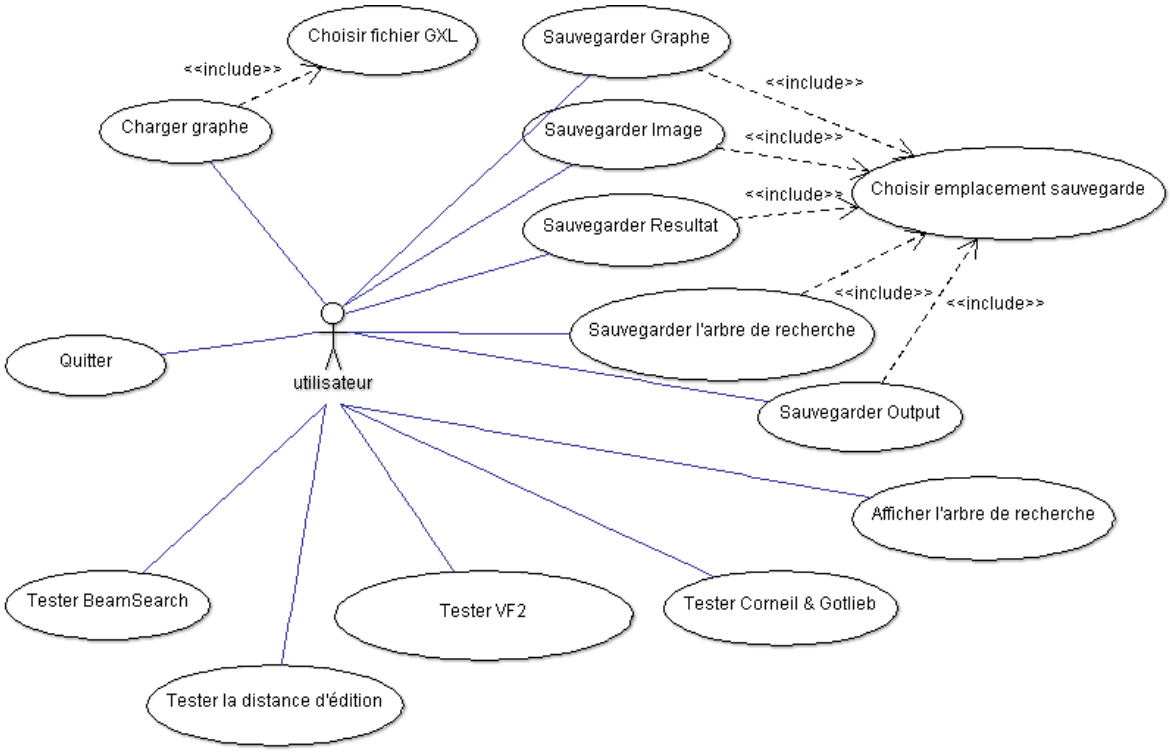


Figure 12. Diagramme de cas d'utilisation

9.2 Interface graphique

L'interface graphique principale se présente sous cette forme :

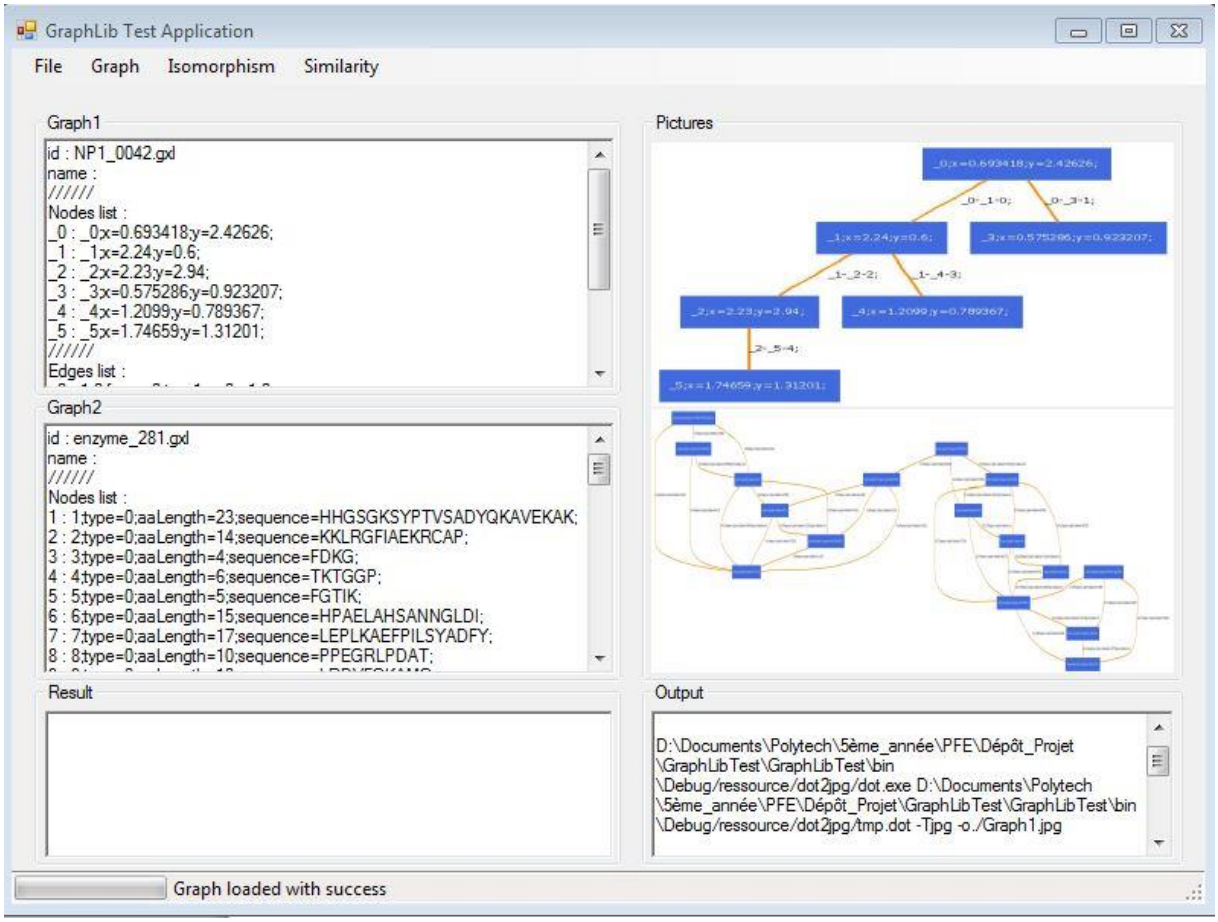


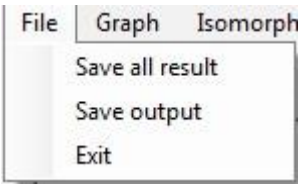
Figure 13. Interface de base

Elle est séparée en plusieurs parties :

- Une partie Graph1 et Graph2 permettant d'afficher les deux graphes en version textuelle (id, nom, liste des nœuds, liste des edges + affichage des attributs)
- Une partie pictures permet d'avoir affiché à l'écran les deux graphes en format image
- Une partie Output qui n'est autre qu'une redirection de l'output
- Une partie Result qui permet d'afficher les différents résultats des algorithmes

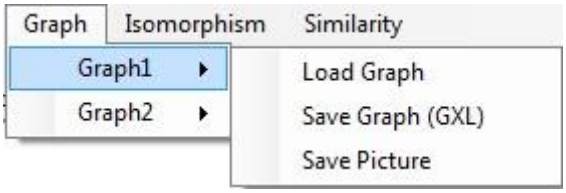
9.2.1 Description des Menus

Le menu file :



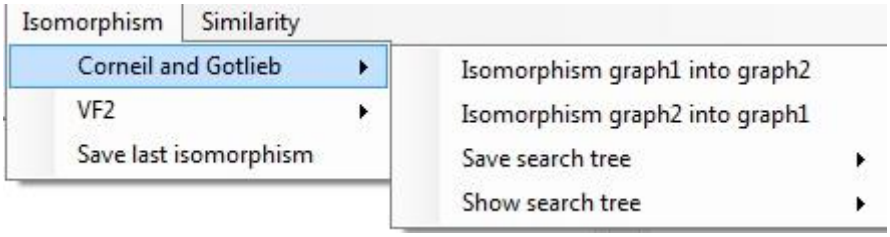
Il permet de sauvegarder les résultats présents dans la fenêtre de résultat et de sauvegarder le contenu de la fenêtre d'Output. De plus il permet de quitter l'application.

Le menu Graph :



Le menu Graph donne accès aux Graph1 et Graph2. Il permet de charger les graphes dans le logiciel ou de les sauvegarder en format GXL ou sous forme d'image.

Le menu Isomorphism :



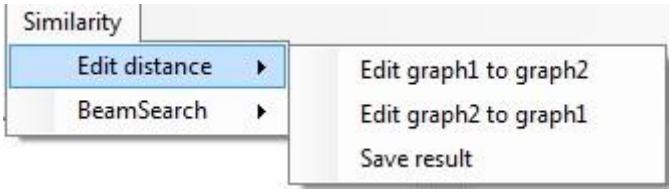
Le menu Isomorphism permet d'accéder aux algorithmes d'isomorphismes.

Le sous-menu VF2 permet d'exécuter l'algorithme VF2 en choisissant d'effectuer un test d'isomorphisme de graph1 dans la graph2 ou l'inverse.

Le sous-menu Corneil and Gotlieb, en plus de pouvoir tester l'isomorphisme du graph1 dans le graph2 et inversement, permet aussi d'accéder aux arbres de recherches associés. Ces arbres de recherche peuvent être affichés ou sauvegardés. Dans le cas de l'affichage, une fenêtre de type TreeWindow s'ouvrira (cf. 9.2.4).

Le menu propose aussi d'enregistrer les résultats du dernier isomorphisme effectué.

Le menu Similarity :



Le menu Similarity permet d'effectuer les tests de distance d'édition pour passer du graph1 vers le graph2 ou inversement. Il permet aussi d'enregistrer le dernier résultat.

Le test du BeamSearch peut aussi être effectué via ce menu. Lors de son utilisation, une fenêtre s'affichera pour laisser à l'utilisateur le choix de la valeur K souhaitée.

9.2.2 Les résultats

Les résultats des isomorphismes sont affichés de la façon suivante :

```
//////////////////////////////////////////////////
Isomorphism :
    Graph1 : AP1_0100.gxl
    Graph2 : FP1_0039.gxl
    AP1_0100.gxl into FP1_0039.gxl
    SubGraph : False
    Graph : False
    Depth of deepest node : 0
//////////////////////////////////////////////////
```

Les deux premières lignes correspondent aux identifiants des graph1 et graph2 permettant de reconnaître les graphes. La troisième ligne indique le sens de l'isomorphisme (graph1 dans graph2 ou l'inverse). Les 4ème et 5ème lignes indiquent si c'est un isomorphisme de sous-graphe ou un isomorphisme de graphe. La dernière ligne indique la profondeur de la recherche pour obtenir ce résultat.

Les résultats des algorithmes de distance sont affichés de la façon suivante :

```
//////////////////////////////////////////////////
Edit Distance :
    Graph1 : AP1_0100.gxl
    Graph2 : FP1_0039.gxl
    AP1_0100.gxl to FP1_0039.gxl
    Node Operations realized :
    [| s _0 _0 || s _1 _1 || s _2 _4 || s _3 _2 || s _4 _3 |]
    Edge Operations realized:
    [| s _0-_1-0 _0-_1-0 || i eps _0-_4-1 || s _1-_2-1 _1-_4-2 || s _3-_4-2 _2-_3-3 |]
    Cost :6
//////////////////////////////////////////////////
```

Les deux premières lignes correspondent aux identifiants des graph1 et graph2 permettant de reconnaître les graphes. La troisième ligne indique le graphe source et le graphe cible (graph1 vers graph2 ou l'inverse). Les lignes suivantes permettent de connaître les opérations sur les nœuds et les opérations sur les arcs réalisés. La dernière ligne correspond au coût de ce passage.

9.2.3 La fenêtre isoWindow

La fenêtre isoWindow est de la forme suivante :

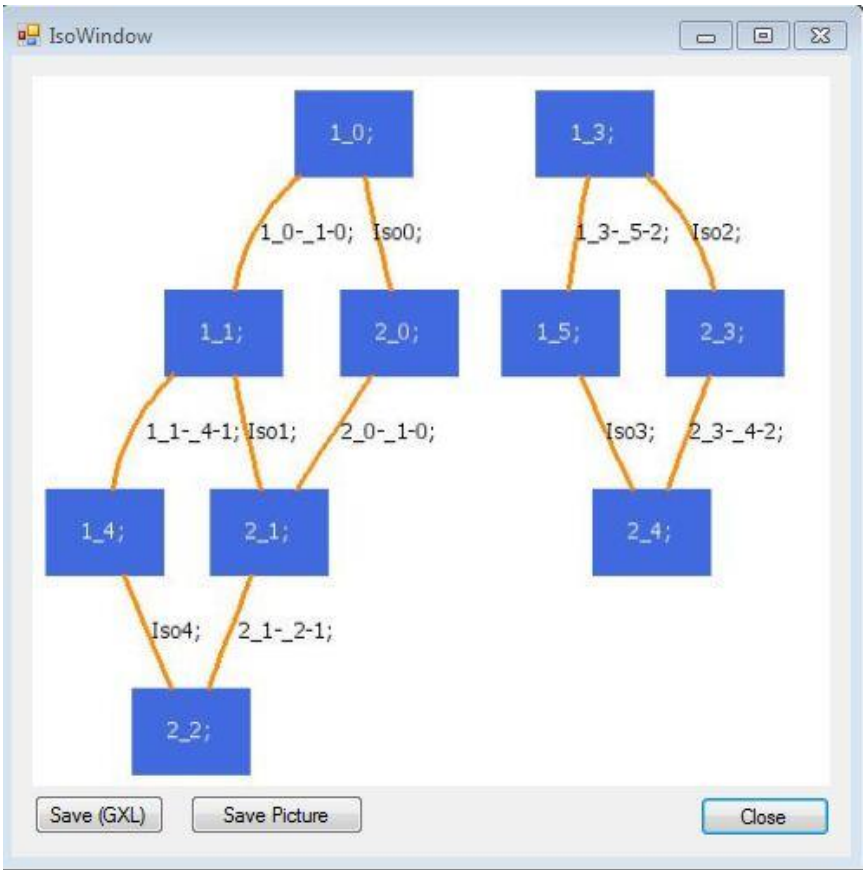


Figure 14. Interface IsoWindow

La fenêtre IsoWindow permet d'afficher l'appariement entre les deux graphes qui ont subis un isomorphisme ou les deux graphes qui ont été évalués via une fonction de distance (avec la fonction de distance, un nouveau nœud espsilon apparait à l'écran pour gérer les substitutions et les insertions de nœuds).

Cette fenêtre permet de sauvegarder le graphe d'appariement obtenu sous format GXL ou sous format image.

9.2.4 La fenêtre treeWindow

La fenêtre treeWindow est de la forme suivante :

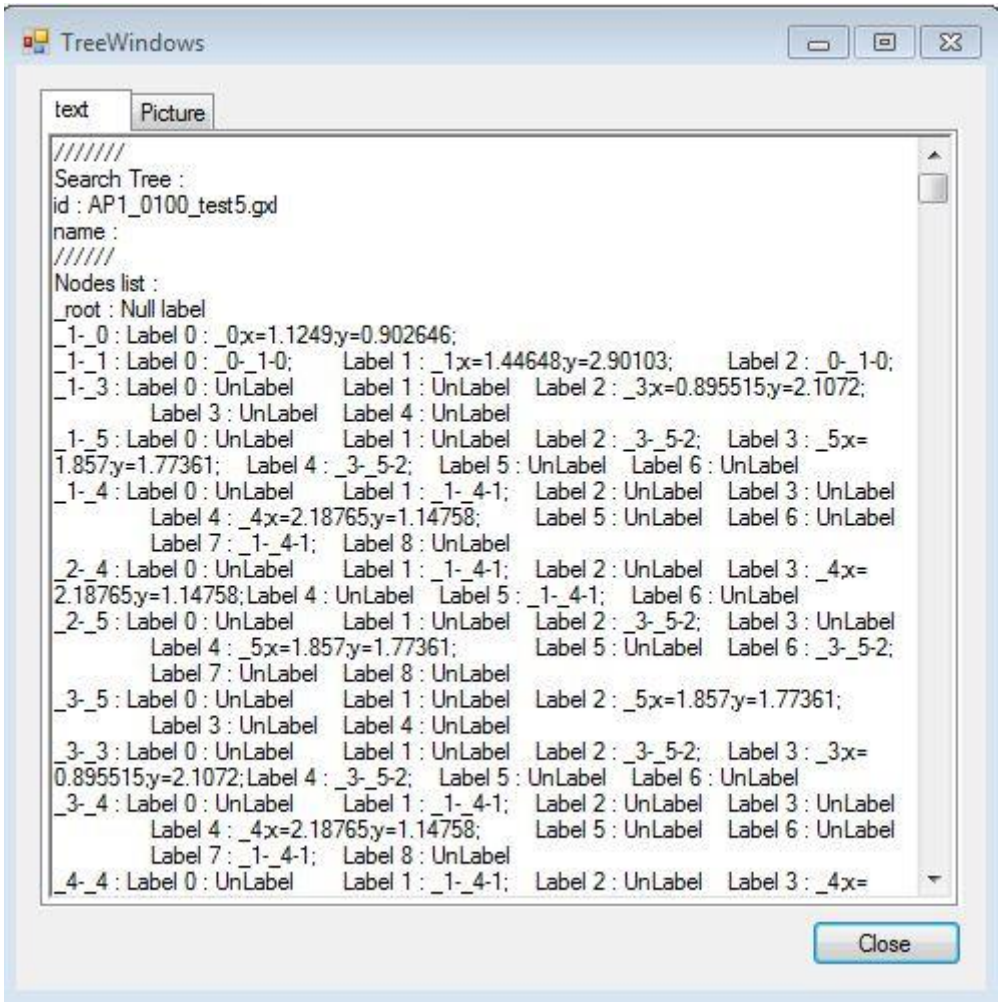


Figure 15. Interface TreeWindow

La fenêtre treeWindow est la fenêtre qui s'ouvre lorsque l'on veut afficher l'arbre de recherche associé au graphe. La version « Picture » n'est volontairement pas affichée car l'arbre est trop grand et n'arrive pas à être enregistré au format Jpeg avec un graphe de 5 nœuds. Toutefois, pour un graphe de 5 nœuds, il est possible de l'afficher au format PNG.

9.3 Procédure d'utilisation du logiciel

Le logiciel est simple d'utilisation. Toutefois pour utiliser un algorithme d'isomorphisme ou de similitude, il faut s'assurer d'avoir déjà chargé deux graphes dans l'interface. Sinon, le logiciel renverra un message d'erreur. Pour afficher l'arbre de recherche d'un graphe il faut s'assurer que le graphe en question soit bien chargé dans le logiciel.

Lorsqu'un algorithme est lancé (Isomorphisme ou Similarity), il est impossible à l'utilisateur de pouvoir lancer un deuxième algorithme dans le même temps (menus inaccessibles). Il faut attendre que le calcul en cours soit finit avant d'en lancer un deuxième. Il est toutefois possible d'accéder à l'interface pendant qu'un calcul est en cours (gestion de threads).

10 Résultats des tests

Les tests ont été réalisés sur les algorithmes de distance d'édition et l'algorithme BeamSearch. Lorsqu'il s'agit de l'algorithme de BeamSearch, le K utilisé est précisé.

Il n'est pas facile d'évaluer l'efficacité de l'algorithme de distance d'édition, car le temps de réalisation et les résultats obtenus vont varier en fonction des 2 graphes utilisés sans avoir de liens direct avec le nombre de nœuds de ces graphes. Il a donc été choisi d'effectuer différents types de test. Les premiers tests effectués sont des tests de comparaison entre les résultats obtenus avec la distance d'édition et le BeamSearch en terme de valeur et de temps (la valeur doit toujours être minimale pour la distance d'édition). Ces tests doivent être effectués sur des petits graphes à cause de la complexité de la distance d'édition. Les tests suivants permettent ensuite d'évaluer les valeurs obtenues avec le BeamSearch uniquement en faisant varier la valeur K sur des gros graphes. Un dernier test permet de comparer l'algorithme de distance d'édition implémenté ici à un algorithme existant.

Les tests ont été réalisés en comparant l'ensemble des graphes d'un dossier entre eux. Ces dossiers de graphes sont tirés de dossiers existants et disponibles sur internet. Les tests utilisant la distance d'édition ont été réalisés sur un dossier Letter (150 graphes de lettres (10 de chaque lettre)) appartenant au dossier Letter->LOW. Ceux utilisant uniquement le BeamSearch ont été réalisés sur un dossier Protein (50 Protein au hasard) appartenant au dossier Protein.

Les graphes de type Letter sont des petits graphes peu attribués, tandis que les graphes de type Protein sont des gros graphes plus attribués.

10.1 Résultats en terme de valeur

10.1.1 Dossier Letter

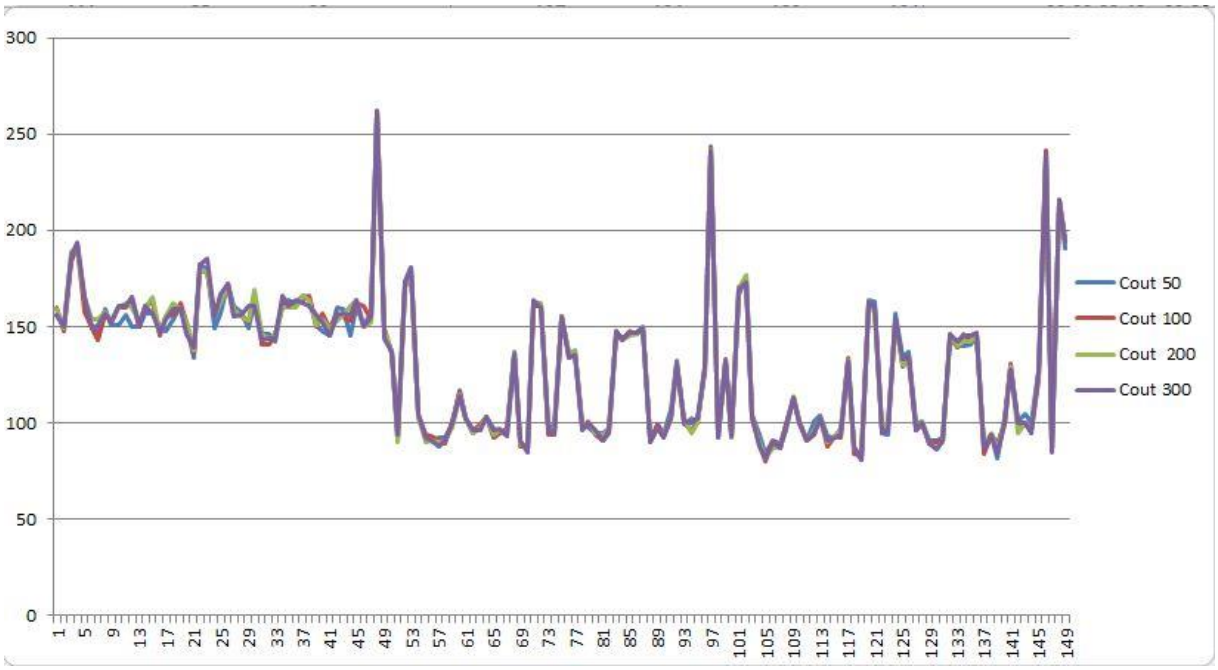
Le dossier a été testé avec la distance d'édition de base et avec plusieurs tests de BeamSearch (plusieurs valeurs de K). Voici les résultats obtenus sur les 150 premiers résultats :

Figure 16. Résultat obtenus sur le dossier Letter

Le coût Min correspond à la distance d'édition de base, tandis que les autres coûts correspondent à un BeamSearch de K = coût. On voit bien ici que seul le K=20 donne des résultats moins bons que la distance d'édition de base. Même avec un K=50, la valeur de coût minimum est la bonne.

10.1.2 Dossier Protein

Le dossier a été seulement testé sur le BeamSearch. Le seul test effectué sur la distance d'édition entre les deux premiers graphes du dossier a été arrêté au bout de 5 heures d'exécution. Voici les résultats obtenus sur les 150 premiers résultats :



Les résultats sont intéressants, car on y décèle quelques incohérences. En effet si pour les letter la valeur minimum de coût pouvait être atteinte rapidement, pour les protéines elle n'est parfois jamais atteinte. Les résultats sont tous à peu près les mêmes pour chaque calcul mais ce n'est pas tout. Nous pouvons observer sur la courbe, des résultats de BeamSearch avec un K=50 possédant un meilleur coût qu'un BeamSearch avec un K=300. Nous pouvons nous demander alors, dans le cas où la valeur minimale ne peut être atteinte, si augmenter la valeur de K apporte un réel intérêt sur le résultat obtenu.

10.2 Résultats en terme de temps d'exécution

10.2.1 Dossier Letter

Les résultats obtenus avec le dossier Letter en terme de temps tournent tous au niveau de 0 secondes. On peut voir une augmentation du temps avec l'augmentation du K pour le BeamSearch. Le coût au niveau temps peut d'ailleurs être supérieur à celui d'une distance d'édition de base, pour un BeamSearch avec un K élevé. Ceci est sûrement dû au tri de la liste OPEN dans le BeamSearch qui demande un certain temps.

10.2.2 Dossier Protein

On peut faire le même constat dans le dossier Protein avec des résultats toutefois un peu plus long que la seconde. Le temps croît lorsque la valeur de K s'agrandit. Vu qu'aucun test avec la distance d'édition de base n'a donné de résultat, il est difficile de la prendre en considération pour les tests sur ce dossier.

10.3 Comparaison avec un algorithme existant

La distance d'édition a pu être comparée avec une version Java de cette dernière, créée par l'un de mes encadrants (Romain Raveaux). Les résultats obtenus sont très bons car en terme de valeurs, on obtient la même chose avec l'algorithme implémenté en C#. Voici le résultat obtenu sur le dossier Letter précédemment testé :

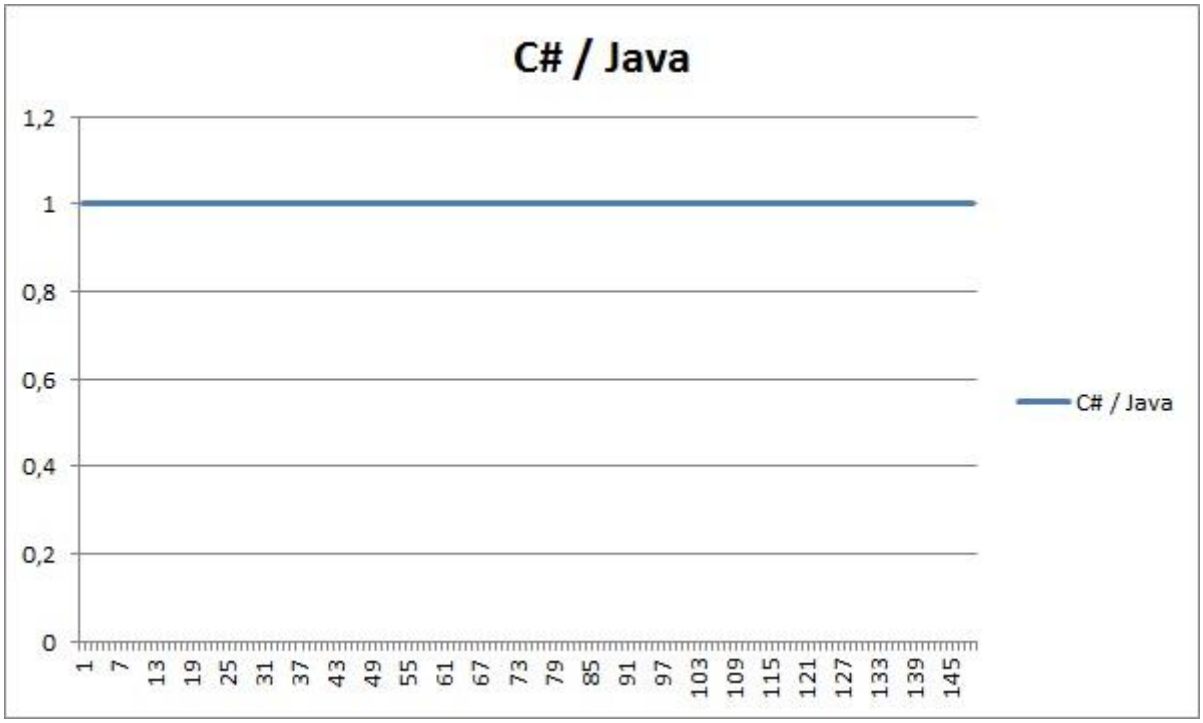


Figure 17. Comparaison C# et Java

Les résultats obtenus en C# et en Java ont été divisés pour obtenir une courbe constante égale à 1 pour les 150 premiers résultats de tests.

Les temps de calcul évoluent de la même manière avec le code C# et le code Java. Il n'y a pas de gagnant au niveau temps car le meilleur temps est parfois obtenu avec la version C# et parfois obtenu avec la version Java.

11 Les améliorations possibles

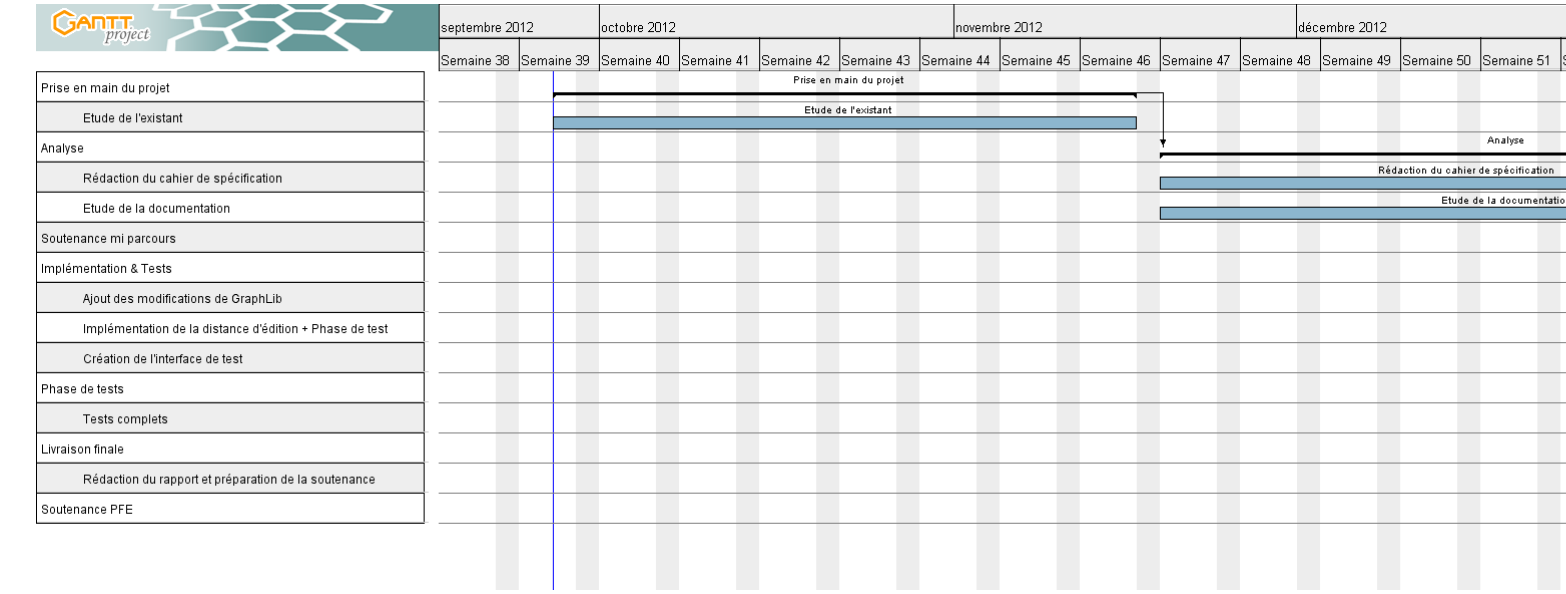
11.1 Choix des valeurs de coût

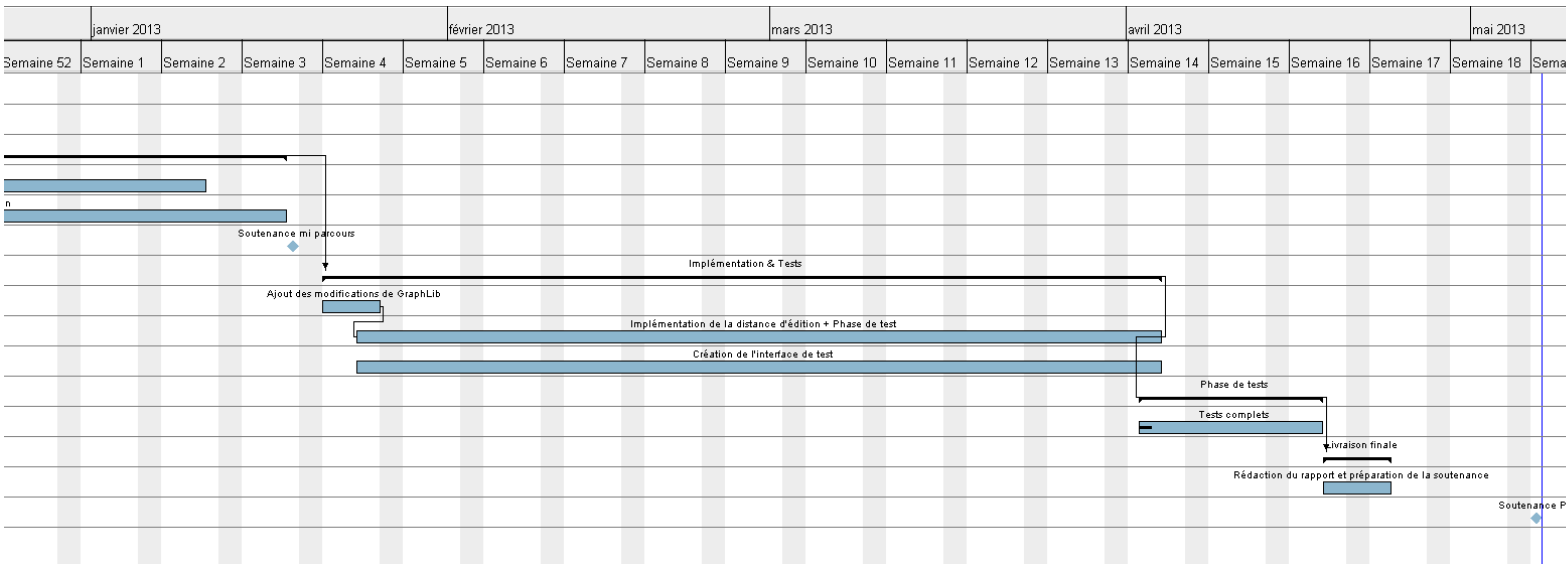
Le gros problème de la version C# de la distance d'édition est le coût au niveau de l'insertion et de la suppression d'un nœud ou d'un arc. En effet ces coûts sont évalués dans le code à 1. Seule la substitution est évaluée en fonction de la dissimilarity entre Label qui peut donc être modifiée par un programmeur.

La solution à ce problème est de tout gérer à partir de la dissimilarity et de ne plus utiliser un nœud ou edge null pour l'insertion et la substitution. Il faudrait un GraphComponent spécial de type epsilon, qui puisse donc être adapté aux nœuds et aux edges, et qui soit pris en compte dans la dissimilarity des labels.

12 Planning

Le planning initialement créé dans le cahier de spécification n'a pas été correctement respecté pour de multiples raisons et choix pris avec les encadrants. En effet, la distance d'édition a été revue à de multiples reprises, pour être modifiée, améliorée ou même testée sous différentes versions. De plus le logiciel de Picture2Graph a été abandonné au cours du projet, pour se concentrer sur la réalisation de la distance d'édition, plus importante pour la librairie. Le planning suivi est représenté par le diagramme de Gantt suivant :





Conclusion

Ce document m'a permis de détailler toutes les étapes et tout le travail qui a été réalisé durant ce projet. L'objectif principal de ce projet était d'ajouter un nouvel algorithme à la librairie et de créer un logiciel permettant de tester ces algorithmes, ce qui a été mené à bien.

La première partie de ce projet a été consacrée à la prise en main de l'existant avec la compréhension du système de label créé par mon prédécesseur.

La seconde partie fût consacrée à l'amélioration de la librairie et à l'utilisation de cette dernière dans un nouvel algorithme et dans un logiciel approprié.

Bien que certains objectifs initiaux n'aient pas été atteints, ce projet m'aura beaucoup appris sur la modélisation de projet, un point très important pour la réalisation d'un projet. De plus, il aura enrichi mes connaissances à propos des graphes et sur un langage très intéressant et très puissant qu'est le C#.

Glossaire

GXL : Graph eXchange Language
Edge : Dans ce document, le mot edge sera utilisé pour qualifier un arc ou une arête.

Bibliographie

[CG70] D. G. Corneil , C. C. Gotlieb, “An Efficient Algorithm for Graph Isomorphism”, in Journal of the ACM (JACM), v.17 n.1, p.51-64, Jan. 1970.

[Cor98] L.P. Cordella, P. Foggia, C. Sansone, F. Tortorella, M. Vento, “Graph Matching: A Fast Algorithm and its Evaluation”, Proc. of the 14th International Conference on Pattern Recognition, Brisbane, Australia, August, 16-20, pp. 1582-1584, 1998.

[Neu06]Michel Neuhaus, Kaspar Riesen, and Horst Bunke , “Fast Suboptimal Algorithms for the Computation of Graph Edit Distance”, SSPR&SPR 2006, LNCS 4109, pp. 163–172, 2006.

[Rie09]Kaspar Riesen , Horst Bunke , “Approximate graph edit distance computation by means of bipartite graph matching”, in Image and Vision Computing 27 , pp. 950-959, 2009.

[Fer10]M. Ferrer a, E.Valveny a , F.Serratosa b, K.Riesen c, H.Bunke c, “Generalized median graph computation by means of graph embedding in vector spaces”, in Pattern Recognition 43, pp. 1642-1655, 2010.

[Gao10]Xinbo Gao Æ Bing Xiao Æ Dacheng Tao Æ Xuelong Li, “A survey of graph edit distance”, in Pattern Anal Applic, pp 113-129, 2010.

Diagramme de classe Graphs :

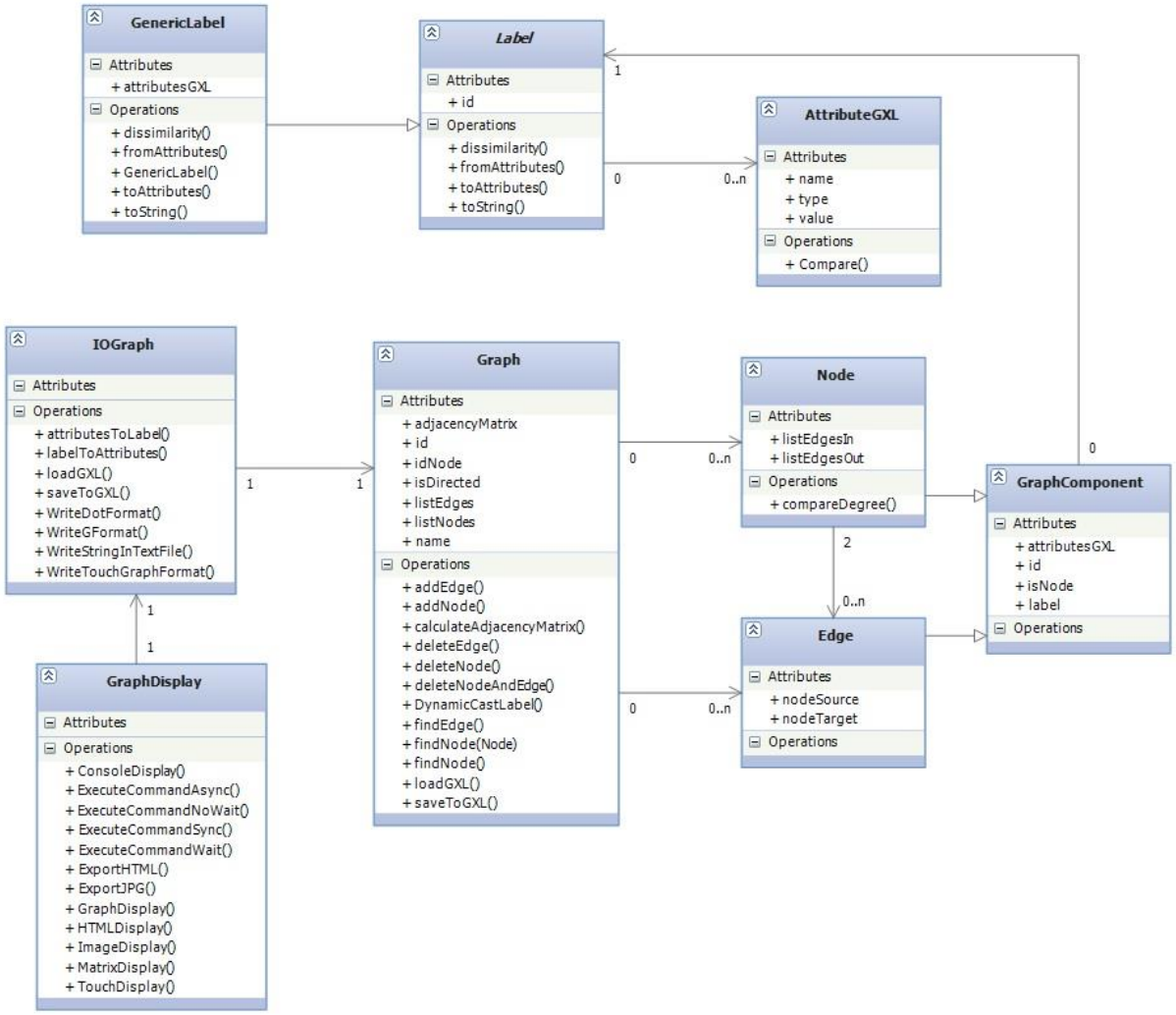


Figure 18. Diagramme de classe Graphs

Diagramme de classe Matching :

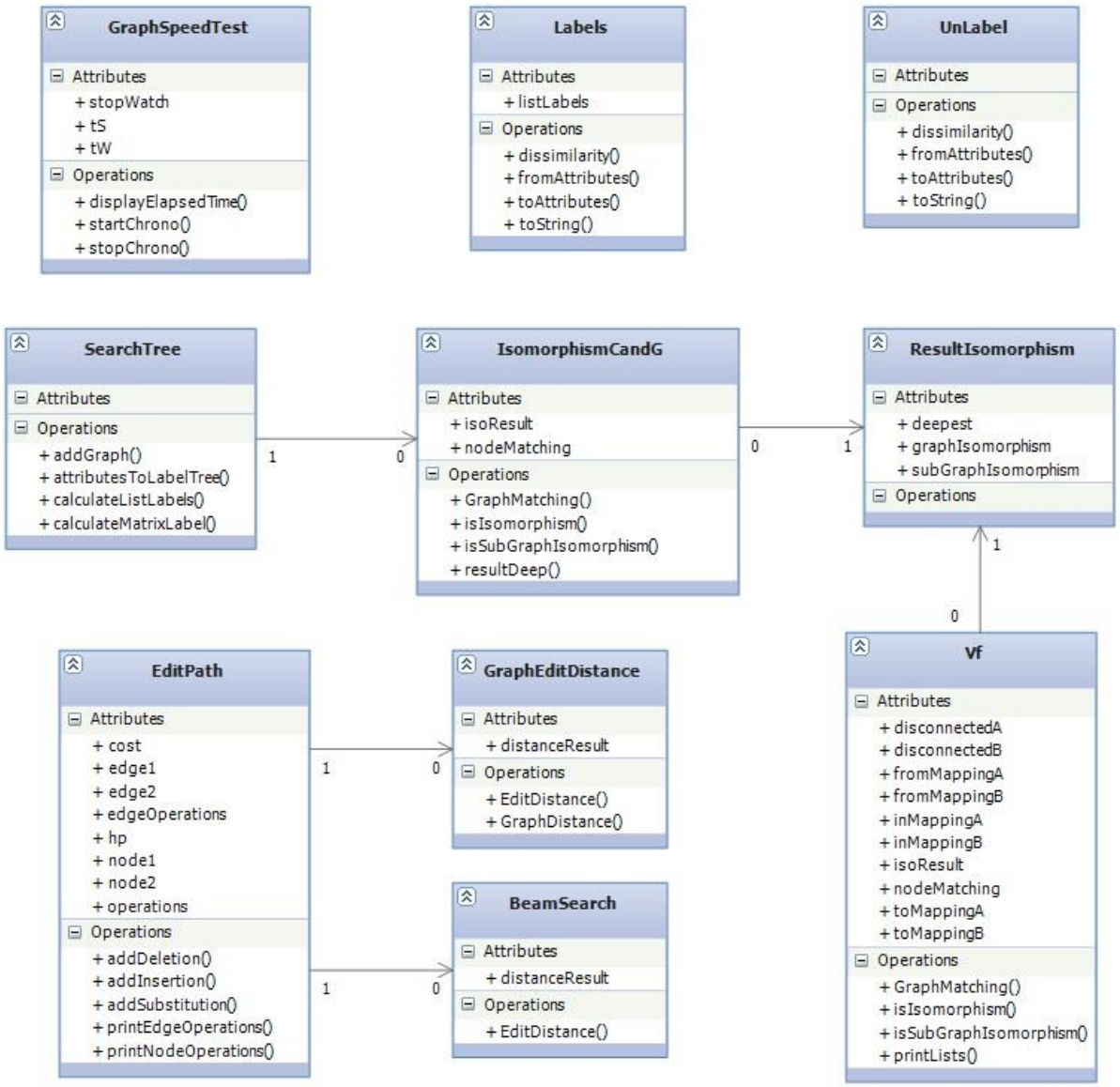


Figure 19. Diagramme de classe Matching

Exemple de fichier GXL :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd">
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">

<graph id="AP1_0004" edgeids="false" edgemode="undirected">

<node id="_0"><attr name="x"><float>0.742574</float></attr><attr
name="y"><float>0.787024</float></attr></node>
<node id="_1"><attr name="x"><float>1.49378</float></attr><attr
name="y"><float>2.60672</float></attr></node>
<node id="_2"><attr name="x"><float>2.21823</float></attr><attr
name="y"><float>0.767851</float></attr></node>
<node id="_3"><attr name="x"><float>0.919723</float></attr><attr
name="y"><float>1.5454</float></attr></node>
<node id="_4"><attr name="x"><float>2.20976</float></attr><attr
name="y"><float>1.43692</float></attr></node>

<edge from="_0" to="_1"/>
<edge from="_1" to="_2"/>
<edge from="_3" to="_4"/>

</graph>
</gxl>
```

Fonction EditDistance() :

```
public EditPath EditDistance(Graph _Graph1, Graph _Graph2)
        {

            int maxNode = Math.Max(_Graph1.ListNodes.Count, _Graph2.ListNodes.Count);

            List<EditPath> openPath = new List<EditPath>();
            EditPath bestPath = new EditPath();

            //Fin des initialisations
            EditPath currentPath;
            for (int i = 0; i < _Graph2.ListNodes.Count; i++)
            {
                currentPath = new EditPath(_Graph1.ListNodes, _Graph2.ListNodes,
_Graph1.ListEdges, _Graph2.ListEdges);
                currentPath.addSubstitution(_Graph1.ListNodes[0],
_Graph2.ListNodes[i]);
                openPath.Add(currentPath);
            }
            currentPath = new EditPath(_Graph1.ListNodes, _Graph2.ListNodes,
_Graph1.ListEdges, _Graph2.ListEdges);
            currentPath.addDeletion(_Graph1.ListNodes[0]);

            double costMin;
            int indexMin;

            int cptAff = 5000;

            while (true)
            {

                costMin = double.MaxValue;
                indexMin = -1;

                //Debuggage
                if (openPath.Count > cptAff)
                {
                    Console.WriteLine("Taille Open: " + openPath.Count);
                    Console.WriteLine("Nombre d'opérations dans PMIN : " +
bestPath.Operations.Count + " / " + maxNode);
                    Console.WriteLine(bestPath.printNodeOperations());
                    cptAff = cptAff + 5000;
                }


                //On recherche le chemin au cout minimum parmis les chemins dans OPEN
                for (int i = 0; i < openPath.Count; i++)
                {
                    if (openPath[i].Cost + openPath[i].Hp < costMin)
                    {
                        costMin = openPath[i].Cost + openPath[i].Hp;
                        indexMin = i;
                    }
                }
                bestPath = openPath[indexMin];
                openPath.RemoveAt(indexMin);

                if ((bestPath.Operations.Count == maxNode))
                {
                    return bestPath;
                }
```

```
                //Sinon on ajoute les nouvelles opérations dans OPEN
                else
                {
                    //Si tous les noeuds de G1 n'ont pas été traités on continue
d'ajouter des substitutions et la suppression
                    if (bestPath.Operations.Count < _Graph1.ListNodes.Count)
                    {
                        for (int i = 0; i < bestPath.Node2.Count; i++)
                        {
                            currentPath = new EditPath(bestPath);
                            currentPath.addSubstitution(currentPath.Node1[0],
currentPath.Node2[i]);
                            openPath.Add(currentPath);
                        }
                        currentPath = new EditPath(bestPath);
                        currentPath.addDeletion(currentPath.Node1[0]);
                        openPath.Add(currentPath);
                    }
                    //Sinon on ajoute l'insertion
                    else
                    {
                        for (int i = 0; i < bestPath.Node2.Count; i++)
                        {
                            currentPath = new EditPath(bestPath);
                            currentPath.addInsertion(currentPath.Node2[i]);
                            openPath.Add(currentPath);
                        }
                    }
                }
            }
        }
```

GraphLib : une librairie C# pour l'exploitation de graphes en reconnaissance des formes

Département Informatique
5ème année
2012-2013

Rapport de Projet de Fin d'Etudes

Résumé : Ce document traite de la réalisation d'une librairie en C# portant sur la création et la gestion de graphes. Cette librairie propose aussi plusieurs types d'algorithmes portant sur le calcul d'isomorphisme et de similarités entre graphes. Le projet porte aussi sur la réalisation d'une interface permettant de tester ces différents algorithmes.

Mots-clés : Graphes, isomorphismes, similarité, distance d'édition, librairie, C#

Abstract : This document deals with the realisation of a C# library about graph creation and operation. This library propose a lot of algorithm about isomorphism and similarity computation between graphs. This project also deals with an interface realization, permitting to test its different algorithms

Keywords : Graph, isomorphism, similarity, edit distance, library , C#

Encadrants :
Jean-Yves Ramel
jean-yves.ramel@univ-tours.fr
Romain Raveaux
romain.raveaux@univ-tours.fr

Etudiant :
Sébastien Schaal
sebastien.schaal@etu.univ-tours.fr

Année 2012-2013
DI5 – Promotion 2013