Projet de Recherche
Création d'une interface graphique pour un prototype de logiciel de reconnaissance de symboles.

Antoine MERCIER et Philippe SACHOT

Master 1ère année
Mention Informatique, Mathématiques et leurs Applications
Université de La Rochelle

Responsables :
Stéphanie Guillas et Karell Bertet

9 mai 2007

Table des matières

Introduction

Dans le cadre de notre première année de Master Informatique, Mathématique et leurs Applications, nous avons à réaliser un projet de recherche. Celui-ci consiste en la réalisation d'une interface graphique pour un prototype de logiciel de reconnaissance de symboles dans des images bruitées. Ce prototype a été élaboré lors de recherches réalisées dans le cadre de la thèse de Stéphanie Guillas, elle-même encadrée par Karell Bertet et Jean-Marc Ogier. Comme illustré dans la figure 1, il s'agit de tester diverses méthodes d'apprentissage et de reconnaissance de symbole afin d'obtenir le plus rapidement possible le résultat le plus fiable. Pour ce faire, nous avons une base d'images "modèles" ainsi qu'une base d'images "détériorées" sur lesquelles nous pouvons effectuer tous nos essais.

FIG. 1 – Problématique

Après avoir pris connaissance de l'architecture existante du logiciel, nous avons pu constater que la charge de travail était conséquente et qu'elle ne s'arrêtait pas à la seule création de cette interface graphique. En effet, cette architecture était le fruit d'ajouts successifs de fonctionnalités dont certaines sont devenues obsolètes et d'autres trop complexes. Suite à cette constatation, nous avons défini avec nos responsables de projet les différentes étapes à accomplir afin de le mener à bien.

Une double finalité apparaît clairement lors de l'analyse globale de ce projet. La première finalité est orientée recherche : nous allons devoir étudier et comprendre les principes utilisés dans cette étude de la reconnaissance de symbole automatique. De plus, comme nous allons devoir recréer une interface graphique, nous devrons réfléchir à celle-ci de façon à ce qu'elle soit la plus conviviale possible tout en étant modulaire. Une seconde finalité est orientée développement puisque nous allons certes devoir faire de l'analyse mais aussi de la programmation pure.

Chapitre 1

Analyse de l'existant

1.1 Description globale du système

Ce système de reconnaissance se divise en deux principales étapes : une phase d'apprentissage à partir de symboles d'"apprentissage" et une phase de reconnaissance de symboles détériorés (cf. Annexes D et E).

La phase d'apprentissage commence par le calcul des signatures propres aux modèles de symboles. Ces signatures sont un nombre déterminé de valeurs caractérisant l'image de ce symbole. Les données sont ensuite normalisées avant d'être organisées sous forme d'une table que l'on découpe à l'aide d'un critère choisi (Distance Maximum, Hotelling ou Entropie). Un treillis de Galois (cf. Annexe A) ou un arbre de décision est généré à partir des données précédemment calculées.

La phase de reconnaissance nécessite le calcul des signatures des symboles à traiter. Enfin, le processus peut être initié. Il s'agit d'une navigation dans le treillis permettant de valider les attributs jusqu'à ce que les attributs d'une classe se distinguent. Cette navigation est initialisée à partir de l'élément minimum du treillis où aucun attribut n'est validé et nécessite un critère de choix et un critère d'arrêt.

1.2 Travail à effectuer

Le langage de programmation choisi est le Java pour sa portabilité et sa simplicité de mise en œuvre d'interfaces graphiques. De plus, l'application sur laquelle nous nous basons est déjà écrite dans ce langage mis à part quelques bouts de code écrits en C. De ce fait, nous avions un choix à faire sur un environnement de développement entre les deux principaux que sont NetBeans [7] et Eclipse [6], tous deux gratuits et disponibles sur toutes les plateformes (Windows Linux, MacOS). Nous avons retenu le premier car, d'expérience, nous avions une préférence pour celui-ci et aussi parce qu'il nous semble plus rigoureux sur certains aspects que son concurrent notamment sur la gestion des packages, des bibliothèques et surtout par l'utilisation du programme "ant" pour la compilation des projets. Par ailleurs, chacun propose son propre module UML afin de pouvoir créer des diagrammes.

1.2.1 Intégrer le système de calcul des signatures

Cette étape consiste à adapter le code d'extraction des signatures du langage C (existant) vers le Java afin de l'intégrer au projet global. Pour l'instant, trois algorithmes de signatures sont implémentés pour calculer : la transformée de Fourier-Mellin[1][2], la transformée de Radon[3][4] et les moments de Zernike[5] d'une image (de symbole). Un quatrième type de signature est en cours de développement par Mickaël Coustaty dans le cadre de son stage de recherche de fin de deuxième année de Master. Cette signature sera basée sur les propriétés structurelles des symboles à reconnaître.

1.2.2 Remodeler l'architecture globale du logiciel prototype

Le prototype sur lequel nous nous basons est en fait l'assemblage de divers travaux effectués séparément à l'Université (projets M1, stages M2, thèses ...). De ce fait, la structure de ce prototype est très hétéroclite. En effet, en analysant le diagramme de classes correspondant à ce prototype, nous constatons par exemple un grand nombre de "compositions" présentes mais inutiles au fonctionnement du programme. Par ailleurs, certaines classes sont trop générales et de ce fait, la structure globale est à reprendre afin d'obtenir une structure cohérente et évolutive.

1.2.3 Créer une interface graphique conviviale

L'interface graphique de ce prototype a été faite dans le but de tester puis sélectionner sommairement les fonctionnalités à implémenter. De ce fait, celle-ci est peu évolutive, difficilement maintenable et peu conviviale. C'est pourquoi une refonte complète de cette interface doit être effectuée (Fig. 1.1).

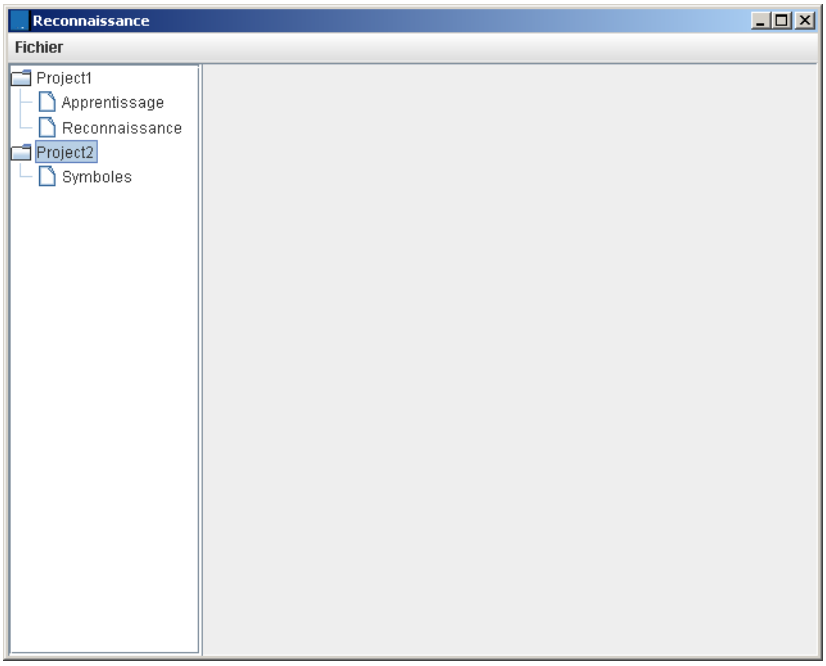


FIG. 1.1 – Première ébauche de la future IHM.

Chapitre 2

Signatures

2.1 Les bibliothèques

Avant même de commencer la traduction des algorithmes de signature d'image, nous avons dû trouver des bibliothèques pour gérer les images facilement et les nombres complexes utilisés pour le calcul des signatures qui n'étaient pas gérés par la bibliothèque « J.A.M.A. »[10] utilisée dans le prototype.

Pour la gestion des images, après une longue recherche, nous nous sommes fixés sur l'utilisation du programme « ImageJ »en tant que bibliothèque car il nous semblait être le plus complet, le plus pratique et le plus léger.

Du point de vue mathématiques, il nous a paru plus simple de trouver une bibliothèque qui gérait les matrices mais aussi les nombres complexes (à l'inverse de « J.A.M.A. »qui ne gère pas ces derniers). Nous avons décidé d'utiliser la bibliothèque « JScience »[11] qui nous paraissait très complète. Cependant, à l'utilisation, nous nous sommes rendu compte d'incohérences dans celle-ci, c'est pourquoi nous avons décidé d'utiliser la version expérimentale de « JScience »[12] beaucoup plus complète et surtout entièrement réétudiée du point de vue de l'architecture et des méthodes disponibles dans les classes.

2.2 Les signatures implémentées

2.2.1 La transformée de Fourier-Mellin

Cette signature est basée sur le calcul de la transformée de Fourier-Mellin d'une image. Le résultat de ce calcul est une certaine quantité de nombres complexes déterminée par deux paramètres P et Q. Ceux-ci correspondent respectivement à un coefficient radial et à un coefficient circulaire. Afin de pouvoir calculer ces complexes, nous avons dû utiliser les classes `Complex` et `ComplexMatrix` de « JScience Experimental ». En effet, avant de pouvoir obtenir l'invariant, un descripteur doit être calculé sur la base d'une matrice de complexes ayant les mêmes dimensions que l'image à traiter. Logiquement, nous pouvons imaginer que cette matrice est assez lourde à manipuler étant donné qu'avec « JScience », les complexes sont gérés sur 128 bits (64 bits pour chacune des deux parties, réelle et imaginaire). Dans la première version de l'algorithme, toutes les matrices (17 exactement avec P égal à 2 et Q à 3) étaient allouées simultanément. Dans la version écrite en langage C, cela ne posait pas de problème mais une fois transcrite en Java, cela saturait la mémoire

allouée par défaut au programme par la machine virtuelle. Nous avons donc dû trouver une méthode afin d'éviter d'avoir à donner des instructions spécifiques à la machine virtuelle pour pallier ce problème. La solution trouvée fut très simple en théorie mais un peu plus compliquée à mettre en pratique. En effet, il s'agissait d'allouer une matrice, de calculer le descripteur correspondant, d'en déduire l'invariant avant de libérer cette matrice pour passer à la suivante. La figure 2.1 représente le diagramme de classes de la classe `SignatureFM` faite pour calculer la signature de Fourier-Mellin sur une image. Nous pouvons y retrouver dans la classe interne `Filter` ladite matrice de complexes nommée "data".

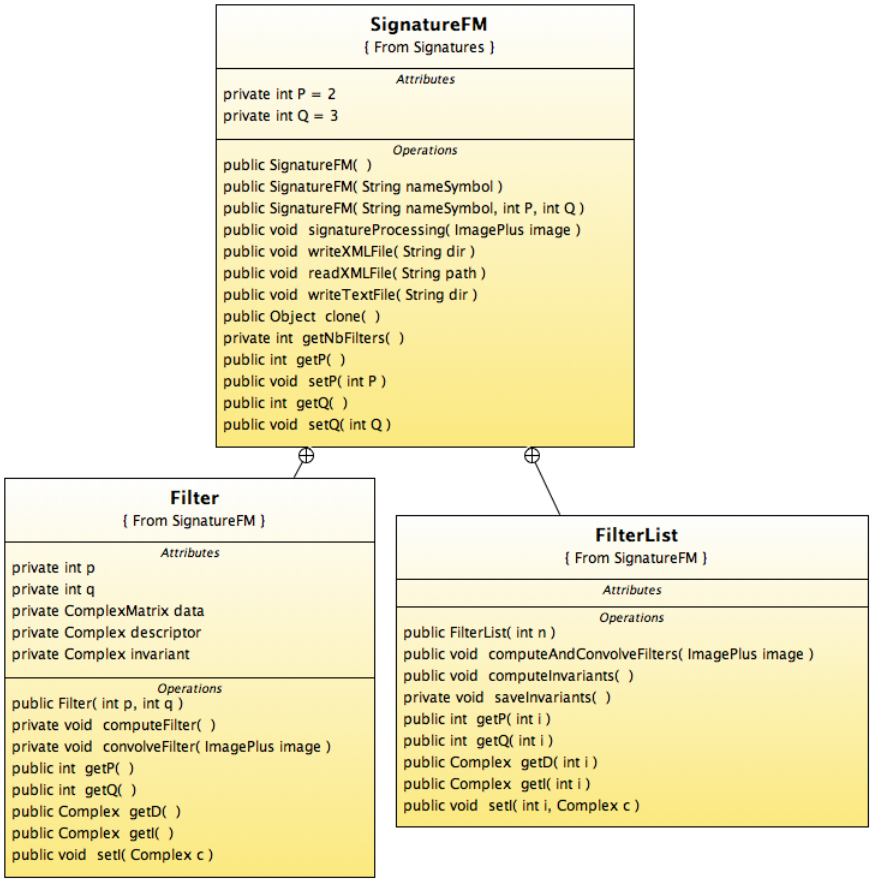


FIG. 2.1 – Diagramme de classes de `SignatureFM`.

2.2.2 La transformée de Radon

Cette signature est basée sur le calcul de la transformée de Radon. Cette classe a été réalisée par Stéphanie Guillas. Il n'y a pas eu de problèmes rencontrés pour l'adapter en Java. Une optimisation a été faite en utilisant les classes `DoubleMatrix`, `DoubleVector` et `IntegerVector` à la place de simple tableaux.

2.2.3 Les moments de Zernike

La signature du symbole est obtenue en calculant les moments de Zernike sur l'image à traiter. La version originale en C comportait de nombreuses méthodes utilisées lors des étapes de calcul dont des opérations essentielles sur les complexes. Pour adapter le code en Java, nous avons décidé d'utiliser la bibliothèque « JScience Experimental »qui gère les complexes et ses opérations (calcul du module, de l'argument, de la division entre deux nombres...). Elle permet d'alléger le code et par conséquent de le rendre plus lisible. Lors de l'implémentation de cette classe, nous avons, par souci d'exactitude, comparé les résultats obtenus en Java avec ceux obtenus en C. Les résultats étaient relativement proches, mais pas identiques. L'écart était trop grand pour qu'il s'agisse d'un problème de transtypage ou de précision après la virgule. Après s'être penché sur le problème, il s'est avéré que deux erreurs s'étaient glissées dans le code en C :

- une erreur lors du calcul de l'argument de complexe.
- une erreur lors du calcul de la division de complexe (problème d'allocation de pointeur).

2.3 Travail effectué

2.3.1 La classe abstraite `Signature`

Afin de rendre l'utilisation de ce package la plus dynamique possible, nous avons dû constituer cette classe de deux tableaux statiques contenant respectivement le nom des classes héritant de la classe abstraite et les noms usuels de ces mêmes classes. Ceci a dû être fait car en Java il n'existe aucun moyen pour obtenir de façon dynamique le nom des classes héritant d'une classe donnée. Les accesseurs nécessaires à l'accès à ces attributs ont été aussi implémentés. Lors de l'ajout d'une signature dans le futur, il sera nécessaire de mettre à jour correctement ces tableaux statiques.

Les données des signatures sont stockées dans une `TreeMap<String, Double>` car il a été défini avec nos responsables que les identifiants pourraient être des chaînes de caractères et que les valeurs seraient systématiquement des nombres réels. Comme un `TreeMap` n'accepte que des objets, nous avons du faire appel à la classe `Double` fournie par le langage Java qui encapsule le type primitif `double` correspondant aux nombres réels double précision.

Celle-ci force l'implémentation des méthodes suivantes dans les classes filles qui en héritent :

- `void signatureProcessing(ImagePlus image)`
  ⇒ pour le calcul de la signature.
- `void writeTextFile(String dir)`
  ⇒ pour l'écriture de la signature dans un fichier texte.
- `void writeXMLFile(String dir)`
  ⇒ pour l'écriture de la signature dans un fichier XML.
- `void readXMLFile(String path)`
  ⇒ pour la lecture du fichier XML modélisant la signature.
- `Object clone()`
  ⇒ pour obtenir un clone de la signature source.

Cette classe implémente de plus les méthodes :

- `void write(String dir)`
  ⇒ pour l'écriture de la signature en tant qu'objet dans un fichier.
- `static Signature read(String path)`
  ⇒ pour la lecture à partir de ce même fichier.
- `String toString()`
  ⇒ pour représenter la signature en tant que chaîne de caractères.
- `Double getValueAt(String key)`
  ⇒ pour obtenir la valeur associée à la clef dans la `TreeMap` contenant les données de la signature.

Enfin, un constructeur `protected` a été fourni afin d'initialiser les variables héritées. De ce fait, les classes filles n'ont qu'à appeler ce constructeur à l'aide de la méthode `super()` afin d'effectuer cette opération. Étant donné que cette classe est abstraite et malgré la présence de ce constructeur, celle-ci ne peut être instanciée.

Les classes héritant de la classe abstraite `Signature` (Fig. 2.2) :

- La classe `SignatureFM`.
- La classe `SignatureRadon`.
- La classe `SignatureZernike`.

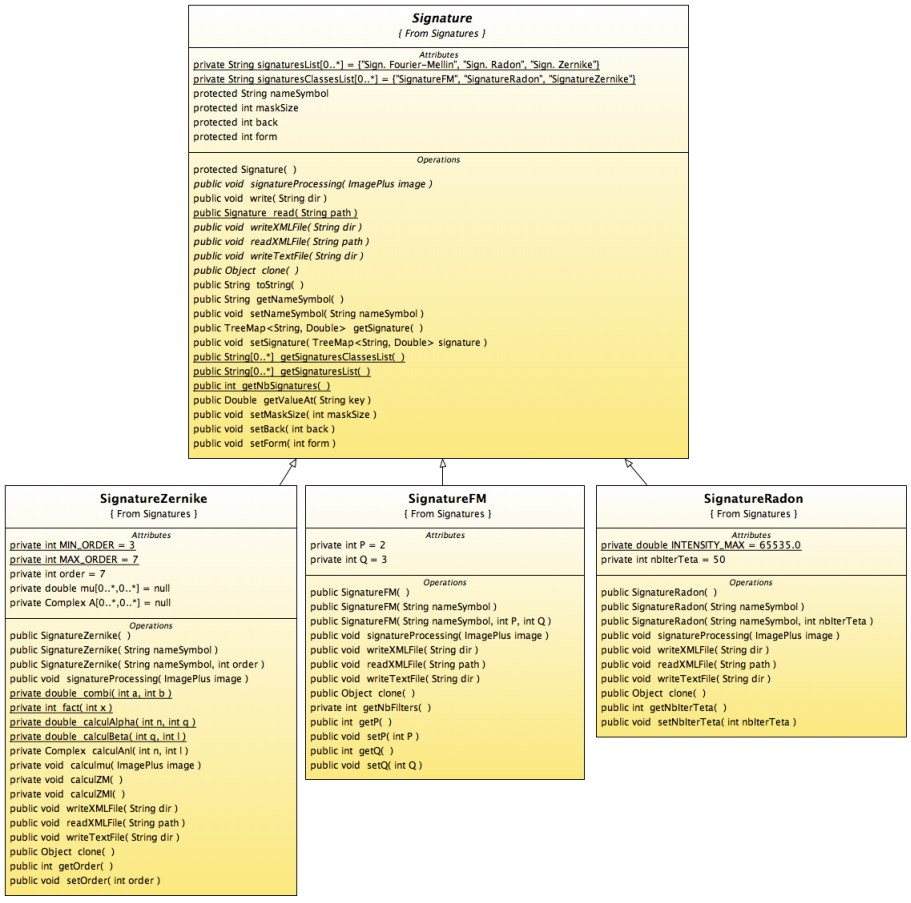


FIG. 2.2 – Diagramme de classes du package `Signatures`.

2.3.2 Fonctionnalités

Pour l'ajout dans le futur d'un nouveau type de signature, il suffira de créer une classe nommée `SignatureX` où X serait le nom de cette nouvelle signature (i.e. dans le cas d'une signature basée sur les propriétés structurelles des symboles, celui-ci pourrait être `SignatureStructurelle`). Les conditions pour ajouter une nouvelle signature seront donc :

- Hériter de la classe `Signature`.
- Implémenter toutes les méthodes abstraites de cette dernière.
- Avoir son nom simplifié et son nom usuel indiqués dans les champs statiques prévus à cet effet dans la classe `Signature` (i.e. Nom simplifié : `SignatureStructurelle`, nom usuel : "Sign. Structurelle").
- Faire partie du package `Kernel.Signatures`.
- Avoir son pendant visuel dans le package `GUI.Tree.Dialogs.SignaturesPanel` (Chapitre 4.2.2).

Toutes ces signatures répondent aux mêmes fonctionnalités :

- Enregistrement et lecture en mode texte.
- Enregistrement et lecture en XML.
- Enregistrement et lecture en tant qu'Objet.
- Utilisation de la bibliothèque optimisée « JScience Experimental ».

2.3.3 Les tests

Afin de tester ces classes correctement et facilement, nous avons créé une petite interface graphique (Fig. 2.3).

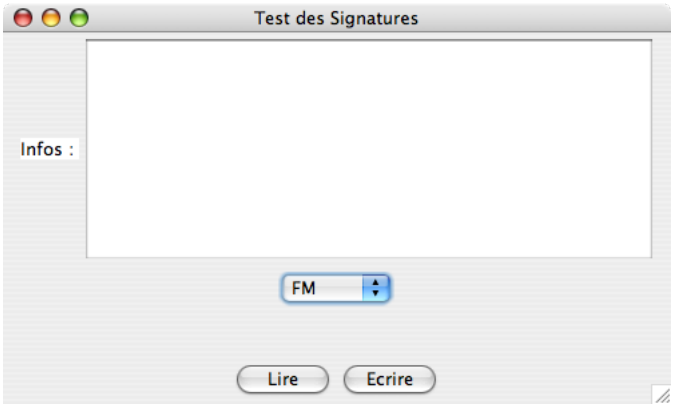


FIG. 2.3 – Fenêtre d'accueil vierge.

Elle permet de choisir le type de signature à tester (Fourier-Mellin, Radon ou Zernike) puis l'opération à effectuer sur celle-ci : calcul de la signature (Fig. 2.4) ou lecture d'une signature déjà calculée (Fig. 2.5).

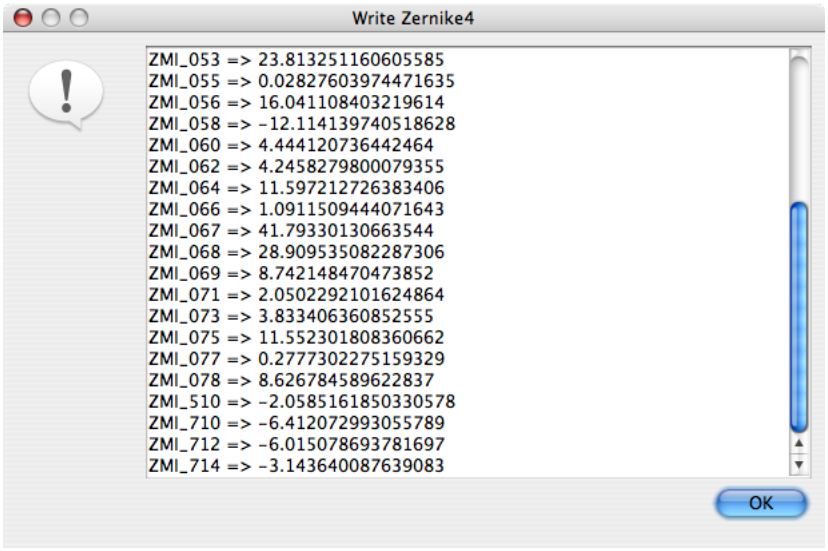


FIG. 2.4 – Résultat du calcul d'une signature Zernike.

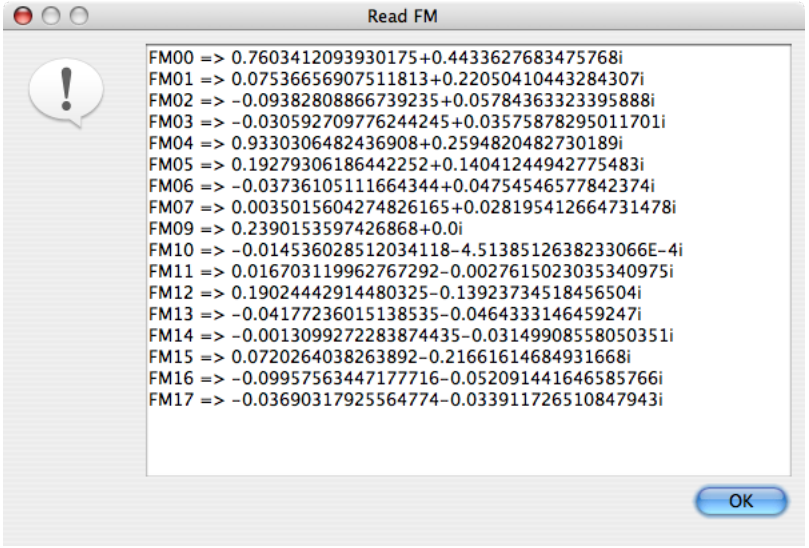


FIG. 2.5 – Lecture d'une signature FM déjà calculée.

Cette interface permet aussi de voir quasiment en temps réel l'avancement des opérations par un rendu textuel des opérations effectuées avec, le cas échéant, un message d'erreur expliquant ce qui s'est produit (Fig. 2.6).

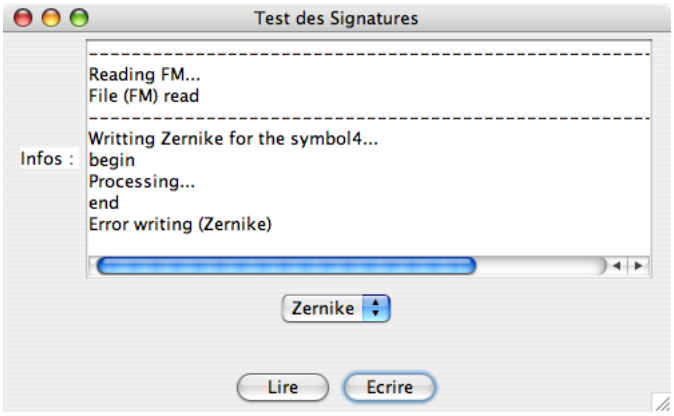


FIG. 2.6 – Fenêtre d'accueil après diverses opérations.

2.4 Problèmes généraux rencontrés

Mis à part les quelques problèmes d'implémentation spécifiques auxquels nous avons dû faire face dans les cas des signatures de Fourier-Mellin et de Zernike, nous n'avons eu qu'un seul réel problème général. Celui-ci a été découvert lors de la comparaison des résultats entre la version C et la version Java. En effet, pour la signature de Radon, les opérations de calcul du code C étaient identiques à celles du code Java, et pourtant le résultat était différent. Le problème venait en fait de la lecture des images bitmap implémentée en C qui rognait la partie basse de l'image. Le symbole (Fig. 2.7) ainsi faussé était donc différent du symbole (Fig. 2.8) lu en Java à l'aide de la bibliothèque ImageJ. Cette différence n'était pas localisée sur la signature de Radon mais nous nous sommes servis de celle-ci comme base car c'est sur celle-ci que Stéphanie Guillas a le plus travaillé. De plus, c'est aussi cette signature qui a le code le plus simple des trois donc si cela avait été un problème venant de la traduction, cela aurait été rapide à détecter.

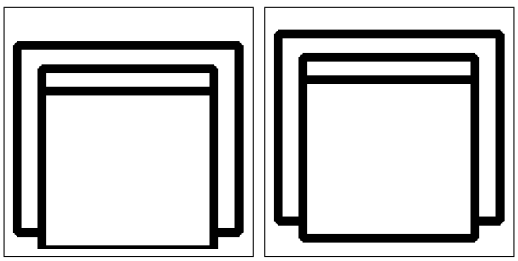


FIG. 2.7 – À gauche : Symbole lu en C.

FIG. 2.8 – À droite : Symbole lu en Java.

Chapitre 3

Architecture du programme

Avant de mettre en place la nouvelle interface graphique, nous devons nous assurer que la partie non graphique est correctement structurée. De ce fait, nous avons dû effectuer une analyse de l'existant afin de pouvoir décider de la structure à implémenter.

3.1 La version existante

3.1.1 Description

L'architecture actuelle du programme est très désordonnée. En effet, chaque classe (ou presque) a été développée lors d'un projet de Maîtrise ou de stage de DEA, il y a quelques années, de façon individuelle. Ceci fait que, pour faire fonctionner l'ensemble, des artifices ont dû être utilisés. Après analyse du code en détail, nous nous sommes rendu compte que certains étaient inutiles. Par exemple, chaque classe contient une référence vers l'instance de la classe `ControlleurTreillis`. Par ailleurs, la classe `interfaceGI` correspondant à l'ancienne interface graphique contient donc une référence vers le contrôleur mais ce dernier contient aussi une référence vers cette interface. En somme, lorsque nous avons généré le diagramme de classes UML de cette version de l'application (Fig. 3.1 et 3.2), nous avons mieux compris le travail que nous aurions à faire.

3.1.2 Les problèmes

Étant donné que nous sommes parti de rien pour l'implémentation des signatures, lorsque nous avons voulu faire fonctionner l'ensemble Signatures + Treillis nous avons eu des soucis d'implémentation. Comme nous étions sûrs du bien fondé de notre implémentation des signatures, nous nous sommes résolus à modifier profondément le code du programme principal. Par ailleurs, le programme d'origine avait été écrit selon la norme Java 1.3 ou 1.4 [9] ce qui implique un code peu lisible du fait de l'absence de typage générique dans les collections de données utilisées. De plus, un apport de la version 1.5 [8] du langage est l'utilisation de la syntaxe `for(Element e : collectionAparcourir)`, une écriture qui masque l'utilisation des itérateurs mais surtout rend le code bien plus lisible.

Les figures 3.1 et 3.2 représentent le diagramme de classes généré par NetBeans à partir du code source fourni par Stéphanie Guillas au début du projet.

FIG. 3.1 – Diagramme de classes de la version originale de l'application. (1ère partie)

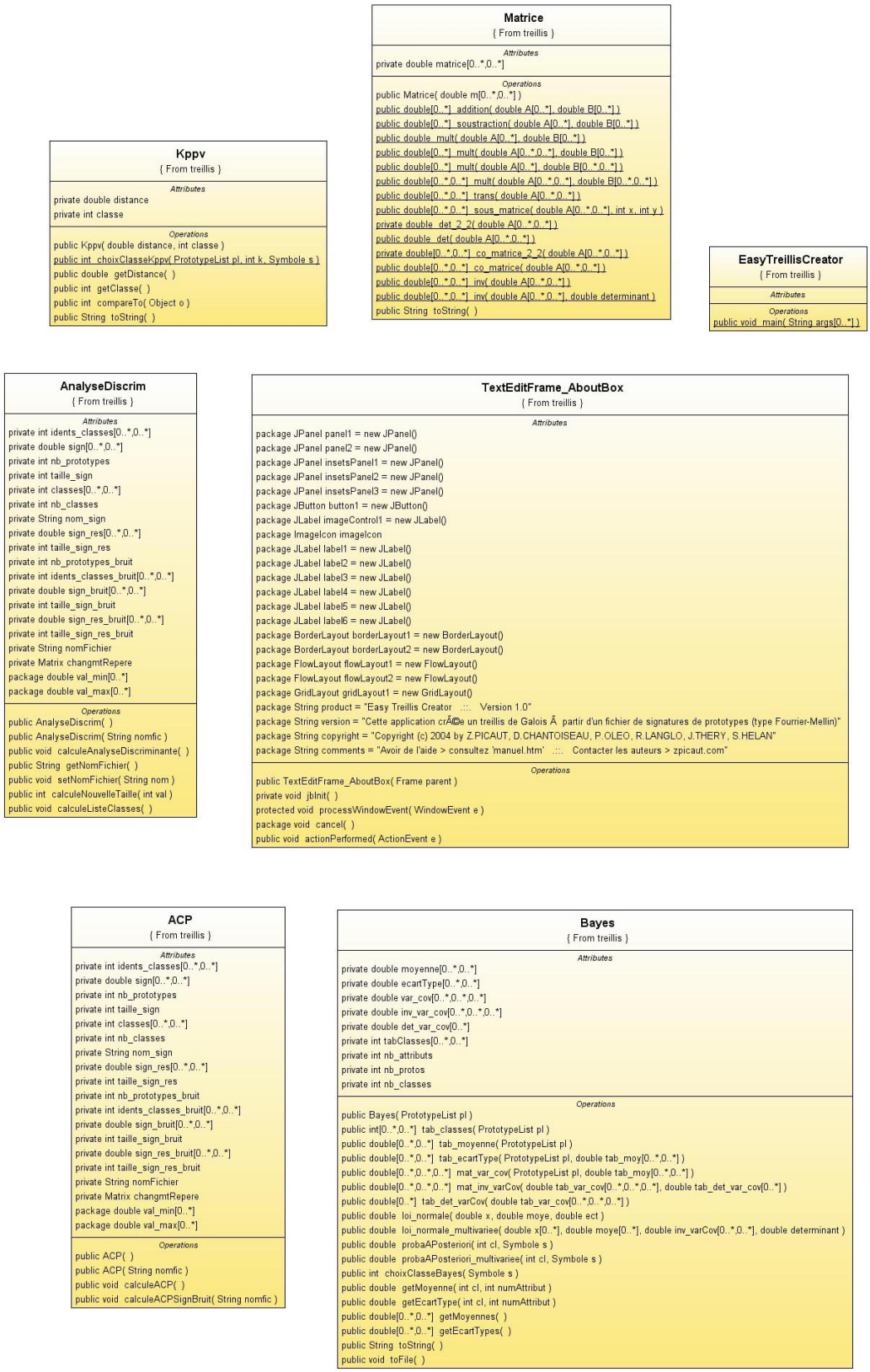


FIG. 3.2 – Diagramme de classes de la version originale de l'application. ($2^{\text{ème}}$ partie)

3.2 La version développée

Après analyse de la version originale de l'architecture du programme, nous avons remarqué qu'en utilisant la propriété d'introspection du langage Java, nous gagnerions en modularité. Cette propriété consiste à utiliser les classes `Package`, `Class` et `Constructor` afin d'essayer de charger dynamiquement un objet en connaissant son nom.

Par exemple, si nous savons à un moment donné dans une méthode qu'un objet de nom "Symbole" devrait être instanciable alors il nous suffit d'appeler la méthode statique `Class.forName()` en lui donnant en paramètre le nom de la classe ("Symbole" dans notre exemple) pour obtenir une référence sur la classe correspondante. Pour instancier un objet de ce type, il suffit alors d'appeler la méthode `newInstance()` sur cet objet de type `Class`.

Étant donné que tout ceci est dynamique, plusieurs types d'exceptions peuvent être levés comme `ClassNotFoundException`, `InstantiationException`, etc. Cette technique implique qu'il y ait un constructeur par défaut (sans paramètre) car c'est celui-ci qui est implicitement appelé lors de l'exécution de la méthode `newInstance()`. Si nous souhaitons utiliser un constructeur particulier, alors nous devons faire appel à la classe `Constructor` qui permet de scruter la classe et d'en extraire tous les constructeurs disponibles.

La classe `Package` quant à elle permet d'obtenir dynamiquement une référence sur le package contenant une certaine classe et donc d'en obtenir son nom. En effet, nous avons besoin de celui-ci car l'utilisation de la méthode `forName()` implique de fournir à cette dernière le nom qualifié de la classe pour laquelle nous demandons une référence.

3.2.1 Travail effectué

Nous avons décidé d'intégrer le calcul des signatures en premier afin de partir sur des bases sûres. De ce fait, la première chose à faire fut de transcoder en Java les différents programmes écrits en C (Chapitre 2.3). Pour rendre cette partie modulaire, nous avons utilisé l'introspection des classes de façon à ce qu'il y ait le moins de choses à modifier pour ajouter/supprimer une signature. Une fois ceci fait, nous avons pu essayer de reprendre le code déjà existant pour le reste des opérations non graphiques.

L'étape suivante consistait en la normalisation des valeurs des signatures précédemment calculées de façon à restreindre l'étendue des valeurs possibles sur $[-1, +1]$. Cette normalisation se fait, soit selon une loi linéaire, soit selon une loi normale. Ainsi, nous avons décidé de créer une seule et unique classe `Normalisation` ne comprenant que des méthodes statiques dont une seule est publique. En effet, cela permet de n'appeler que celle-ci qui, elle, s'occupera d'appeler la méthode adéquate en fonction du type de normalisation précisé en paramètre.

Une fois cette normalisation effectuée, nous avons pu alors générer la table discrétisée. Cette table est représentée par une liste de symboles contenant les intervalles discrétisés. Pour cette étape, nous avons entièrement réécrit les méthodes de calcul de cette table du fait de notre nouvelle structure de signature qui n'existait pas dans la version précédente.

Comme convenu avec Karell Bertet, nous nous sommes ensuite focalisés sur la génération du treillis de Galois, aussi appelé treillis des concepts. Pour ce faire, nous nous sommes basés sur le code écrit par nos aînés. Après une dure adaptation de ce code (du

fait des divers problèmes cités précédemment), nous avons tenté de générer un treillis correspondant à une des expériences qu'avait menées Stéphanie Guillas dans son travail. À ce moment là, nous avons constaté le plus gros des problèmes auxquels nous n'avions pas pensé. Avec nos structures de données optimisées par rapport à l'ancienne version, les méthodes de génération du treillis ne fonctionnent plus. De plus, étant donné que le code en question était très peu, voire pas du tout, commenté, nous n'avons pas réussi à comprendre quel était l'algorithme employé. Après avoir contacté Karell Bertet, nous nous sommes retrouvés pour qu'elle nous explique cet algorithme en nous faisant un déroulement sur un exemple simplifié. Nous avons pu ainsi le réimplémenter avec succès en Java 1.5 en utilisant nos structures.

Enfin, nous avons découpé cette nouvelle version en plusieurs packages tant au niveau de l'interface graphique qu'au niveau de du noyau (Fig. 3.3). Ceux-ci ont été pensés de façon à rendre l'ensemble le plus cohérent et le plus modulaire possible. Le package `Kernel` est donc découpé en plusieurs packages que sont `Project` pour la gestion d'un projet, `Structures` pour la gestion des différentes structures de données utilisées et `Tools` pour les outils divers ne trouvant pas leur place dans les autres packages. Le package `Structures` est lui même composé des packages `Signatures`, `Graphs` et `Table`. Pour l'interface graphique, nous avons essayé de faire de même avec les packages `Dialogs`, `Listeners`, `Models`, `TabbedPane`, `Tree`, `ViewPanels` et enfin `Tools`. Nous avons essayé d'utiliser des noms les plus explicites possible de façon à ce qu'on puisse s'y retrouver assez rapidement.

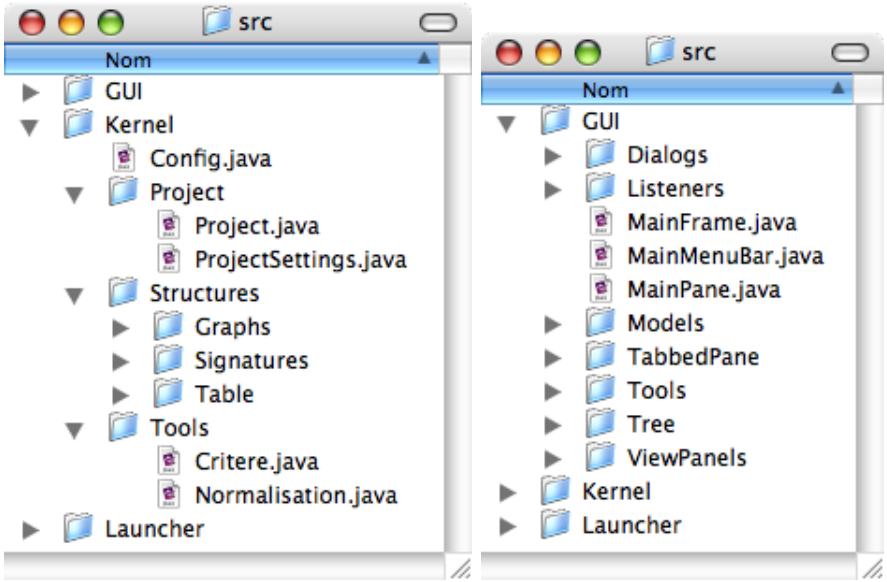


FIG. 3.3 – Arborescences des packages Kernel et GUI.

3.2.2 Structures utilisées

Afin de générer la table discrétisée, dans un premier temps, nous avons été amené à refaire toute une structure d'objets en nous inspirant de la version existante. Nous avons finalement tout refait *a novo* car nous avons décelé de nombreuses incohérences dans le

code existant. De plus, les types génériques n'étaient pas exploités alors que ce cas s'y prêtait particulièrement bien.

Nous avons fusionnés les anciennes classes `Prototype` et `Symbole` en une unique classe `Symbole` et de la même façon pour `PrototypeList` et `SymboleList` en `SymboleList`. Un `Symbole` hérite d'un `TreeSet<Cell>` et contient un identifiant, sa classe d'appartenance et une instance de `Signature`. Une `SymboleList` hérite d'une `ArrayList<Symbole>` et contient une `IntervalleList`.

Les classes `Intervalle` et `IntervalleList` existent toujours mais ont été aussi refaites complètement pour les mêmes raisons. Un `Intervalle` hérite d'un `TreeSet<Cell>` et contient un identifiant, la clef correspondant à la valeur associée se trouvant dans la signature, le centre de gravité de l'intervalle, le point de coupe ainsi que la valeur du critère utilisé pour la discrétisation. Une `IntervalleList` hérite d'un `TreeSet<Intervalle>`.

Enfin, la classe `PrototypeReduit` qui était interne à l'ancienne classe `Intervalle` est devenue une classe à part entière renommée `Cell`. Celle-ci est primordiale car elle permet de passer d'un symbole à ses intervalles associés et inversement. Elle représente une "cellule" de la table à discrétiser ou encore un couple de la relation entre `Intervalle` et `Symbole`. Un `Cell` contient donc une référence vers un `Symbole` et une vers un `Intervalle`.

Nous avons souvent fait appel à la structure `TreeSet`, afin de gérer facilement l'unicité des valeurs au sein des structures. De plus, cette structure permet de conserver les informations stockées dans un certain ordre défini à l'aide de la méthode surchargée `compareTo`.

La figure 3.4 représente la modélisation d'une table discrétisée telle que nous l'avons programmée en Java.

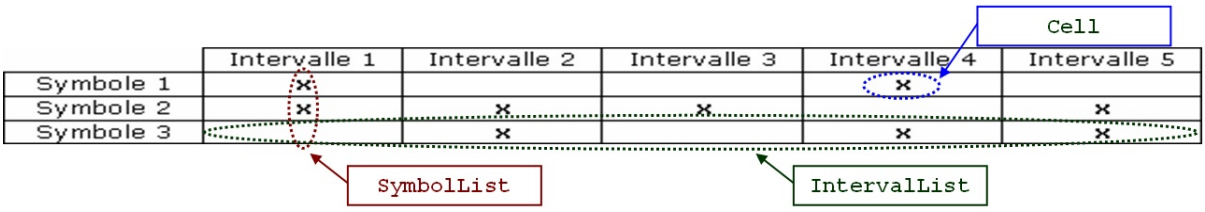


FIG. 3.4 – Représentation d'une table discrétisée.

Chapitre 4

L'Interface Graphique Utilisateur

4.1 La version existante

La version originale de l'interface graphique est très sommaire. En effet, celle-ci n'est pas praticable sans explications. Nous avons tout de même essayé de l'utiliser afin de comprendre les différentes étapes du processus de reconnaissance et nous avons subi beaucoup d'échecs. Toute l'activité se passe sur la même fenêtre. Le seul retour que nous avons s'arrête à un log nous informant de la commande que nous venons d'exécuter. Par ailleurs, en voulant générer le treillis, nous avons constaté avec quelle facilité le fichier contenant les signatures pouvait être détruit.

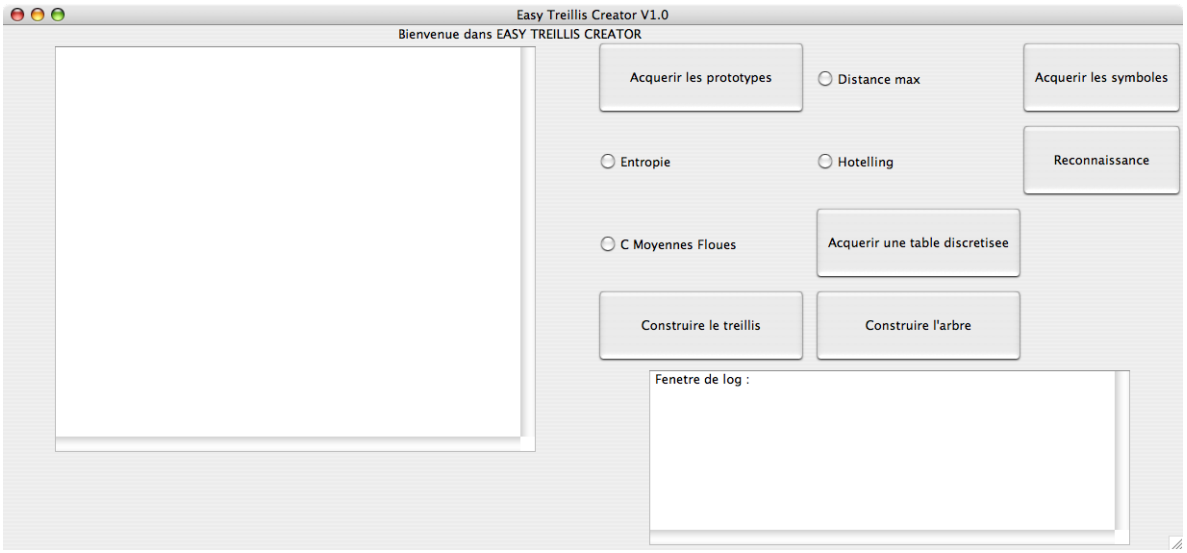


FIG. 4.1 – Interface graphique originale.

4.2 La version développée

4.2.1 Description et caractéristiques

La nouvelle interface se doit donc d'être conviviale, compréhensible et modulaire. De plus, un retour autre qu'un "simple" Log est obligatoire. Après analyse, voici une liste non-exhaustive des retours visuels et fonctionnalités qui seraient appréciés :

- Affichage des symboles.
- Affichage des signatures.
- Comparaison aisée des symboles et de leur signature associée.
- Affichage du treillis.
- Affichage de l'arbre de décision.
- Configuration des paramètres de calcul des signatures.
- Configuration des paramètres des méthodes de discrétisation.

4.2.2 Travail effectué

Pour garder une cohérence maximum avec le noyau du programme, nous avons ici aussi utilisé l'introspection des classes pour rendre dynamique l'ajout et la suppression de signatures. En effet, pour la partie graphique, il suffit de créer une nouvelle classe héritant de la classe `PanelSignature` qui permet de gérer l'affichage des paramètres de calcul de la signature. S'il n'y en a pas, un panel devra tout de même être fait, quitte à ce qu'il n'y ait qu'un message indiquant l'absence de réglage pour celle-ci. En effet, l'instanciation se base sur le contenu d'un champ statique se trouvant dans la classe `Signature`.

Afin de pouvoir gérer plusieurs expériences en parallèle, nous avons mis en place la notion de "Projet". Un projet est partagé en deux étapes distinctes que sont l'apprentissage et la reconnaissance. Lors de la création d'un nouveau projet, un répertoire au nom du projet est automatiquement créé à l'emplacement indiqué. Ce répertoire contient des sous-répertoires pour chacune des étapes précédemment citées ainsi qu'un fichier au format XML contenant les propriétés du-dit projet (paramètres d'apprentissage et de reconnaissance). Dans chacun des sous-répertoires, un dernier répertoire "Symboles" est créé afin d'accueillir les images à traiter pour cette expérience. Nous obtenons donc à la fin de la création d'un nouveau projet une arborescence décrite figure 4.2.

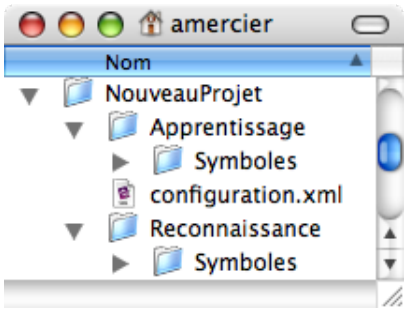


FIG. 4.2 – Arborescence d'un projet nouvellement créé.

Une fois ce projet créé, nous pouvons ajouter des images à traiter dans la liste de Symboles de l'apprentissage. Lors de l'ajout ou de la suppression d'un symbole dans la partie apprentissage, l'application ne permet que de (re)générer les signatures associées afin de conserver un état stable du projet. Une fois cette étape effectuée, la génération des signatures devient possible au travers des différents menus. À l'appel de cette fonction, une fenêtre (Fig. 4.3) apparaît vous invitant à sélectionner les types de signatures que vous souhaitez générer sur vos symboles. En effet, nous avons décidé de ne pas générer automatiquement tous les types de signature car dans le cas de la signature de Fourier-Mellin et malgré les optimisations faites, celle-ci est très longue à calculer. Dans cette même fenêtre, un bouton permet d'ouvrir une autre fenêtre (Fig. 4.4) nous autorisant à modifier les paramètres des signatures avant de lancer le calcul.

FIG. 4.3 – À gauche : Fenêtre de choix des signatures à générer.

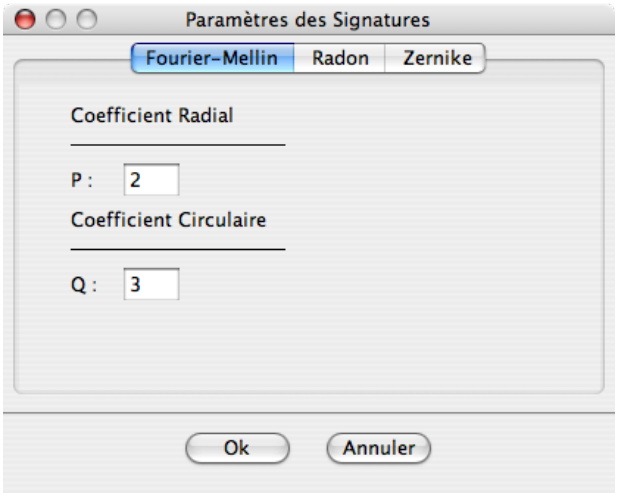


FIG. 4.4 – À droite : Fenêtre de configuration des signatures.

Dès lors que les signatures ont été calculées, la génération de la table de discrétisation est disponible dans les différents menus. Lors de l'appel à celle-ci, une fenêtre vous invite à choisir la signature précédemment générée à utiliser pour créer la table. La chronologie de l'utilisation de notre version de ce logiciel est décrite dans la figure 4.5. Cependant, par manque de temps, la partie "Reconnaissance" n'a pas pu être programmée complètement.

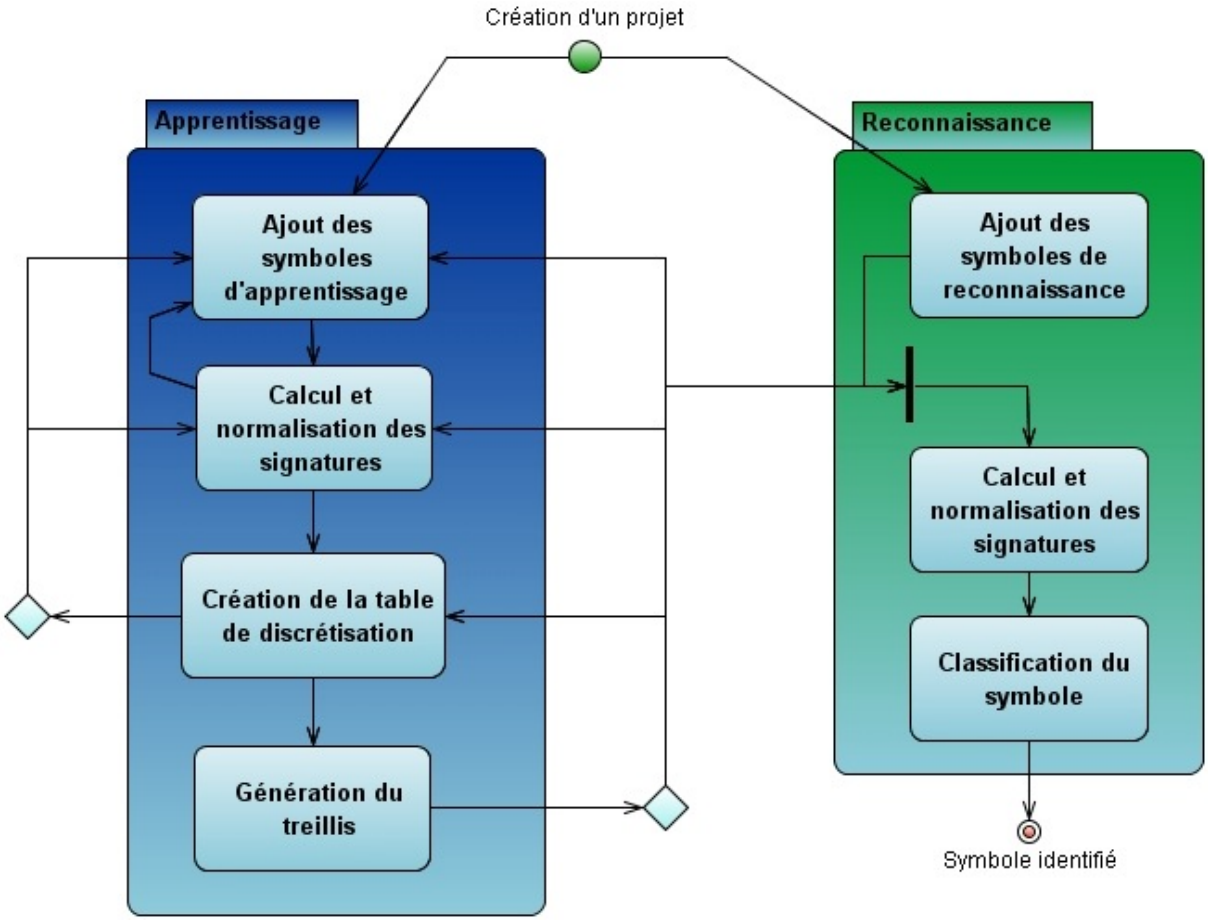


FIG. 4.5 – Diagramme d'états - transitions de la nouvelle application.

4.2.3 Aperçu

Voici une liste non-exhaustive des fonctionnalités que nous avons implémentées dans la nouvelle version du programme :

1. – Utilisation d'un arbre afin d'afficher les projets ouverts et leurs documents associés.
   – Utilisation du bouton droit de la souris afin de lancer les traitements sur les éléments d'un projet dans l'arbre.
   – Mise en place de la notion de projet principal afin d'effectuer toutes les manipulations via la barre de menu.

2. – Utilisation d'onglets afin de gérer l'ouverture de plusieurs documents en même temps.
   – Affichage des symboles.
   – Affichage des signatures avec le symbole associé en miniature et les paramètres utilisés lors du calcul.
   – Affichage du treillis.
3. – Fenêtre de configuration des préférences du programme.
   – Fenêtre de configuration des paramètres de calcul des signatures.
   – Fenêtre de configuration des paramètres des méthodes de discrétisation.
4. – Mise en place de la notion de "préférences" pour la configuration du programme.
   – Gestion des caractéristiques propres aux différents systèmes d'exploitation disponibles (Windows, Linux, MacOS X).

La figure 4.6 présente un aperçu de notre nouvelle interface graphique avec une grande partie des fonctionnalités mises en place. Cette fenêtre est décomposée en trois grandes parties : la barre de menus, l'arborescence des projets sur la gauche et les onglets de visualisations sur la droite.

L'arbre (1) représentant l'arborescence sur la droite est constitué de zéro à plusieurs racines. Chacune d'entre elle représente un projet. Dans chacun de ceux-ci, nous retrouvons obligatoirement les deux sous-parties "Apprentissage" et "Reconnaissance". Celles-ci sont constituées d'au moins une partie "Symboles" qui contiendra les symboles propres à chacune des deux phases. De plus, la partie "Apprentissage" peut aussi contenir des parties liées aux signatures calculées ainsi que des tables discrétisées ou encore des treillis. Dans la partie "Reconnaissance", nous devrions retrouver aussi des sous-parties pour les signatures calculées.

La zone de visualisation (2) est vide par défaut. Celle-ci se remplit dès lors qu'un double clic est effectué sur un nœud de l'arbre. En fonction du nœud cliqué, un onglet apparaîtra avec les informations associées à ce nœud. Dans le cas d'une image de symbole, celle-ci sera affichée dans un onglet qui lui sera propre. S'il s'agit d'une signature, celle-ci sera affichée dans une zone de texte avec au-dessus les paramètres ayant servi au calcul ainsi que l'image du symbole qui lui est rattachée. Dans le cas d'une table discrétisée, celle-ci sera affichée dans un onglet à l'aide d'un tableau digne d'un tableur.

De plus, toute une série de fenêtres (3) ont été créées afin de renseigner différentes informations telles que les paramètres de calcul ou encore les répertoires par défaut et de dotty pour la génération des graphes.

Une gestion d'ordre plus globale a été mise en place pour sauvegarder les réglages du logiciel de façon à ne pas avoir à les refaire à chaque lancement de celui-ci. Par ailleurs, une gestion automatique des particularités des différentes plateformes a été prise en compte. Par exemple, la barre des menus globale propre à MacOS X est utilisée de façon automatique (4), ou encore les chemins d'accès aux fichiers ou les séparateurs dans ces même chemins.

FIG. 4.6 – Nouvelle version de l'interface graphique.

Conclusion

Nous pouvons considérer l'essentiel de nos objectifs atteints. En effet, l'interface graphique est faite en grande partie et celle-ci est modulaire donc les détails qui ne sont pas encore gérés ne seront pas difficiles à rajouter. De plus, avec l'intégration du calcul des signatures, nous obtenons un logiciel entièrement autonome et portable étant donné que celui-ci est programmé en Java et n'utilise pas de bibliothèques comportant du code natif (dépendant de la plateforme). Par ailleurs, nous avons reprogrammé entièrement le noyau même du logiciel qui est lui aussi modulaire, optimisé et utilise énormément la généricité offerte par le langage dans sa version 1.5. De plus, il semblerait que cette nouvelle version du noyau soit bien plus efficace que l'ancienne autant en terme de temps de calcul qu'en complexité algorithmique.

Malheureusement, par manque de temps, nous n'avons pas pu finir ce que nous avions prévu. Presque toute la partie "Reconnaissance" reste à programmer. Celle-ci de devrait pas être trop difficile à implémenter car une grosse partie du calcul est déjà programmé. Il reste principalement à coder le parcours du treillis. Pour les autres fonctionnalités non présentent dans cette nouvelle version, cela ne devrait pas non plus être trop complexe étant donné la modularité de notre produit.

Ces fonctionnalités manquantes sont :

- Utilisation en lieu et place du treillis :
  - Des K plus proches voisins.
  - De la méthode de Bayes.
  - D'un arbre de décision.
- Utilisation d'une Loi Linéaire pour la normalisation.
- Utilisation d'autres Critères de coupe comme :
  - L'entropie.
  - Hotelling.

De plus, après nos tests, nous avons constaté que l'affichage du treillis posait des problèmes de rafraîchissement à partir d'un nombre trop important de nœuds. Pour cela, l'affichage interactif devra être désactivé au bénéfice de l'affichage d'une image générée par l'appel à la commande "dot".

Enfin, lors de ce projet, nous avons beaucoup appris tant au niveau analyse que développement Java. En effet, lors de nos réunions avec nos responsables, nous avons dû interpréter leurs explications de façon à les implémenter de la meilleure manière possible en fonction de notre travail déjà effectué. Par ailleurs, nous devions aussi expliquer clairement nos méthodes de programmation lors de nos séances de travail en commun avec Stéphanie Guillas notamment.

Annexes

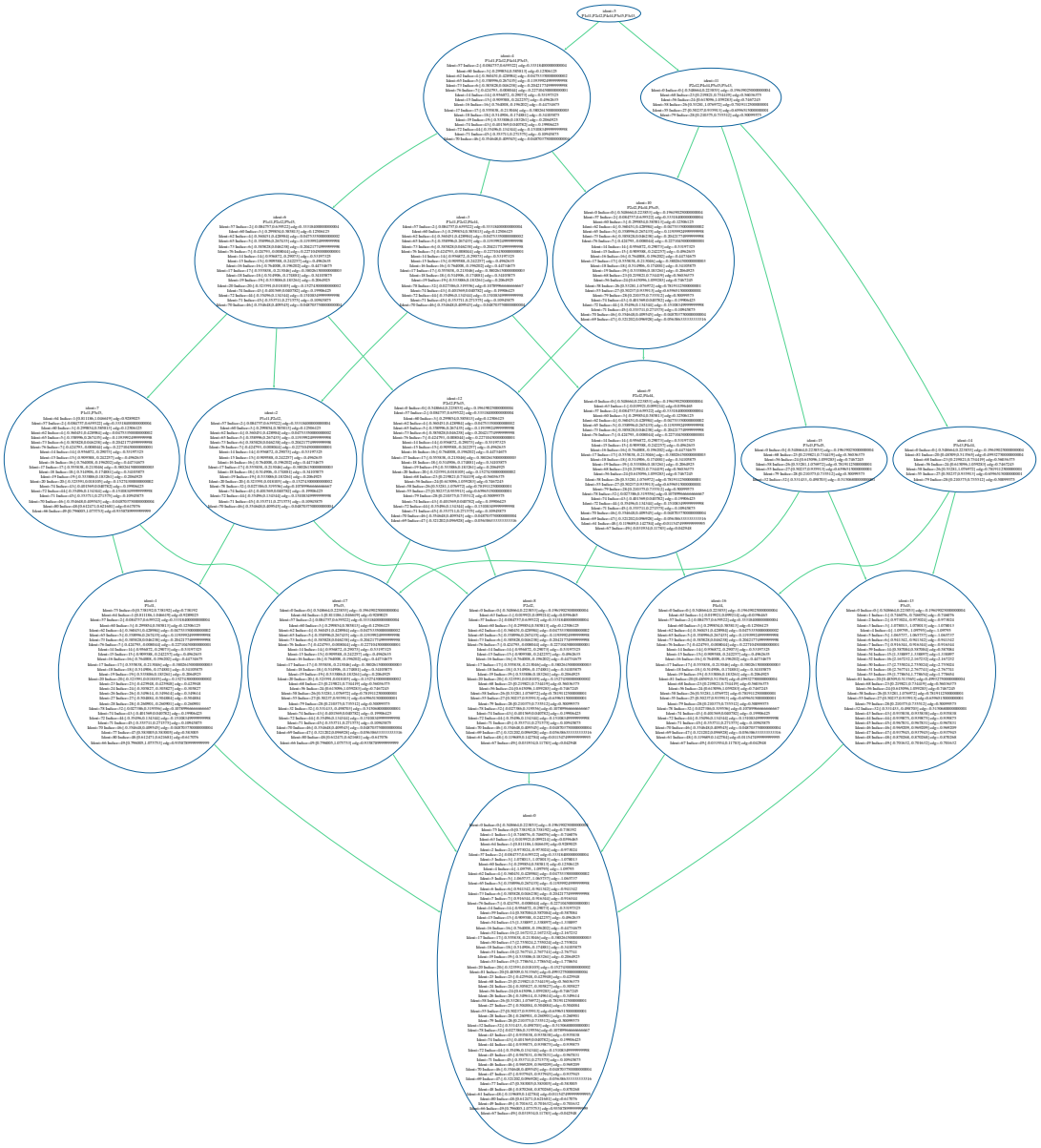
A - Exemple de Treillis de Galois.

B - Diagramme de classes de la nouvelle version de l'application (Partie 1).

C - Diagramme de classes de la nouvelle version de l'application (Partie 2).

D - Poster Stéphanie Guillas, Doctoriales 2005

E - Poster Stéphanie Guillas, Colloque des doctorants $2^{\text{ème}}$ année, 2006



Exemple de treillis de Galois (ou treillis des concepts) utilisé pour effectuer la reconnaissance de symboles.

Mettre ici le diagramme de classes général.

Mettre ici le diagramme de classes général.

Stéphanie GUILLAS

Doctorante 1ère année
Université de La Rochelle
Encadrée par K. Bertet et J.M. Ogier

Laboratoire L3I
Informatique Image Interaction

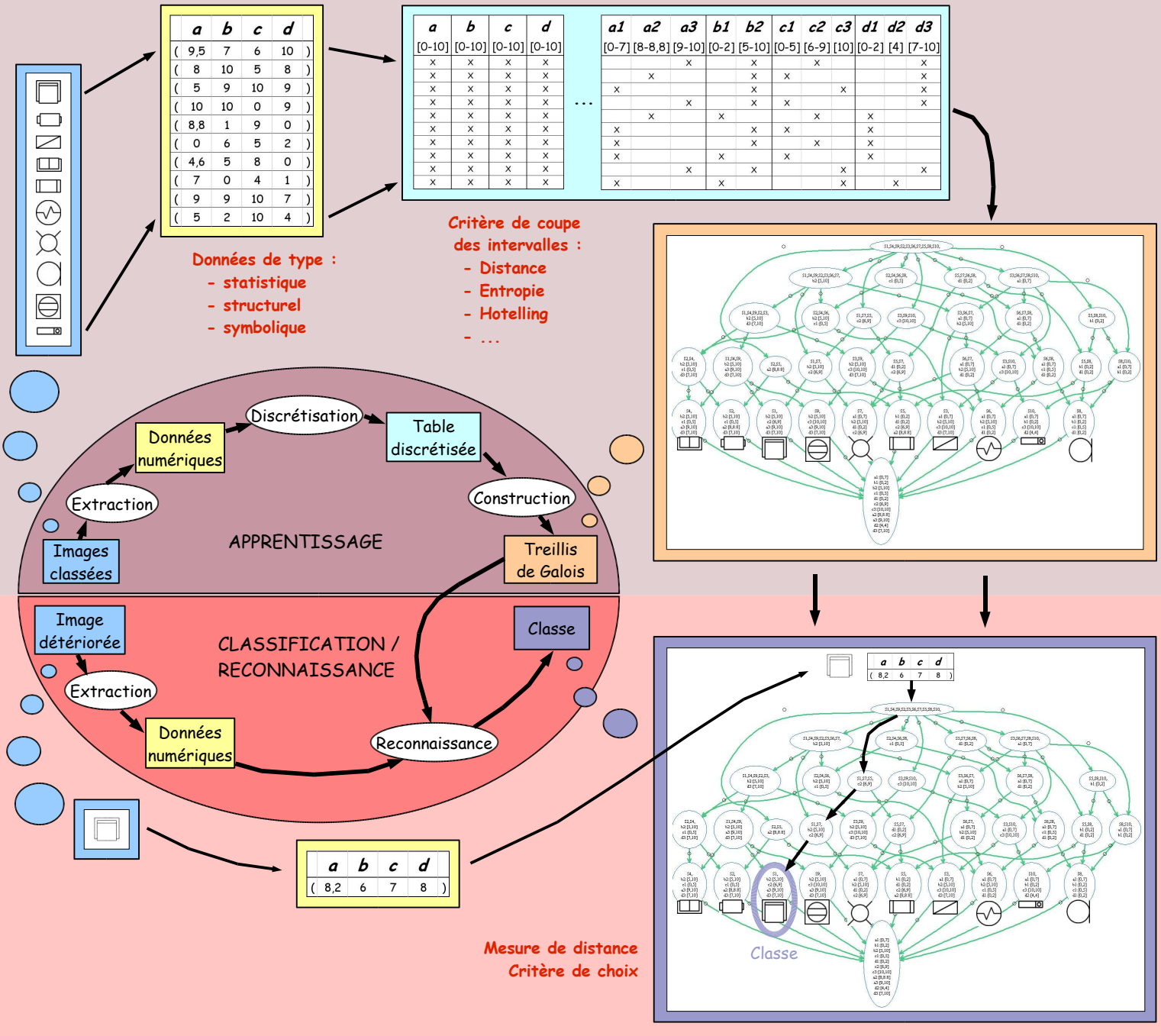
Thèse financée par le
Conseil Général
de la Charente-Maritime

Reconnaissance d'objets graphiques détériorés : approche basée sur un treillis de Galois

La reconnaissance des formes est un domaine de recherche ayant pour objectif l'automatisation des tâches. Les objets que l'on sait reconnaître peuvent être organisés par groupes selon leurs similarités dans des structures particulières (bases de données). Il est alors possible par la suite d'effectuer des recherches rapides et pertinentes.

Le treillis de Galois est un graphe particulier qui est ici utilisé comme espace de navigation pour la reconnaissance. Dans notre cas, les objets à reconnaître sont des images de symboles qui peuvent subir des transformations de type : rotation, changement d'échelle et des détériorations : taches, déformations, effacement, ...

Le schéma ci-dessous présente l'ensemble des étapes d'un processus de reconnaissance générique qui se décompose en deux phases principales : apprentissage et classification, ainsi que les éléments paramétrables permettant une adaptation aux cas spécifiques.



Les éléments paramétrables sont identifiés, il faut maintenant réussir à les adapter à notre problématique.

Stéphanie GUILLAS

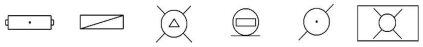
Doctorante 2ème année
Université de La Rochelle
Encadrée par K. Bertet et J.M. Ogier

Laboratoire L3I
Informatique Image Interaction

Thèse financée par le Conseil
Général de la Charente-Maritime

Comment reconnaître des symboles détériorés ?

Le treillis de Galois est un graphe particulier qui est ici utilisé comme espace de navigation pour la reconnaissance. Dans notre cas, les objets à reconnaître sont des images de symboles qui peuvent subir des transformations de type : rotation, changement d'échelle et des détériorations : taches, déformations, effacement...

PROBLEME : Connaissant les symboles  on cherche à savoir comment reconnaître le symbole  qui est effacé ou encore le symbole  qui est taché.

Le schéma ci-dessous présente l'ensemble des étapes d'un processus de reconnaissance à l'aide d'un treillis de Galois composé de deux phases principales :

APPRENTISSAGE et RECONNAISSANCE

Les 6 symboles connus

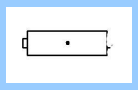
On recherche les éléments qui composent les symboles pour construire la table ci-dessous.

Éléments constituant les symboles									
Symbole 1 	X	X	X						
Symbole 2 	X			X					
Symbole 3 					X	X	X	X	
Symbole 4 	X				X				X
Symbole 5 			X		X	X			
Symbole 6 	X				X	X	X		

La table permet de construire le graphe appelé treillis de Galois.

2 symboles détériorés inconnus que l'on étudie successivement.

Exemple 1

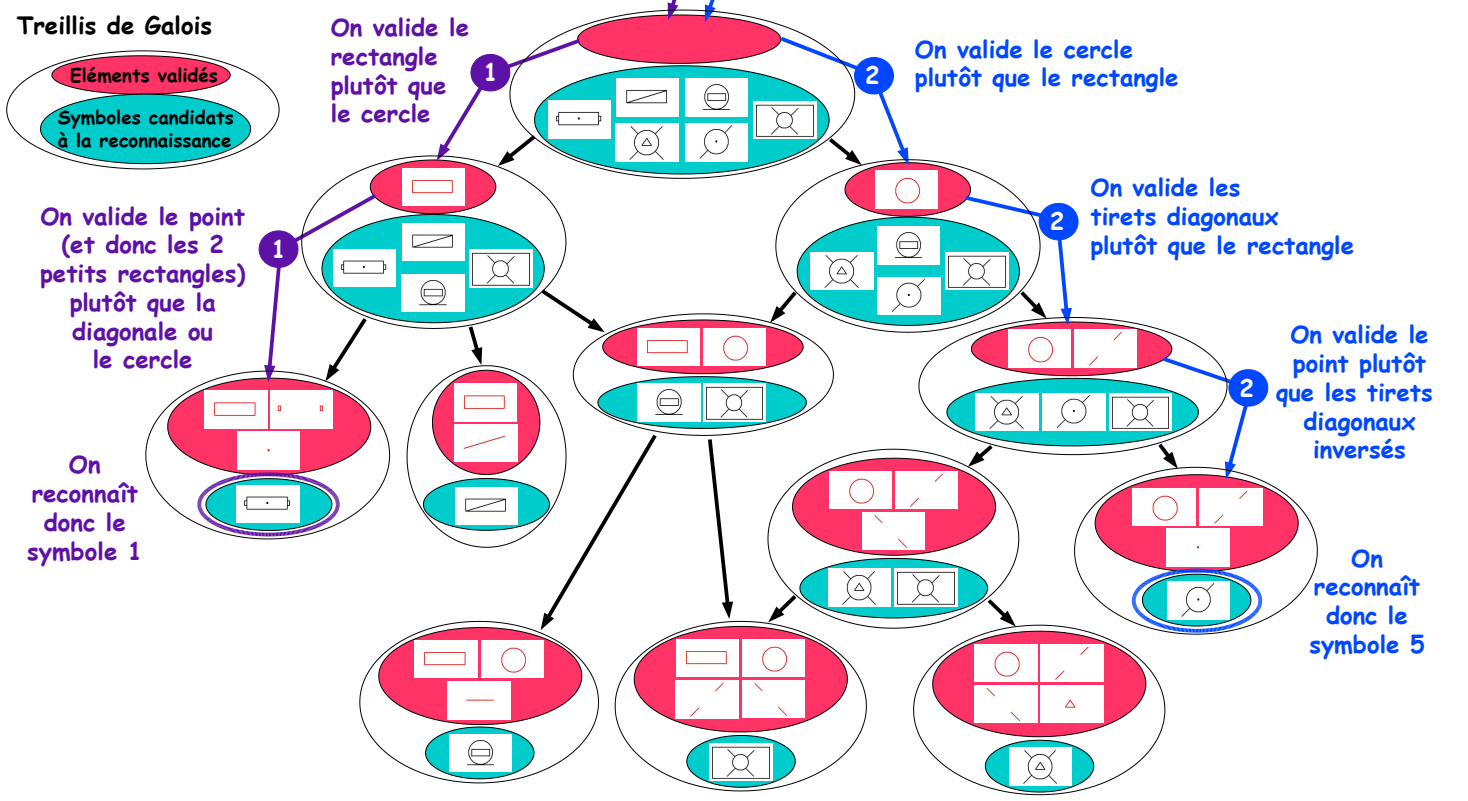


Symbole effacé

Exemple 2

Symbole taché

On recherche quels éléments composent le symbole détérioré.



Bibliographie

[1] Adam, S., Ogier, J.M., Cariou, C., Mullot, R., Gardes, J. and Lecourtier, Y. « Utilisation de la transformée de Fourier-Mellin pour la reconnaissance de forme multi-orienté et multi-échelle : application à l'analyse automatique de documents techniques » *Traitement du Signal*, 18(1) :17–33, 2001

[2] Derrode, S., Daoudi, M. and Ghorbel, F. « Invariant content-based image retrieval using a complete set of Fourier-Mellin descriptors » *Int. Conf. on Multimedia Computing and Systems (ICMCS'99)*, pages 877–881, June 1999

[3] J. Radon « Über die Bestimmung von Funktionen durch ihre Integralwerte längs gewisser Mannigfaltigkeiten » *Berichte Saechsische Akademie der Wissenschaften*, pages 262–277, 1917

[4] Tabbone, S. and Wendling, L. « Recherche d'images par le contenu à l'aide de la transformée de Radon » *Technique et Science Informatiques*, 2003

[5] Teague, M. « Image analysis via the general theory of moments » *Journal of Optical Society of America (JOSA)*, 70 :920–930, 2003

[6] Eclipse Foundation « Eclipse - an open development platform » http ://www.eclipse.org

[7] Sun Microsystems Inc. « NetBeans IDE 5.5 » http ://www.netbeans.org

[8] Sun Microsystems Inc. « Java 2 Platform Standard Edition (J2SE) 5.0 » http ://java.sun.com/j2se/1.5.0/docs/relnotes/features.html

[9] Sun Microsystems Inc. « Java 2 Platform Standard Edition version 1.4.2 » http ://java.sun.com/j2se/1.4.2/datasheet.1_4.html

[10] J.A.M.A. http ://math.nist.gov/javanumerics/jama

[11] JScience http ://jscience.org

[12] JScience Experimental http ://jade.dev.java.net