Vision Group

Computer Science Department
University of Geneva, Switzerland.

Mémoire de diplôme

UTILISATION D'UN PROTOCOLE DE CO UNICATION ET D'UNE STRUCTURE DE IC IER IN ERS POUR LA REC ERC E DANS UNE ASE DE DONNE S D'I A ES

Jilali Raki
Responsable le docteur D. McG. Squire
Assistants W. Müller et H. Müller
Sous la direction du professeur Thierry Pun

Université de Genève
Centre Universitaire d'Informatique

avril 1999

TABLE DES MATIÈRES

PRÉFACE

Ce document décrit le procédé et la manière utilisés pour archiver des images dans une base de données et rechercher celles qui sont les plus similaires à une requête d'images. La technique d'archivage des images est inspirée de celle utilisée pour archiver et rechercher du texte. Les points essentiels des chapitres traités dans le manuel sont:

- Un survol du système *Viper*[1], qui est le nom de baptême donné au système de recherche d'images à réaliser. Dans ce chapitre, la description des caractéristiques de couleur et de texture utilisées dans le système et la manière de calculer la similarité entre une image d'une requête et les images de la base données sont abordées.

- La spécification d'un protocole de communication pour l'échange de messages entre la base de données et une interface utilisateur graphique. Différents types de messages sont décrits et la grammaire du protocole de communication est spécifiée.

- L'introduction de la grammaire avec les outils de compilation ***JavaCup*** et ***JLex*** pour le langage ***Java*** et avec les outils ***Yacc*** et ***Lex*** pour le langage ***C++***. La manière de générer les analyseurs syntaxiques et lexicaux correspondants à la grammaire avec ces outils est décrite. Du fait que *JavaCup* et *Jlex* sont moins connus que *Lex* et *Yacc*, une explication plus détaillée de ces outils est donnée dans ce chapitre.

- La manière de construire l'interface graphique de l'utilisateur et une description de ses différentes fonctionnalités. On explique dans ce chapitre la manière d'implanter l'interface afin de communiquer avec un serveur lié à la base de données et d'analyser les messages du protocole de communication.

- L'archivage des images dans une structure de données appelée "***Inverted File***" ou fichier inversé. Cette structure est décrite dans ce chapitre ainsi que la manière d'implanter le serveur de la base de données afin de communiquer avec l'interface et d'analyser les messages du protocole de communication.

Après ces chapitres, des résultats de requêtes de recherche d'images similaires sont présentés afin de les comparer aux résultats d'autres systèmes.

[1] *Visual Information Processing for Enhanced Retrieval*

1 Description du projet du diplôme

Dans le cadre d'un projet du Fond National Suisse de la Recherche Scientifique, nous souhaitons créer un système qui permette de stocker un grand nombre d'images, et de rechercher et extraire automatiquement celles qui correspondent le mieux à une requête portant sur leurs caractéristiques visuelles[I].

Nous mettons dans ce projet un accent particulier sur l'interaction entre l'utilisateur et le système pendant l'évaluation de la requête. Nous croyons qu'un système efficace doit utiliser le jugement que l'utilisateur porte sur la similitude entre images retournées, afin d'améliorer sa représentation interne de la requête et de permettre l'apprentissage d'une meilleure mesure de similitude pour le système.

Une requête sera donc un processus qui aura une étendue temporelle, ne portant pas seulement sur un seul événement. Ce principe s'appelle "***relevance feedback***"; bien que courant dans les systèmes de recherche textuelle, son emploi dans le contexte des bases de données d'images est encore extrêmement rare.

En nous inspirant des grands systèmes de recherche de textes, nous voudrions un système qui représente les images et leurs caractéristiques en utilisant une structure de données connue sous le nom de fichier inversé ("***inverted file***")[II]. Cette structure permet le stockage de très grandes quantités de données et s'adapte bien au cas de données éparses. Les images peuvent donc avoir un très grand nombre de caractéristiques[III], pour autant que chaque image ne soit décrite que par quelques unes des caractéristiques possibles.

Dans ce travail de diplôme, il s'agit de:

- spécifier un protocole de communication entre l'interface et la base de données.
- développer une interface graphique permettant des requêtes utilisant la méthode du "*relevance feedback*". L'interface doit pouvoir être connectée à différentes bases de données d'images.
- implanter une base de données structurée selon l'approche du fichier inversé, ainsi que les mécanismes de requêtes permettant d'accéder à cette base de données.

L'implantation de l'interface graphique doit être en langage ***Java***. L'implantation de la base de données elle-même doit être en langage ***C++***, sur station *Sun/Unix*.

2 Introduction

2.1 *Les systèmes déjà existants*

Au cours des dernières années, les systèmes multimédia sont devenus une part importante de la vie quotidienne, sur le *World Wide Web* aussi bien dans les environnements de publication. La caractéristique propre aux documents multimédia est la présence d'images statiques ou de séquences vidéo. Aussi existe-t-il une forte demande en systèmes qui permetteraient aux utilisateurs de créer et de manipuler d'une manière efficace une base de données d'images, et d'y effectuer des recherches d'images basées sur la similitude.

Les systèmes de recherche d'images actuels sont encore loin d'utiliser le critère véritable de la similarité d'images recherchée. Cependant, il existe plusieurs systèmes ***CBIRS***[1] commerciaux sur le marché. On peut citer le système ***QBIC***[2] d'IBM[IV], le système *Visual Retrieval Ware*[V] de *Excalibur Technology*, et finalement le moteur de recherche *Virage Visual Information Retrieval*[VI].

Des systèmes de recherche d'images sur le *World Wide Web* ont également fait leur apparition, comme le système *ImageRover*[VII] et le système *Excalibur Technology* qui a récemment introduit *Internet Spider* et *Image Surfer*.

L'apparition de ces systèmes commerciaux n'indique pas une maturité de la technologie dans ce domaine, mais uniquement une forte demande de ce type de systèmes.

2.2 *Les caractéristiques des images*

2.2.1 Les caractéristiques globales

- **Couleur**

De loin, la caractéristique des images la plus utilisée est la couleur[VIII], [IX], [X], [XIV]. Dans plusieurs systèmes, on utilise seulement les propriétés *globales* de la couleur. Ces dernières sont habituellement calculées dans un espace de couleurs en ayant l'espoir qu'elles correspondent à la perception humaine aux différences de couleurs (par exemple *HSV* ou *CIE*). La représentation la plus souvent utilisée de ces propriétés est l'histogramme de couleurs. *Histogram Intersection* définit la similarité entre deux histogrammes, comme la somme (normalisée) des nombres de pixels en commun entre chaque paire de colonnes. Un désavantage de cette mesure est qu'elle ne tient pas compte de la similarité perceptuelle entre les colonnes. Des mesures qui utilisent une

[1] *Content Based Image Retrieval Systems*
[2] *Query by Image and Video Content*

matrice de colonnes de similarité entre colonnes existent, mais la difficulté réside dans le choix des coefficients, et la complexité de calcul est une fonction quadratique du nombre de colonnes.

- **Texture**

Pour mieux caractériser une image, beaucoup de systèmes utilisent la texture. Il existe une variété importante de caractéristiques de texture disponibles pour les concepteurs de système:

- Les hiérarchies de filtres de *Gabor*[XI].
- Les caractéristiques de *Wold*[XII] utilisées dans le système *Photobook* du *MIT*[XIII].
- Les caractéristiques de grossièreté (*coarseness*), de contraste et de directionnalité (directionality) utilisées dans le système *QBIC* d'IBM[XIV].
- Les décompositions basées sur les ondelettes (*wavelet-based decompositions*)[XV].
- Et beaucoup d'autres…

Lorsque de telles caractéristiques sont globales, elles présentent les mêmes inconvénients que les caractéristiques de couleur citées précédement.

- **Forme**

La troisième classe de caractéristiques qui apparaît fréquemment dans les systèmes ***CBIRs*** actuels est basée sur la forme. Ces caractéristiques sont également d'habitude globales: chaque image est présumée contenir une seule forme, et ceci demande souvent une séparation de l'objet et du fond. Cette restriction signifie que les caractéristiques de forme conviennent la plupart du temps aux images tirées de domaines restreints. Un bon exemple est la correspondance modale, une méthode basée sur la déformation, qui était appliquée aux poissons isolés, lapins et les outils de machines[XVI] par exemple. D'autres approches basées sur la forme comprennent:

- une représentation des courbes à plusieurs échelles[XVII], [XVIII].
- Les histogrammes des directions des contours (*edge directions*), qui étaient appliqués aux marques de fabriques[9].
- Et la recherche de templates de formes simples tels que les angles, les segments de droites et les arcs circulaires[XIX].

2.2.2 Les caractéristiques locales

De nombreux chercheurs ont reconnu que les caractéristiques globales sont insuffisantes pour beaucoup de cas de recherches d'images. Les utilisateurs sont souvent intéressés à une distribution spatiale (*spatial layout*) des couleurs, des textures et des formes dans une image, et peuvent n'être intéressés qu'à certains objets. Il y a ceux qui ont approché ce problème en

cherchant les caractéristiques qui retiennent l'information spatiale aussi bien que d'autres propriétés de l'image, telles que la décomposition en ondelettes (*wavelets*). Cependant, ces dernières sont habituellement extrêmement vulnérables aux petites modifications des objets dans l'image. Une autre approche pour extraire l'information locale est de segmenter l'image en régions, et d'en extraire les propriétés spatiales telles que leurs tailles, leurs emplacements et leurs relations entre elles. Cette approche ramène le problème de recherche d'images à un problème non trivial d'assortiment de graphes étiquetés.

2.3 *La similarité*

Les systèmes ***CBIR*** ont pour but de trouver des images similaires à une image exemple, à un croquis ou à une collection de régions. Il faut remarquer que la signification de "similarité" dans ce contexte est rarement discutée. Ceux qui l'ont approché ont découvert sa difficulté, par exemple, "*the results of the subjective test indicated that human judgments of shape similarity noticeably differ*"[1]. La similarité entre les images est typiquement définie en utilisant la distance entre les points qui représentent l'image dans un espace de caractéristiques multidimensionnel. Les systèmes métriques présument l'affirmation suivante: les images proches d'une image requête sont "similaires" à l'image de la requête. Cependant, le but de tels systèmes est de trouver, suite à une requête, des images qui sont similaires selon les perceptions de l'utilisateur. Le fait que ces deux notions de similarité peuvent être très différentes est rarement discuté. Il est souvent sous-entendu que si on choisit les "bonnes" caractéristiques (un espace de couleur convenable, des caractéristiques de texture "correspondant à la perception humaine", etc.), alors la proximité dans l'espace des caractéristiques doit correspondre à la similarité de la perception.

Il y a plusieurs raisons d'avoir des doutes sur ces suppositions dont la plus fondamentale est **l'hypothèse de la métrique**. Il apparaît évident selon la psychophysique que les jugements humains sur la similarité n'obéissent pas aux demandes d'une métrique. Spécifiquement: "*[Self-identity] is somewhat problematic, symmetry is apparently false, and the triangle inequality is hardly compelling*"[2]. Pour la recherche d'images, le manque de symétrie est un problème très important. Essentiellement, les caractéristiques qui sont significatives en calculant la similarité, dépendent de quelles paires de termes sur lesquelles la requête est basée: la variante est plus similaire au prototype que vice versa. Même s'il y a eu quelques tentatives préliminaires pour appliquer l'ensemble des fonctions de la similarité basées sur la théorie de *Tversky* aux systèmes ***CBIR***, la littérature psychophysique sur la similarité semble avoir été largement ignorée par les chercheurs qui travaillent sur les systèmes ***CBIR***.

Cependant, quelques auteurs ont considéré le fait que la distance dans l'espace des caractéristiques n'est pas nécessairement équivalent à la similarité de la perception. Les cartes auto-organisantes ont été utilisée pour regrouper des caractéristiques de texture selon des étiquettes de classes fournies par des utilisateurs. Les Réseaux d'Apprentissage de Distance (*Distance Learning Networks*) ont été utilisés pour essayer de faire apprendre une mise en correspondance de l'espace des caractéristiques à l'espace de la similarité perceptuelle en utilisant les données des jugements humains sur la similarité.

[1] les résultats de test subjectif ont indiqué que les jugement humains sur la similarité de formes sont visiblement différents.
[2] l'auto-identité est un peu problématique, la symétrie est apparemment fausse et l'inégalité du triangle est à peine astreignante

3 Le système *Viper*

3.1 *Introduction*

Le système[XX] que nous devons implanter dans ce projet est baptisé *Viper*[1]. Il s'inspire des systèmes de recherche de texte. Son but est d'utiliser un grand nombre de caractéristiques simples. La version actuelle de ce système utilise la couleur locale et globale des images, les caractéristiques de fréquence spatiale extraites à plusieurs échelles, ainsi que leurs statistiques en fréquences dans les images et dans la collection entière. Le but ici est de mettre à disposition du système des caractéristiques de bas niveau correspondant approximativement à celles auxquelles le système visuel humain est sensible.

La différence fondamentale entre la vision par ordinateur traditionnelle et les applications de base de données d'images est la présence d'un humain "**dans la boucle**". Le système travaille avec des caractéristiques de bas niveau, et une interaction avec l'utilisateur permet d'affiner une combinaison de ces caractéristiques qui correspondent à ses désirs.

Plus de 80'000 caractéristiques sont disponibles dans le système, tel que la couleur modale des différentes régions ou les énergies des moyennes quantifiées des sorties des filtres de *Gabor* à différentes orientations et échelles. Chaque image possède un nombre de l'ordre de $O(10^3)$ caractéristiques de ce type, la correspondance entre caractéristiques et images étant stockée dans une structure de fichier inversé.

3.2 *Fichier inversé*

Les "*inverted files*" ou fichier inversés sont les structures de données les plus souvent utilisées dans les systèmes de recherche de texte. Une structure "fichier inversé" contient une entrée pour chaque caractéristique possible (terme), consistant en une liste d'images (documents) qui contiennent cette caractéristique, ainsi que la fréquence de cette caractéristique dans les images. La communauté des bases de données basées sur les informations textuelles, a développé des techniques pour la construction et recherche dans les fichiers inversés très efficaces. Cette structure est discutée plus en détail dans le chapitre 7.

3.3 *Caractéristiques de couleur*

Il est préférable que l'espace de couleurs utilisé dans un système de recherche d'images soit perceptuellement uniforme, ce qui veut dire que de petits changements dans les coordonnées de la couleur doivent correspondre à de faibles différences perceptuelles. L'espace *RGB* ne possède pas cette propriété. L'espace de couleur ***HSV*** offre une meilleure uniformité perceptuelle et il est plus facile à calculer que les systèmes tels que *CIE-LUV* ou *CIE-LAB*.

[1] *Visual Information Processing for Enhanced Retrieval*

Le système *Viper* utilise une palette de 166 couleurs, dérivées uniformément par la quantification de l'espace de couleurs cylindrique ***HSV*** en 18 teintes, 3 saturations et 3 valeurs. Elles sont augmentées de 4 niveaux de gris. Ce choix de quantification signifie que plus de tolérance est donnée aux changements au niveau de la saturation et de la valeur. Ce choix est souhaitable étant donnée que ces canaux peuvent être influencés par les conditions d'illumination et de point de vue.

Deux ensembles de caractéristiques sont ensuite extraits de l'image ***HSV*** quantifié. Le premier est équivalent à un histogramme de couleurs conventionnel, avec la différence que les colonnes contenant zéro pixels sont écartées. Il y a 166 caractéristiques d'histogrammes de couleurs possibles, et la plupart des images n'en contiennent que 40 environ.

La seconde classe de caractéristique représente les propriétés locales de la couleur de l'image. L'image est divisée en blocs carrés à quatre échelles, allant de 16x16 à 128x128. La couleur modale est calculée pour chaque bloc. L'occurrence d'une couleur donnée dans un bloc particulier est traitée comme une caractéristique binaire. Pour nos images de 256x256, ce sont 56'440 caractéristiques de blocs de couleurs possibles. Chaque image possède 340 de ces caractéristiques.

3.4 *Caractéristiques de texture*

Les filtres de ***Gabor*** bidimensionnels ont été fréquemment proposés comme une structure permettant de décrire et comprendre les propriétés sélectives de l'orientation et de la fréquence des neurones dans le cortex visuel. Les bancs des filtres de ***Gabor*** ont souvent été appliqués à la segmentation et la classification de la texture, aussi bien qu'aux tâches de la vision en général. Nous employons un banc de filtres de ***Gabor*** réels symétriques d'une manière circulaire, définis dans le domaine spatial par:

$$f_{mn}(x,y) = \frac{1}{2\pi\sigma_m^2} e^{-\frac{x^2+y^2}{2\pi\sigma_m^2}} \cos(2\pi(\mu_{0m} x\cos\theta_n + \mu_{0m} y\sin\theta_n)) \qquad \text{EQ 1}$$

où m indexe les échelles des filtres, et n leurs orientations. La fréquence centrale du filtre est spécifié par μ_{0m}. La moitié du pic de la bande passante radiale est donnée par:

$$B_r = \log_2\left(\frac{2\pi\sigma_m\mu_{0m} + (2\ln 2)^{\frac{1}{2}}}{2\pi\sigma_m\mu_{0m} - (2\ln 2)^{\frac{1}{2}}}\right) \qquad \text{EQ 2}$$

B_r est choisie égale à 1 (par exemple une bande passante d'une octave), ce qui nous permet de calculer ensuite σ_m:

$$\sigma_m = \frac{3(2\ln 2)^{\frac{1}{2}}}{2\pi\,\mu_{0m}} \qquad \text{EQ 3}$$

La plus grande fréquence centrale est choisie comme $\mu_{0_1} = \frac{0.5}{1+\tan(1/3)} \approx 0.5$ pour qu'il soit à l'intérieur du domaine de fréquence discrète. La fréquence centrale diminue de moitié à chaque changement d'échelle, ce qui implique que σ est doublé (voir équation EQ3). L'orientation des filtres varie en pas de $\pi/4$, et trois échelles sont utilisées. Ces choix aboutissent à un ensemble de 12 filtres qui donne une bonne couverture du domaine de fréquence avec peu de chevauchement entre les filtres. Pour une implantation pratique, les filtres sont tronqués à 3σ, donnant des noyaux de tailles 9x9, 17x17 et 35x35.

L'utilisation de filtres symétriques circulaires signifie que l'équation EQ2 est séparable. La convolution 2-D peut être ainsi calculé en utilisant quatre convolutions 1-D, ce qui réduit le nombre de calculs demandés pour un noyau de *N x N* par un facteur d'ordre *N*.

Ces filtres sont appliqués à l'image et l'énergie moyenne de chaque filtre est calculé pour chaque bloc de taille 16x16 de l'image. L'énergie est ensuite quantifié en 10 bandes, qui sont choisies en examinant les histogrammes d'énergie du filtre à chaque pixel pour 500 images. Une caractéristique est stockée pour chaque filtre qui a une énergie dans une bande plus grande que la bande [0, 2). Ce qui signifie qu'il y a 27'648 caractéristiques possibles pour une image de 256 x 256, et une image donnée pouvant en avoir au plus 3'072. Les histogrammes des moyennes des sorties du filtre sont aussi stockés, donnant une mesure globale des caractéristiques de la texture de l'image.

3.5 *Le calcul de la similarité et la Relevance Feedback*

La ***Relevance Feedback*** est reconnue comme extrêmement utile dans les applications de recherche de textes, et elle a été appliquée dans quelques systèmes ***CBIRS***. Dans une application de base de données d'images, la *Relevance Feedback* offre deux avantages. Le premier est la possibilité d'augmenter la requête avec les caractéristiques des images significatives retrouvées, afin de produire une requête qui représente mieux l'attente de l'utilisateur. Le deuxième avantage est propre au problème de recherche d'images. Dans un système de recherche de textes, l'extraction de la caractéristique est libre: les mots composant le document sont eux même les caractéristiques du document. Ce n'est pas le cas dans la recherche d'images. Nous envisageons un système dans lequel des caractéristiques coûteuses en temps calcul sont extraites des images lorsque la base de données est construite, bien que les caractéristiques soient nombreuses et très coûteuses à évaluer pour une nouvelle image de la requête. Néanmoins, une fois que quelques images sont recherchées en utilisant un sous-ensemble de caractéristiques moins coûteuses, des caractéristiques complexes peuvent être ajoutées à la requête via la *relevance feedback*.

L'utilisation de la structure de données du fichier inversé signifie que l'adjonction des caractéristiques "rares" ajoute un coût à l'évaluation de la requête seulement quand elles sont susceptibles d'être significatives.

La similarité entre les images de la base de données et les images de la requête de référence est toujours calculé comme s'il s'agissait d'une requête de *relevance feedback* – la première requête consistant simplement à une seule image significative. Les caractéristiques sont combinées en une somme de leurs fréquences d'occurrence df (les blocs de caractéristiques ont une fréquence de 1). Pour une requête q contenant N images i avec des niveaux de "signifiance" $R_i \in [-1,+1]$ et des caractéristiques j avec des fréquences df_{ij}, nous avons:

$$df_{qj} = \frac{1}{N} \sum_{i=1}^{N} df_{ij} . R_i \qquad \text{EQ 4}$$

Les caractéristiques peuvent être évaluées dans l'ordre suivant df_{qj}, permettant à la recherche d'être élaguée.

Pour chaque caractéristique j des M caractéristiques dans q, la liste des images contenant cette caractéristique est recherchée et ajoutée au groupe des images candidates. Pour les caractéristiques qui ne font pas partie d'un histogramme, le score s_k de chaque image k est mis à jour selon la formule:

$$s_{k_{now}} = s_{k_{old}} + df_{qj} df_{kj} \log cf_j^{-1} \qquad \text{EQ 5}$$

où cf_j est la fréquence de la caractéristique j dans la totalité de la base de données. Cette équation provient des systèmes classiques de recherche de textes. Sa devise est très simple: les caractéristiques qui sont communes dans une image caractérisent bien cette image; les caractéristiques qui sont communes dans la collection à laquelle appartient cette image, ne distinguent pas bien cette image des autres. Les schémas de réduction de la dimension tels que l'analyse en composants principaux[1] éliminent ces dimensions "rares" qui peuvent en effet être très utiles pour créer une requête spécifique. Pour les caractéristiques d'histogramme, le score est mis à jour selon la formule:

$$s_{k_{now}} = s_{k_{old}} + sign(df_{qj}) \min(|df_{qj}|, df_{kj}) \log cf_j^{-1} \qquad \text{EQ 6}$$

qui est une variante pondérée de l'intersection de l'histogramme standard.

[1] *Principal Components Analysis*

4 Protocole de communication

4.1 *But du protocole de communication*

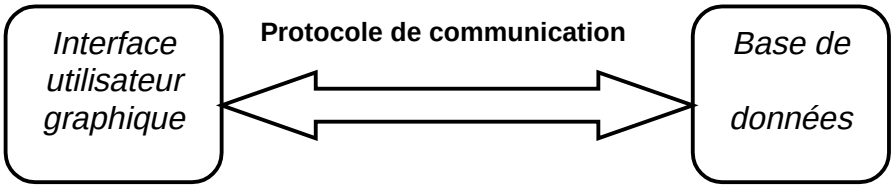

Figure 1. Protocole de communication

Le système *Viper* de recherche d'images similaires est formé de trois entités comme le montre la figure 1. La première entité est composée d'une interface graphique où un utilisateur peut formuler des requêtes de recherche d'images et visualiser les résultats de la recherche. La deuxième entité comporte une base de données où un ensemble de caractéristiques d'images est stocké d'une manière spécifique et inspirée des systèmes d'archivage de textes. La troisième entité est un protocole de communication reliant les deux premières entités pour leur permettre de communiquer.

L'idée principale dans ce travail, à part l'utilisation de la structure du fichier inversé, est d'établir et de mettre en place un protocole de communication standard entre l'interface graphique de l'utilisateur et la base de données structurée en fichier inversé. Ce protocole permet de transférer différents types de messages entre l'interface et la base de données comme le montre la figure 1. Le protocole joue trois rôles importants dans notre système:

- il permet un choix flexible de l'interface et de la base de données: on peut connecter n'importe quelle interface à n'importe quelle base de données à partir du moment où elles respectent le protocole imposé par le système. On peut ainsi créer plusieurs versions d'interfaces .

- il permet de laisser des traces en mémoire des actions de l'utilisateur et d'établir des statistiques sur les interactions de différents utilisateurs avec la base de données depuis une interface. Ces traces pourraient être utilisées pour un apprentissage les caractéristiques des utilisateurs individuels afin de modifier le système de recherche d'images, afin qu'il utilise cette connaissance.

- La *relevance feedback* est très facile à mettre en œuvre avec un protocole de communication.

Il reste à spécifier un protocole de communication qui peut être facile à interpréter et offrant également des possibilités d'extension par le rajout de nouvelles fonctionnalités afin de l'améliorer. La version actuelle du protocole ne permet que la recherche d'images, mais sa

spécification permet son extension à d'autre systèmes de recherches comme les vidéos ou les informations textuelles etc.

4.2 *Spécification du protocole*

La communication entre l'interface utilisateur et la base de données doit suivre un protocole bien spécifié. On doit se pencher maintenant sur le contenu de ce protocole et connaître les différents messages que doivent s'échanger l'interface et la base de données.

Nous avons défini le protocole de communication comme une suite de messages qui diffèrent les uns des autres par leur type et leur contenu. Cette structure permet au protocole d'être facilement extensible et mieux interprété par l'interface et la base de données. Un message du protocole de communication est composé de deux parties. Une partie qui définit son type et l'autre partie qui définit son contenu comme le montre la figure 2.

Structure d'un message

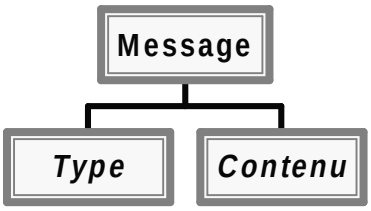

Figure 2.

Il est difficile de spécifier à l'avance les différents types de messages pouvant transiter entre l'interface et la base de données pour notre système *Viper*. Nous avons commencé avec une première version du protocole qui contient neuf types de message, mais cette version peut être améliorée au fur et à mesure selon les besoins du système. Les différents types de messages sont énoncés dans les chapitres qui suivent.

4.2.1 Le message "*HANDSHAKE_INTERFACE*"

C'est un message de bienvenue de l'interface à la base de données. Dans le contenu du message, comme le montre la figure 3, l'interface spécifie la version du protocole et le nom de l'utilisateur.

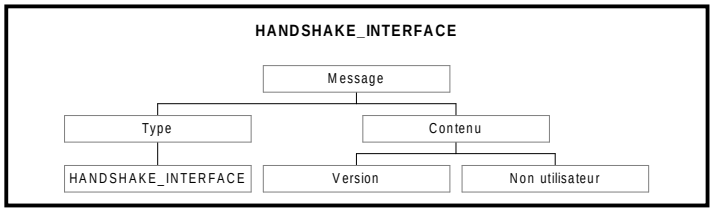

Figure 3

4.2.2 Le message "*HANDSHAKE_SERVER*"

C'est le message de bienvenue de la base de données à l'interface. Le message contient, comme le montre la figure 4, en plus de la version du protocole de communication, les champs suivants:

- **Sessions**: Dans le système *Viper*, chaque utilisateur de l'interface peut créer une ou plusieurs sessions. La base de données garde en mémoire toutes les sessions de l'utilisateur qui sont proposées lors du message de bienvenue. Les sessions correspondent aux deux champs suivants:
  - **Nombre de Sessions**: correspond au nombre de sessions antérieures de l'utilisateur.
  - **Nom de Sessions**: correspond à une liste de noms des sessions.

- **Codes**: un système de recherche d'images comme le nôtre doit prévoir bien entendu les messages pour toutes les erreurs qui peuvent survenir lors de l'utilisation de l'interface. La base de données possède déjà une liste de messages avec leurs codes d'erreurs. Cette liste est envoyée à l'interface en utilisant les trois champs suivants:
  - **Nombre de Codes**: le nombre de codes d'erreurs à envoyer à l'interface.
  - **Code Erreur**: une liste des codes des erreurs.
  - **Message Err.**: les messages correspondants aux codes d'erreurs.

- **Collections**: ce champ est prévu afin que la base de données puisse envoyer à l'interface une liste de toutes les collections d'images qu'elle possède. Pour chaque collection, une liste d'index d'algorithmes de recherche applicables à cette collection est prévue dans le champ. Ce champ comprend trois autres sous-champs:
  - **Nombre de Coll.**: nombre de collections existantes dans la base de données.
  - **Nom de la Coll.**: une liste des noms de chaque collection.
  - **Nombre d'Index**: le nombre d'index d'algorithmes de recherche pour chaque collection.
  - **Index d'Algorithmes**: comprend la liste des index.

- **Algorithmes**: ce champ contient une liste des noms de tous les algorithmes de recherche et leurs index. Après l'envoi de ces paramètres, seuls les index des algorithmes seront envoyés ensuite entres l'interface et la base de données. Il possède trois champs:
  - **Nombre d'Algor.**: nombre d'algorithmes de la base de données.

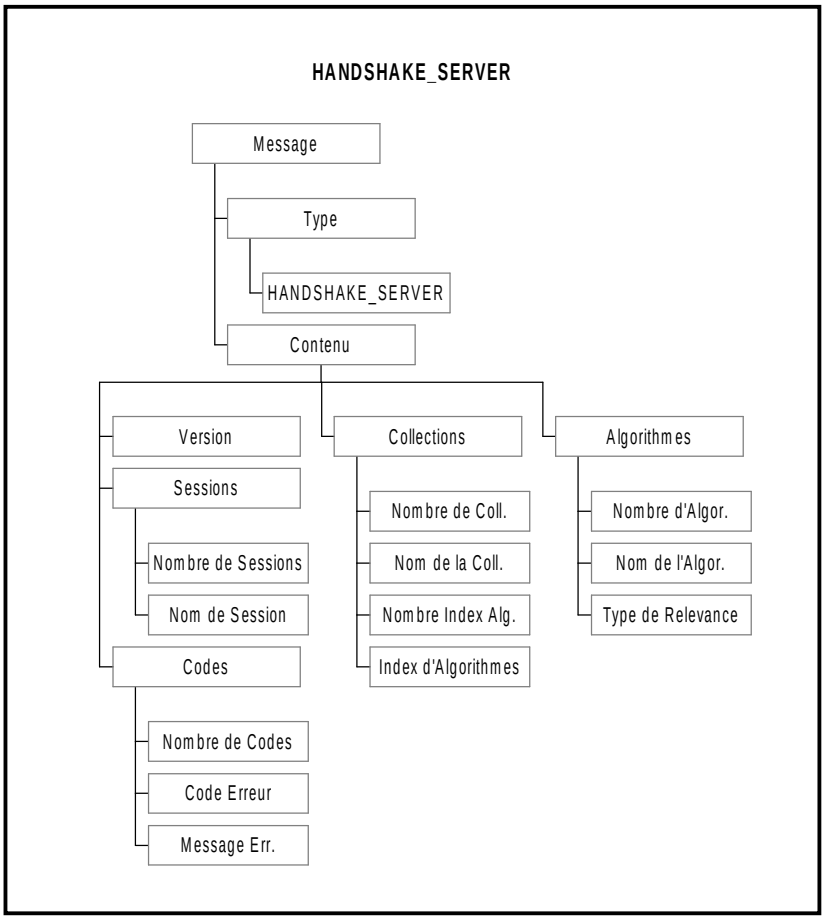

Figure 4

- **Nom d'Algor**.: liste des noms des algorithmes.
- **Type de Relevance**: type de "*relevance*" (caractère significatif) que l'algorithme peut traiter.

4.2.3 Le message "*INTERFACE_EVENTS*"

Dans ce type de message, l'interface spécifie à la base de données les événements que l'utilisateur effectue sur l'interface. C'est un champ très utile pour établir des statistiques des sessions des utilisateurs. Le champ comprend, comme le montre la figure 5, un code d'événement et une extension qui pourrait être n'importe quelle information additionnelle sur cet événement.

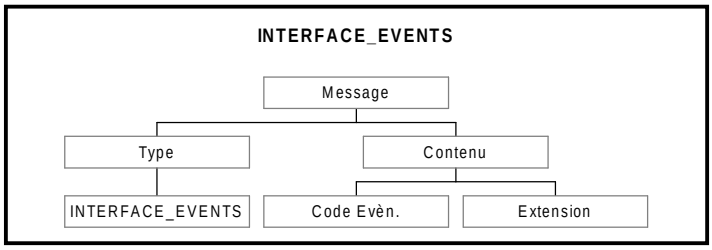

Figure 5

4.2.4 Le message "*REQUEST*"

C'est le champ qui permet à l'interface d'envoyer une requête à la base de données. Il comprend (voir la figure 6):

- **Nom Collection**: le nom de la collection des images.
- **Algorithme**: l'index de l'algorithme de recherche à appliquer.
- **Nombre maximum**: le nombre maximum d'images souhaité en réponse.
- **Seuil**: un seuil à appliquer lors de la recherche d'images. On tient compte de la valeur de ce seuil si le champ **Nombre maximum** ne figure pas dans ce message.
- **Nombre d'Éléments**: le nombre d'éléments envoyés dans la requête. On parle ici d'éléments dans la mesure où l'on voudrait étendre la fonctionnalité du système à la recherche d'autres objets comme par exemple du texte, de la vidéo etc. On pourrait même envoyer des régions d'images comme requête.
- **Éléments**: c'est la liste des éléments correspondants au champ **Nombre d'Éléments**. Il comprend les champs suivants:
  - **Type d'élément**: ce champ indique s'il s'agit d'images ou d'autres types d'éléments. Dans la version actuelle du protocole, on se contente des images. C'est un champ qui permet l'extension à d'autres types.
  - **URL**: correspond à l'*URL* de l'élément.
  - **Niveau de Relevance**: indique le niveau de "*relevance*" de cet élément de la requête. Chaque élément a trois niveaux de "*relevance*" possibles: *Relevant*, *Non-Relevant* ou *Neutral* selon le choix de l'utilisateur.
  - **Similarité**: ce champ n'est pas utilisé pendant une requête. Il est utilisé lors d'une "*END_SESSION*" à une requête comme expliqué plus loin dans le message "*RESPONSE*".

- **Extension**: un champ supplémentaire est laissé libre dans la version actuelle du protocole qui permet d'ajouter une information sur l'élément.

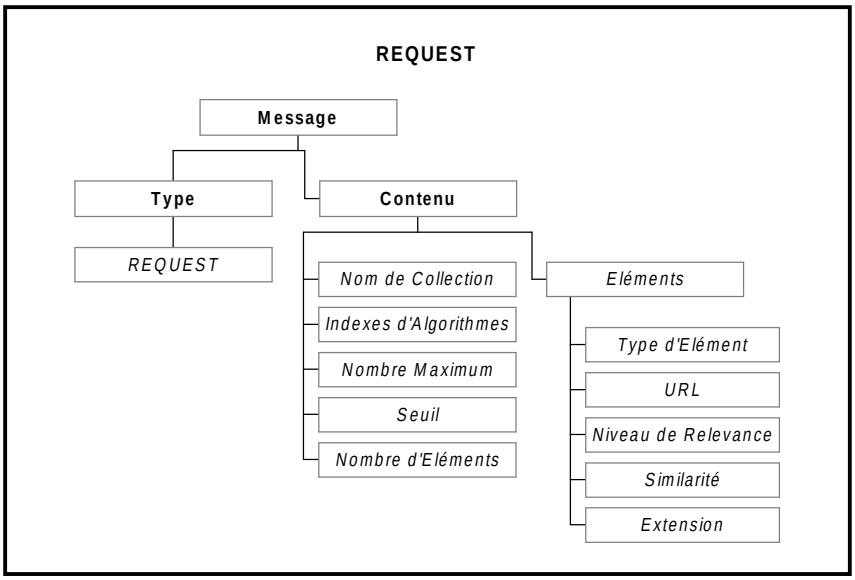

Figure 6

4.2.5 Le message "*LOAD*"

Ce type de message est envoyé par l'interface à la base de données pour demander des éléments à charger sur l'interface. Quand ils sont chargés sur l'interface, ils peuvent servir pour lancer des requêtes à la base de données. Ces éléments peuvent être choisis d'une manière aléatoire ou fixe par la base de données. Le contenu de ce message comprend comme le montre la figure 7:

- **Nom de Collection**: la collection d'éléments qu'on souhaite charger sur l'interface. C'est un champ qui est nécessaire dans le cas où le type de chargement est aléatoire.

- **Type de Chargement**: l'interface spécifie dans ce champ si les éléments doivent être choisis d'une manière aléatoire ou fixe.

- ***URL***: c'est un champ qui est utilisé dans le cas où le type de chargement est fixe. Dans ce champ, on spécifie à la base de données l'*URL* d'un élément chargé sur l'interface depuis le *World Wide Web* pour être utilisé comme image d'une requête. Ce champ permet à la base de données de rajouter un nouvel élément à la liste déjà existante et d'en extraire ses caractéristiques pour les stocker avec celles des autres éléments.

- ***Nombre d'Éléments***: c'est le nombre d'éléments que la base de données doit envoyer aléatoirement à l'interface.

- ***Type d'éléments***: c'est le même type de champ que celui indiqué dans le message "REQUEST".

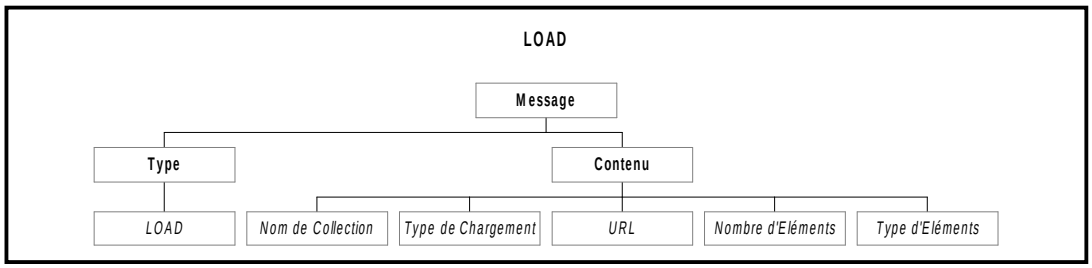

Figure 7

4.2.6 Le message "*RESPONSE*"

C'est le message de réponse de la base de données à l'interface suite au message "*REQUEST*" ou au message "*LOAD*" de l'interface. Comme le montre la figure 8, le contenu de ce message comprend:

- **Nombre d'Éléments**: c'est le nombre d'éléments similaires à une requête ou le nombre d'éléments à charger sur l'interface suite à un message "*LOAD*".

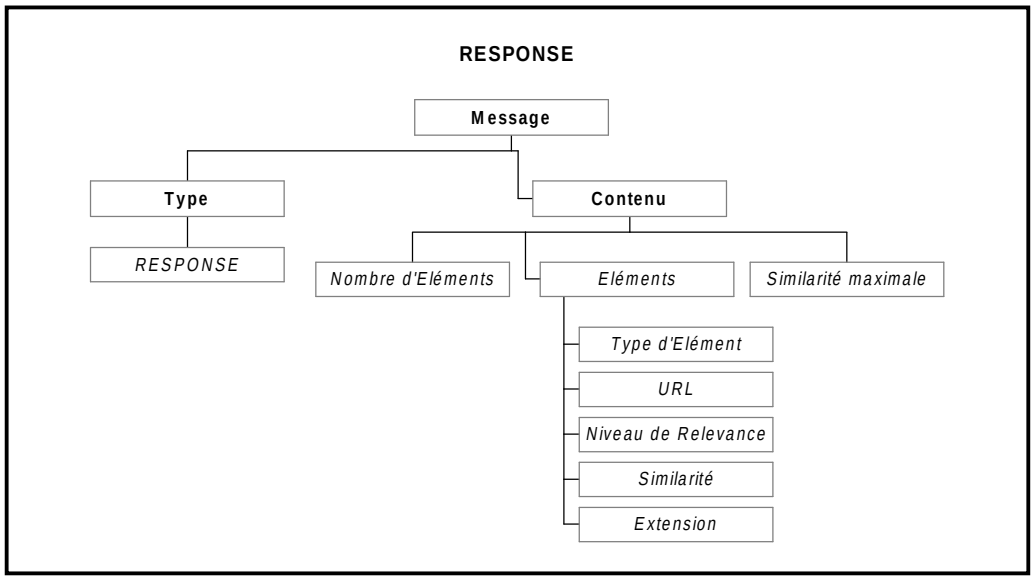

Figure 8

- **Éléments**: c'est le même champ que celui indiqué dans le message "*REQUEST*", sauf que dans ce cas les similarités retournées pour chaque élément sont définies par rapport à une requête.

- **Similarité maximale**: c'est un champ qui indique à l'interface la similarité maximale des images trouvées par rapport à la requête.

4.2.7 Le message "*ERROR*"

Comme son nom l'indique, ce type de message permet d'échanger les erreurs qui pourraient survenir soit au niveau de l'interface, soit au niveau de la base de données, soit au niveau du protocole de communication lui-même. Il ne contient qu'un seul champ "**Code d'Erreur**", comme le montre la figure 9, qui correspond au code d'erreur du message d'erreur correspondant.

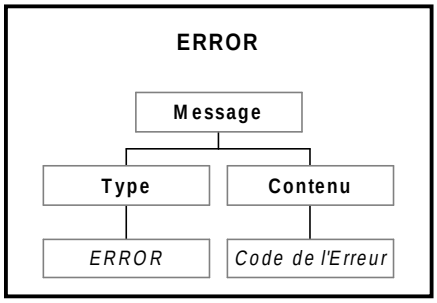

Figure 9

4.2.8 Le message "*TAKE_BACK_SESSION*"

C'est un type de message qui permet à un utilisateur de reprendre une session qu'il avait déjà entreprise et que la base de données avait auparavant envoyé dans le message de bienvenue à l'interface. L'interface indique à la base de données, le nom de la session à reprendre, et qui est le seul contenu du message comme le montre la figure 10.

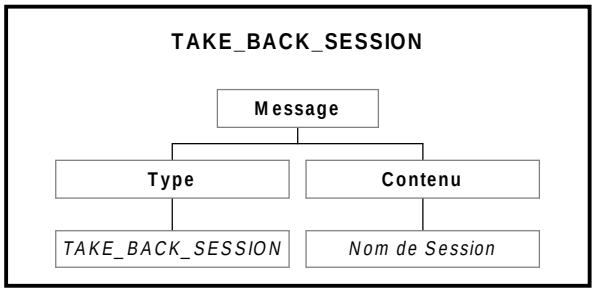

Figure 10

4.2.9 Le message "*END_SESSION*"

Ce type de message est envoyé à la base de données quand l'utilisateur quitte l'interface et qu'il voudrait sauver la session. Il comprend un seul contenu indiquant le nom de la session comme le montre la figure 11.

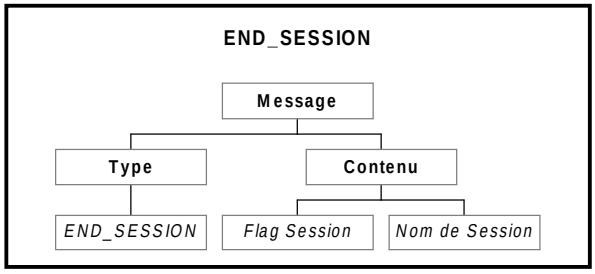

Figure 11

4.3 *Introduction à la grammaire*

Le but du système *Viper* n'est pas seulement de spécifier un protocole de communication, mais également de construire une grammaire qu'on doit introduire du côté de l'interface et du serveur connecté à la base de données. Cette grammaire est une sorte de langage informatique qui sera compilé par les outils de compilation ***Java*** du côté interface et de ***C*++** du côté serveur. Dans ce paragraphe, on va parler uniquement de la grammaire, mais les outils de compilation de cette grammaire seront discutés dans le chapitre 5.

Pour donner une notion de la grammaire[XXIII], celle-ci est définie comme un ensemble de règles permettant de définir la syntaxe et la sémantique d'un langage de programmation. Le langage est un ensemble de phrases satisfaisant à une certaine forme. Chaque phrase est une séquence de mots formés à partir d'un alphabet.

D'une manière plus formelle, une grammaire est un quadruplet: (Terminaux, Non_terminaux, Productions, Axiome). Dans ce quadruplet définissant une grammaire:

- **Terminaux** est l'ensemble des mots formant les phrases.
- **Non_terminaux** est l'ensemble des notions dites non-terminales décrivant des morceaux de phrases.
- **Productions** est l'ensemble des règles grammaticales. Chaque production est de la forme:

  `tête ⇒ corps`

où tête et corps sont des séquences de terminaux ou non-terminaux et où tête contient au moins un non-terminal.

- **Axiome** est une notion non-terminale particulière qui est la racine de la grammaire.

Les ensembles des Terminaux et des Non_terminaux doivent être **disjoints**.

Toutes les grammaires n'ont pas la même puissance d'expression, ni la même facilité de mise en œuvre. Elles sont donc classées (classification de *Chomsky*) selon les restrictions imposées sur les règles de production en quatre type de classes:

- **Type 3**: c'est celui des grammaires régulières. Toutes les productions sont de la forme:

  `notion ⇒ terminal autre_notion_ou_la_même`

  auquel cas elle est dite régulière à droite.
  ou de la forme:

  `notion ⇒ autre_notion_ou_la_même terminal`

  qui est dite régulière à gauche.
  Le type 3 est bien adapté pour l'analyse lexicale et peut être traité par un **automate fini déterministe**.

- **Type 2**: c'est celui des grammaires indépendantes du contexte. Toutes les productions ont exactement un non-terminal dans la tête et sont donc de la forme:

  `notion ⇒ séquence_de_terminaux_ou_non_terminaux`

  Le type 2 est plus utilisé dans l'analyse syntaxique et conduit à une analyse par un **automate à pile**.

- **Type 1**: c'est celui des grammaires dépendantes du contexte. Toutes les productions sont alors de la forme:

  `séquence_tête ⇒ séquence_corps`

  où les deux séquences composant la production sont formées de terminaux et de non-terminaux, avec la contrainte qu'il n'y en a pas plus dans la tête que dans le corps et que la `séquence_tête` n'est pas vide.

  `0 < longueur(séquence_tête) ≤ longueur(séquence_corps)`

  Chaque symbole terminal ou non-terminal compte pour un dans le calcul de la longueur de la séquence.

- **Type 0**: c'est celui des grammaires récursivement énumérables. Ce sont les grammaires de *Chomsky* les moins contraintes. Toutes les productions sont dans ce cas de la forme:

`séquence_tête ⇒ séquence_corps`

où les deux séquences composant la production sont formées de terminaux et de non-terminaux, avec la seule contrainte que la `séquance_tête` n'est pas vide.

`0 < longueur(séquence_tête)`

4.4 *Grammaire du protocole du communication*

La grammaire est choisie de telle manière qu'elle soit facile à comprendre, ce qui a rendu la tâche un peu difficile à réaliser. Après plusieurs versions d'essai, nous avons réussi à mettre au point une grammaire de type 2, qui normalise la manière de construire un message du protocole de communication et peut avoir une séquence de plusieurs terminaux et non-terminaux. Chaque message a la syntaxe grammaticale **GR1** suivante:

$$\textit{Message} \Rightarrow [\ \textit{Type}\ (\ \textit{contenu}\ )\ ] \qquad \textbf{GR 1}$$

Chaque message commence donc par un crochet ouvert '**[**' et se termine par un crochet fermé '**]**'. Cette syntaxe permet à l'interface et au serveur de détecter le début et la fin du message. Comme indiqué dans le chapitre 4.2, un total de neuf messages différents sont construits en suivant la formule **GR1**. Les structures grammaticales des messages sont similaires, puisque chaque message est composé d'un type et d'un contenu. On peut prendre comme exemple la production de la grammaire **GR2** du message "*HANDSHAKE_SERVER*" qui contient le plus grand nombre de champs et qui va nous donner la meilleure idée de la syntaxe grammaticale des messages.

Le message "*HANDSHAKE_SERVER*", comme expliqué dans le chapitre 4.4.2, est composé d'un type qui est le terme `BIENVENU_SERVEUR` et du contenu qui est le terme `ListeHANDSHAKE_SERVER` entouré d'une parenthèse ouvrante et d'une parenthèse fermée.

Tous les types des messages sont des symboles non terminaux fixes et ils seront abordés en détail dans le chapitre 5. Dans la grammaire GR2, on peut constater que les termes fixes qualifiés de "symboles non terminaux" sont en majuscule. Il y a d'autres termes qui sont des symboles fixes comme la version du protocole par exemple dans le terme `ListeHANDSHAKE_SERVER`..

Une autre définition syntaxique peut être observée dans la grammaire GR2, il s'agit des listes de termes qui sont toujours mises entre parenthèses comme le terme `ListeHANDSHAKE_SERVER` qui définit le contenu du message *"HANDSHAKE_SERVER"*. S'il y a plusieurs termes de même type, comme les noms des sessions `NomsSessions` par exemple, ils sont séparés par des virgules. On peut trouver une liste d'éléments incluse dans une autre liste comme le cas de la liste `ListeCollAlgorithme` qui contient la liste `ListeIndexAlgorithme`. Les termes de types différents, les termes et les virgules, les termes et les parenthèses, les termes et les crochets, sont tous séparés par un ou plusieurs espaces.

Le symbole "|" signifie un "ou" dans cette grammaire; c.à.d. qu'un terme peut avoir un contenu ou un autre, ces contenus étant séparés par ce symbole.

Avec cette grammaire, il est facile de créer un langage de programmation qui sera interprété ou compilé par l'un des outils de compilation. Ce qui va nous amener au chapitre suivant.

```
HANDSHAKE_SERVER       :   BIENVENU_SERVEUR ( ListeHANDSHAKE_SERVER )

ListeHANDSHAKE_SERVER  :   VERSION Sessions Codes Collections Algorithmes

Sessions               :   NombreSession ( ListeSessions )

ListeSessions          :   ListeSessions , NomsSessions
                           |
                           NomsSessions

Codes                  :   NombreCodes ( ListeCodes )

ListeCodes             :   ListeCodes , NumCode NomMessage
                           |
                           NumCode NomMessage

Collections            :   NombreCollections ( ListeCollAlgorithme )

ListeCollAlgorithme    :   ListeCollAlgorithme , NomCollection NombreIndexAlg ( ListeIndexAlgorithme )
                           |
                           NomCollection NombreIndexAlg ( ListeIndexAlgorithme )

ListeIndexAlgorithme   :   ListeIndexAlgorithme , IndexAlgorithme
                           |
                           IndexAlgorithme

Algorithmes            :   NombreAlgorithmes ( ListeAlgorithmes )

ListeAlgorithmes       :   ListeAlgorithmes , NomAlgorithme TypeRelenvance
                           |
                           NomAlgorithme TypeRelenvance
```

GR 2

5 Outils de compilation

5.1 *Introduction*

Avant de parler des outils de compilation, il convient de rappeler les indications du cahier des charges qui précise que l'implantation de l'interface graphique est à réaliser en langage ***Java*** et celle de la base de données en langage ***C++***. Le choix du langage ***Java*** pour l'interface est expliqué par le désir d'avoir un système distribué et accessible depuis le *World Wide Web*. Notons également le besoin d'avoir une interface graphique portable d'une machine à l'autre, ce que le langage ***Java*** remplit parfaitement en utilisant une *Applet Java* à travers un navigateur sur le réseau Internet. Le choix du langage C++ s'explique par le besoin de rapidité d'exécution lors du traitement des images du côté de la base de données, ainsi que la facilité d'utiliser la librairie *STL*[1] dans ce langage.

L'implantation de l'interface graphique et de la base de données sont discutées respectivement dans les chapitres 6 et 7. Dans le chapitre présent, nous présentons les outils de compilation *JavaCUP* et *JavaLex* qui génèrent le code *Java* , *Lex* et *Yacc* qui génèrent le code en *C++*. Les outils *JavaCUP* et *JavaLex* sont moins connu que *Lex* et *Yacc* du fait que le langage *Java* n'a fait son apparition qu'en 1995, alors que le langage ***C*** est connu depuis 1972. Mais les outils *JavaCUP* et *JavaLex* restent très proche des outils *Lex* et *Yacc* comme vont nous le montrer les paragraphes 5.2 et 5.3 suivants.

Il faut noter qu'au niveau des outils de compilation pour le langage *Java*, il existe également *JavaCC*[2] [3] mais notre choix s'est basé sur les outils ***JavaLex*** et ***JavaCup*** malgré la facilité d'utilisation de *JavaCC*. La différence entre ces différents outils est que *JavaCC* utilise un seul fichier de spécification contenant la grammaire pour générer les codes *Java* des analyseurs, aussi bien lexical que syntaxique, alors que *JavaCup* et *JavaLex* ont besoin chacun d'un fichier de spécification (l'un syntaxique, l'autre lexical). Le choix de ces deux derniers outils est expliqué par la similarité de leurs fichiers de spécifications par rapport à ceux de *Lex* et *Yacc* et la facilité de transformer les fichiers de *Lex* et *Yacc* à l'aide d'un script pour les utiliser pour *JavaCup* et *JavaLex*.

5.2 *Les outils Lex et Yacc*

Dans ce chapitre, nous allons brièvement parler des outils de compilation *LEX* et *YACC* du coté du serveur de la base de données. Ces outils sont fréquemment utilisés pour générer des analyseurs lexicaux et syntaxiques.

[1] *Standard Template Library*
[2] Consulter l'adresse URL http://www.suntest.com/JavaCC/index.html
[3] *Java Compiler Compiler*

L'outil *JavaLEX* est basé sur le modèle du générateur d'analyseurs lexicaux *LEX*. *JavaLEX* a besoin d'un fichier de spécification similaire à celui de *LEX* pour générer un code source en langage *Java* correspondant à l'analyseur lexical, alors que *LEX* génère un code source en langage *C* correspondant à son analyseur lexical. D'ailleurs, le fichier de spécification construit pour *JavaLEX*, peut aisément être modifié par un script écrit en langage *Perl* pour construire celui de l'outil *LEX*. Dans le fichier de spécification de *LEX*, les caractères acceptés par les expressions régulières sont désignés par la variable *yytext*. La variable *yylval* permet de stocker des valeurs, comme le nom d'un identificateur ou la valeur d'une constante numérique. Ces valeurs pourront être réutilisées dans l'analyseur syntaxique. La fonction *yylex()* synthétisée par *LEX* représente l'analyseur lexical du langage d'expressions régulières spécifié. Elle retourne un entier qui indique quel terminal a été accepté.

Le rapprochement expliqué entre les outils *JavaLEX* et *LEX* est pratiquement le même pour les outils *JavaCUP* et *YACC*. *JavaCUP* joue le même rôle et offre les mêmes prestations que *YACC*. Dans le fichier de spécification de *YACC*, les symboles terminaux sont déclarés avec la directive ***%token*** et les symboles non terminaux avec la directive ***%type***. Les actions dans la section de la grammaire de spécification sont codées en *C++* et comme dans le cas de *JavaCUP*, dans ces actions, les types de données du message reçus par l'analyseur syntaxique sont stockés au fur et à mesure dans des classes *C++*. Comme dans le cas de l'interface, il existe des messages qui sont corrects au niveau de la syntaxe, du fait qu'ils figurent dans la grammaire du protocole de communication, mais qui ne sont pas traités. Dans ce cas, on n'insère pas de code dans les actions de la grammaire.

Au niveau de la compilation avec *LEX* et *YACC*, nous avons utilisé respectivement les outils *FLEX* et *BISON* pour générer l'analyseur lexical et syntaxique.

5.3 *Les outils JavaLex et JavaCup*

5.3.1 L'outil JavaLex

JavaLex[1] est un générateur d'analyseurs lexicaux écrit en Java pour le langage Java. Il est développé par *Eliot Gerk* à l'université de *Princeton*. Sa maintenance est sous l'assistance de *Martin Dirichs* à l'université de *Oldenburg* en Allemagne. La manière dont *JavaLex* fonctionne est similaire à l'utilitaire *LEX* du système d'exploitation *UNIX*. Il utilise un fichier en entrée contenant la spécification d'un analyseur lexical, et il génère un code source pour l'analyseur lexical correspondant. *JavaLex* génère un code source en langage *Java* qui doit être compilé pour obtenir l'analyseur lexical.

La structure de la spécification du fichier d'entrée est organisée en trois sections séparées par deux caractères pour cent (*%%*):

[1] Consulter l'adresse URL http://www.cs.princeton.edu/~appel/modern/java/JLex/

```
Code Utilisateur
%%
Directives de JavaLex
%%
Expressions régulières
```

- **Code Utilisateur**: cette section permet à l'utilisateur d'écrire du code *Java* que l'analyseur lexical pourrait utiliser. Le code qu'on inclue dans cette section est optionnel et il doit être situé avant les deux premiers "pour cent". Il est placé textuellement au début du code Java généré. Cette section est généralement utilisée pour importer les *packages* ou pour définir des classes, des variables et d'autres types de données.

- **Directives *JavaLex***: dans cette section sont placées les directives de l'analyseur lexical. Chaque directive doit être localisée au début d'une ligne. Il est possible de définir des "*Macro*" et des états lexicaux dans cette section. Les définitions d'une "*Macro*" doivent être sur une seule ligne et sa syntaxe est:

```
<Nom_Macro> = <Définition>
```

La définition d'une *Macro* doit être une expression régulière valide. Les états sont déclarés en utilisant la directive *%state*. Par défaut *JavaLex* possède un état lexical appelé *YYINITIAL*. L'analyseur généré commence l'analyse lexical dans cet état.

JavaLex produit un grand nombre de directives dont les principales sont décrites ici:

- ***%{ … %}***: cette directive permet à l'utilisateur d'écrire du code Java qui est copié dans la classe de l'analyseur lexical. Elle est utilisée pour déclarer les variables et les méthodes internes à la classe de l'analyseur lexical.
- ***%type …***: cette directive permet de changer le type retourné par l'analyseur lexical. Par défaut ce type est ***Yytoken***, déclaré par *JavaLex*. Il est possible de changer la spécification de la classe ***Yytoken*** dans la section "**Code Utilisateur**".
- ***%cup***: cette directive permet d'activer la compatibilité avec l'outil *JavaCUP*.
- ***%eofval{ … %eofval}***: avec cette directive, il est possible de changer la valeur de retour du fin de fichier (*end-of-file*). Cette valeur doit être compatible avec le type de la méthode *Ylex.yylex()*. La valeur par défaut est *null****.
- ***%eof{ … %eof}***: cette directive est utilisée pour écrire du code Java qui va être exécuté après la rencontre de la valeur de fin de fichier (*end-of-file*) par l'analyseur lexical.

Les autres directives pouvant également être utilisées sont:

- ***%init{ … %init}***: permet d'écrire du code *Java* qui est copié dans le constructeur de la classe de l'analyseur lexical.
- ***%char, %line***: ces directives indiquent la position rencontrée de la région de texte.

- ***%eofthrow{ … %eofthrow}, % initthrow { … % initthrow }, % yylexthrow { … % yylexthrow***: ces directives sont utilisées pour déclarer des exceptions qui pourraient se produire dans leurs méthodes respectives.
- ***%class <name>, %function <name>***: ces directives permettent de changer respectivement les noms des classes de l'analyseur lexical (*Ylex*) et le nom de la fonction *yylex*.

➢ **Expressions régulières**: les règles de cette section associent des expressions régulières avec des actions écrites en code source Java. Elles ont le format suivant:

```
[<états>] <expression> {<action>}
```

- ***<états>***: cette liste d'états est optionnelle, elle spécifie sous quels états initiaux la règle peut être rencontrée. S'il n'y a aucun état pour une règle, celle ci est appliquée dans tous les états lexicaux.
- ***<expression>***: c'est l'expression régulière commune. L'ensemble des expressions régulières doit rencontrer toutes les entrées possibles, sinon, une erreur se produit. Il est possible que plus de deux règles rencontrent la même chaîne de caractères. Dans ce cas de figure, la règle qui a rencontré la plus longue chaîne de caractères est choisie, et si les longueurs des chaînes de caractères sont les mêmes, c'est la règle donnée en premier dans la spécification de *JavaLex* qui est choisie.
- ***<action>***: c'est l'action associée à la règle lexicale. Elle contient du code Java qui est copié verbalement dans la classe de l'analyseur lexical. Les transitions d'états peuvent être réalisées dans les actions. Ceci se fait avec l'appel à la fonction *yybegin(state)*, l'analyseur lexical généré reste dans l'état donné jusqu'à ce que la transition soit faite.

Dans le chapitre 5.2.3, on décrit la manière d'utiliser l'outil *JavaLex* pour générer l'analyseur lexical en se basant sur la spécification de la grammaire discutée dans le chapitre 4.3.

5.3.2 L'outil JavaCup

CUP[1], [2] a été écrit à l'origine par *Scott Hudson* en août 1995 à l'université de *Georgia Tech*. Il a ensuite été complété par *Frank Flannery* en juillet 1996 et a été récemment modifié et est maintenu par *C. Scott Ananian* en mars 1998. *CUP* est un outil qui permet de générer des analyseurs syntaxiques de type *LALR(1)*. Il a le même rôle que l'utilitaire *YACC* du système d'exploitation *UNIX*. Il est écrit en *Java* et utilise un fichier de spécification contenant du code *Java* et une grammaire permettant de générer l'analyseur syntaxique correspondant en langage *Java*. *CUP* a besoin d'unités lexicales "*tokens*" qui sont des suites de caractères en provenance d'un scanner, qui peut être un analyseur lexical généré par *JavaLex*.

L'analyseur syntaxique généré par *CUP* est composé de deux classes: *Sym.class* et *Parser.class*. *Sym.class* est composée de symboles terminaux déclarés dans la grammaire. Cette

[1] Consulter l'adresse URLhttp://www.cs.princeton.edu/~appel/modern/java/CUP/
[2] *Constructor of Useful Parser*

classe est utilisée par l'analyseur lexical pour référer les symboles. *Parser.class* contient l'analyseur syntaxique.

La structure du fichier de spécification de *CUP* est organisée en quatre sections qui doivent être écrites dans l'ordre suivant:

```
Spécification et packages à importer
Composants du Code Utilisateur
Liste des symboles (terminaux et non terminaux)
Grammaire
```

- **Spécification et packages à importer**: dans cette section optionnelle, on peut placer les *packages* et les déclarations à importer. Ils ont le même rôle que les *packages* et les déclarations à importer dans un programme Java.

- **Composants du Code Utilisateur**: cette section optionnelle permet à l'utilisateur de déclarer du code à inclure dans l'analyseur syntaxique. Elle comprend quatre parties:
  i. ***action code {: … :}***: permet d'inclure du code à l'intérieur de la classe *CUP$actions* qui est utilisée par le code introduit dans la grammaire.
  ii. ***parser code {: … :}***: cette partie est similaire à celle de "***action code***", mais le code est inclu dans la classe de l'analyseur lexical et peut être utilisé par les méthodes de cette classe.
  iii. ***init with {: … :}***: cette partie indique un code que l'analyseur lexical exécute avant de demander la première unité lexicale.
  iv. ***scan with {: … :}***: cette partie indique la manière dont l'analyseur syntaxique demande la prochaine unité lexicale du scanner. Le type de données retourné par le code du "***scan with***" doit être du même type que la classe *java_cup.runtime.token*.

- **Liste des symboles**: Dans cette section, chaque symbole terminal et non terminal qui apparaît dans la grammaire est nommé et déclaré. Le format de déclaration des symboles:

  ```
  terminal nomdelaclasse nom1, nom2, …
  non terminal nomdelaclasse nom1, nom2, …
  ```

  Les classes utilisées pour les symboles non terminaux doivent être des sous-classes de la classe *java_cup.runtime.token*.

- **Grammaire**: dans cette section, on introduit la grammaire du fichier de spécification. Avant d'introduire la grammaire, on peut déclarer le symbole non terminal avec lequel l'analyse syntaxique commence. Cette déclaration se fait de la manière suivante:

  ```
  start with non_terminal
  ```

  Si cette déclaration n'est pas introduite, l'analyseur lexical commence avec le premier symbole non terminal de la grammaire. Après cette déclaration, se trouve la grammaire, composée d'un ensemble de productions. Chaque production est composée d'une partie "**côté gauche**" contenant un symbole non terminal, suivi par le symbole "**::=**", et une partie "**côté droit**" qui

contient des séries de zéros ou plusieurs actions. La partie droite contient également des symboles terminaux et non terminaux et finit par le caractère point virgule.

Dans le chapitre 5.3.3 suivant, on va s'attacher à mieux comprendre l'utilisation de *JavaCUP* avec une grammaire, ainsi que la manière de l'unir avec le générateur d'analyseurs lexicaux *JavaLEX*.

5.3.3 Union de JavaLex et JavaCUP

Dans le chapitre 4, nous avons spécifié une grammaire pour le protocole de communication. Dans ce protocole de communication, nous avons élaboré différents types de messages que l'interface graphique et le serveur de la base de données vont utiliser pour communiquer ensemble et satisfaire les demandes de notre système *Viper*. Avec cette grammaire, et connaissant les deux outils de compilation *JavaLEX* et *JavaCUP*, nous allons associer ces deux outils avec le code source compilé de l'interface graphique, dont nous verrons une description dans le chapitre 6; afin de construire un système prêt à l'échange de messages entre l'interface graphique et la base de données.

Après la description des deux outils de compilations *JavaLEX* et *JavaCUP*, il est important de faire le point sur leur utilisationconjointe en vue de la construction des fichiers de spécification pour les deux outils. Du côté de *JavaLEX*, après la compilation de l'analyseur lexical, ce dernier prend en entrée une suite de caractères qu'il transforme en unités lexicales. Une unité lexicale est rendue par la fonction *yylex()* chaque fois que cette fonction est appelée: c'est la fonction principale de l'analyseur lexical. Du côté de *JavaCUP*, l'analyseur syntaxique a besoin d'unités lexicales pour travailler. Ces unités lexicales sont fournies par la fonction *yylex()* de l'analyseur lexical qu'on doit appeler dans la section ***scan with*** du fichier de spécification de *JavaCUP*. Le type de données des unités lexicales fournie par l'analyseur lexicale doit être du même type que celles reçues par l'analyseur syntaxique. Ce type est mis dans la directive ***%type*** de fichier de spécification de *JavaLEX*.

Les unités lexicales sont déclarées comme des symboles terminaux de la grammaire du fichier de spécification de l'analyseur syntaxique. Quand celui-ci est généré, il crée une classe de symboles (*Sym.class*) qui stocke ces unités lexicales. L'analyseur lexicale utilise les unités lexicales de *Sym.class* comme type de données retourné. L'analyseur syntaxique généré par *JavaCUP* a besoin d'une unité lexicale *Sym.EOF* de fin de fichier (*end-of-file*) généré par lui-même. On peut alors utiliser la directive ***%eofval*** du fichier de spécification de *JavaLEX* en retournant la valeur *Sym.EOF*.

5.3.4 Fichier de spécification de JavaLex.

Nous énonçons dans ce chapitre le contenu du fichier de spécification de *JavaLEX* où l'on retrouve les trois sections décrites dans le chapitre 5.2.1:

```
//////////////////////////////////////////
// Fichier de spécification pour JavaLEX
// Nom du ficher        : protocol.lex
// Auteur                     : Jilali Raki
// Date de création     : Août 1998
// date de modification : Octobre 1998
//////////////////////////////////////////

// Code Utilisateur:
// *****************
import java.lang.System;
import java_cup.runtime.Symbol;
import Sym;    /* la classe générée par l'analyseur syntaxique */

%%

// Directives de JavaLEX:
// **********************
%class Scanner /* le nom de la classe de l'analyseur lexicale sera Scanner.class
                  au lieu de Yylex.class */

%cup           /* actionner la compatibilité avec JavaCup */
%type Symbol   /* l'unité lexical retournée sera du type Symbol */
%eofval{       /* la valeur de fin de fichier retournée */
        return new Symbol(Sym.EOF);
%eofval}

%eof{          /* on affiche à l'écran une ligne pour indiquer la fin de fichier */
   System.out.println(" C'est une fin de fichier (EOF) " );
%eof}

/* définition des Macro */
letter  = [a-zA-Z]               /* un caractère majuscule ou minuscule */
digit   = [0-9]                  /* un digit entre 0 et 9 */
number  = {digit}+               /* un ou plusieurs digits */
real    = {number}("."{number})? /* un réel */
bool    = ("true")|("false")     /* une valeur booléenne "true" ou "false" */
allstr  = ({letter}|{digit})+    /* une chaîne de caractères ou digits non vide */
port    = ":"{number}            /* deux points suivi d'un numéro de port (d'URL) */
host    = {allstr}("."{allstr})* /* adresse IP ou nom d'un serveur */
filename = [^/\t\ ,()\[\]\"]     /* tout caractère exceptés les tabulations, les
                                    espaces, et les caractères utilisés dans la
                                    grammaire */
/* une chaîne de caractères acceptée par l'analyseur commence par des double guillemets
 suivi de tout caractère sauf ceux utilisés dans la grammaire */
stringname = "\""{allstr}[^,()\[\]\"]+"\""
/* le format classique d'une URL */
url      = {allstr}"://"{host}({port})?("/"({filename})+)*("/"({filename})+)?

%%

// Expressions régulières:
// ***********************
[\n\ \t\b\012]  { /* si l'analyseur rencontre des espaces, des tabulations, et des
                     retours à la ligne, il ne fait */ }

"["             {  /* Début d'un message */ return new Symbol(Sym.LBRACK); }

"HANDSHAKE_INTERFACE" { /* le type du message de bienvenue de l'interface[1] */
                    return new Symbol(Sym.HANDSHAKE_INTERFACE, new Integer(1)); }

"HANDSHAKE_SERVER"    { /* le type du message de bienvenue du serveur[2] */
                  return new Symbol(Sym.HANDSHAKE_SERVER, new Integer(2)); }

"REQUEST"             { /* le type du message "REQUEST" de l'interface */
                  return new Symbol(Sym.REQUEST, new Integer(3)); }
```

[1] ce types de message n'est reçu que par le serveur, l'interface ne le traite pas.
[2] ce type de message n'est reçu que par l'interface, le serveur ne le traite pas.

```
"LOAD"                { /* le type du message "LOAD" de l'interface */
                  return new Symbol(Sym.LOAD, new Integer(4)); }

"RESPONSE"            { /* le type du message "RESPONSE" du serveur */
                  return new Symbol(Sym.RESPONSE, new Integer(5)); }

"ERROR"               { /* le type du message "ERROR" du serveur ou l'interface */
                  return new Symbol(Sym.ERROR, new Integer(6)); }

"INTERFACE_EVENTS"    { /* le type du message "INTERFACE_EVENTS" de l'interface
                        */ return new Symbol(Sym.INTERFACE_EVENTS, new Integer(7));
                      }

"TAKE_BACK_SESSION"   { /* le type du message "TAKE_BACK_SESSION" de l'interface */
                  return new Symbol(Sym.TAKE_BACK_SESSION, new Integer(8));}

"END_SESSION"         { /* le type du message "TAKE_BACK_SESSION" de l'interface */
                  return new Symbol(Sym.END_SESSION, new Integer(9)); }

"MREP1.0"[1]       { /* version 1.0 actuelle du protocole de communication,
                  correspond au contenu des messages de bienvenue. */
                    return new Symbol(Sym.VERSION, new String(yytext())); }

"("             { /* début d'une liste de termes. */
                  return new Symbol(Sym.LPAREN); }

{number}        { /* présence de digits d'un nombre entier */
                  return new Symbol(Sym.INTEGER, new Integer(yytext())); }

{real}          { /* un nombre réel. */
                  return new Symbol(Sym.FLOAT, new Float(yytext())); }

{bool}          { /* une valeur booléenne */
                  return new Symbol(Sym.BOOLEAN, new Boolean(yytext())); }

","             { /* pour séparer les termes d'une liste */
                  return new Symbol(Sym.COMMA); }

")"             { /* fin d'une liste de terme */
                  return new Symbol(Sym.RPAREN); }

{stringname}    { /* chaine de caractère (string) */
                  String str =  new String(yytext());
                  str = str.substring(1, str.length()-1);
                  return new Symbol(Sym.STRING, str); }

{url}           { /* URL des images */
                  return new Symbol(Sym.URLS, new String(yytext())); }

"]"             { /* fin d'un message */
                  return new Symbol(Sym.RBRACK); }

.               { /* on refuse tout autre symbole ne figurant pas dans la grammaire
                  */ System.err.println("Caractère non accepté : " + yytext());
                  return new Symbol(Sym.error); }
```

L'analyseur lexicale consomme les caractères spécifiés dans les expressions régulières et génère des unités lexicales du type *Symbol* (les symboles terminaux). Pour simplifier le code source de l'interface graphique, l'analyseur lexical retourne des entiers entre 1 à 9 quand il rencontre les neuf types de messages cités dans la grammaire du chapitre 4.3. La fonction *yytext()* retourne la chaîne de caractères correspondant au symbole. Dans le cas de l'expression régulière ***{stringname}***, l'analyseur rencontre une chaîne de caractères du type *String* qui commence et finit

[1] MREP1.0 = Multimedia Retrieval Exchange Protocol version 1.0

par des doubles guillemets, et retourne comme symbole la même chaîne de caractères, mais sans les guillemets.

Pour terminer le chapitre sur l'outil *JavaLEX*, les étapes suivantes décrivent la compilation et l'utilisation de cet outil (version 1.2.3 mise à jour le 26 juin 1998):

1) Choix d'un répertoire, par exemple "*Java*", qui doit exister dans le *CLASSPATH* et où *JavaLex* doit être installé.
2) Création d'un répertoire "*Java/Jlex*" et copie du code source "*Main.java*"[1] dans ce répertoire.
3) Compilation du code source "*Main.java*" comme n'importe quel code java: *Javac Main.java* et plusieurs classes *Java* sont produites et parmi ces classes *Main.class*.
4) Enfin pour exécuter *JavaLex* avec le fichier de spécification énoncé ci-dessus, il faut lancer l'interpréteur java: *java JLex.Main protocol.lex* où *protocol.lex* est le nom du fichier de spécification.

5.3.5 Fichier de spécification de JavaCUP

Dans le fichier suivant, une partie du fichier de spécification du protocole de communication de *JavaCUP* est donnée. Cette partie montre les quatre sections discutées dans le paragraphe 5.3.2. Seulement des règles de production de quelques symboles non terminaux sont présentés. La suite des règles de production peut être facilement complété en connaissant la syntaxe des messages du protocole de communication. En consultant le fichier de spécification de *JavaLEX*, on peut reconnaître les symboles terminaux (comme *LPAREN*, *RPAREN* etc.) retournés par l'analyseur lexical.

```
//////////////////////////////////////////
// Fichier de spécification pour JavaCUP
// Nom du ficher        : Parser.cup
// Auteur               : Jilali Raki
// Date de création     : Août 1998
// date de modification : Octobre 1998
//////////////////////////////////////////

// Fichier de spécification pour le protocole de comminication version 1.0

import java_cup.runtime.*;

// les variables utilisées dans le code de la grammaire
action code {:
        int      ArrayLen, i, j;
        String   StringArray[];
        Integer  IntegerArray[];
        Integer  IntegerArray2[];
        Float    FloatArray[];
        String   StringArray2[];
        // le clsses Java du protocole
        WelcomeServerList    WelcomeServerLst    = new WelcomeServerList();
        CodeList CodeLst = new CodeList();
        CollectionList CollectionLst = new CollectionList();
        CollAlgorithme CollAlgo[];
        AlgorithmeList AlgorithmeLst = new AlgorithmeList();
        RequestList RequestLst = new RequestList();
```

[1] voir l'adresse URL http://www.cs.princeton.edu/~appel/modern/java/JLex/

```
        ResponseList ResponseLst = new ResponseList();
        ElementList ElementLst = new ElementList();
        LoadList    LoadLst    = new LoadList();
        ErrorList    ErrorLst    = new ErrorList();
        EventList    EventLst    = new EventList();
        TakeBackSessionList    TakeBackSessionLst = new TakeBackSessionList();
        EndSessionList    EndSessionLst    = new EndSessionList();
:};

// Interface to scanner generated by JLex.
parser code {:
        Parser(Scanner s, MessageProtocol MessProto) {
                super();
               this.MessProto = MessProto;
                scanner = s;
        }
        private Scanner scanner;
        public static  MessageProtocol MessProto;
:};

/* Preliminaries to set up and use the scanner.  */
scan with {: return scanner.yylex(); :};

/* Terminals (tokens returned by the scanner). */
terminal          LPAREN, RPAREN, LBRACK, RBRACK, COMMA;
terminal Integer  HANDSHAKE_INTERFACE, HANDSHAKE_SERVER, REQUEST, LOAD, RESPONSE;
terminal Integer  ERROR, INTERFACE_EVENTS, TAKE_BACK_SESSION, END_SESSION;
terminal String   STRING, VERSION, URLS;
terminal Integer  INTEGER;
terminal Float   FLOAT;
terminal Boolean  BOOLEAN;


/* Non terminals */
non terminal          empty, message, message_list, welcome_interface_list;
non terminal          request_list, load_list, response_list, error_list, events_list;
non terminal          take_back_session_list, end_session_list, welcome_server_list;

non terminal          welcome_interface_content, welcome_server_content;
non terminal          load_content, response_content, error_content, events_content;
non terminal          take_back_session_content, end_session_content, request_content;

non terminal          session_name_list, session, codes, collection, algorithme;

non terminal          error_code_message_list;
non terminal          collection_algorithme_list , element_list;
non terminal          algorithme_relevance_list;

non terminal String   url_string, www_url_str;
non terminal String   algorithme_index_list, user_name, error_message;
non terminal String   collection_name, algorithme_name, extension, session_name;

non terminal Integer  code_number, collection_number, algorithme_index_number, session_number;
non terminal Integer  algorithme_number, relevance_type, algorithme_index_part;
non terminal Integer  algorithme_id, maximum_number, element_number, element_type;

non terminal Integer  relevance_level, load_type, high_similarity, error_codes, event_code;

non terminal Float    threshold, similarity;

non terminal Boolean  session_flag, rafinement_flag;


start with message;




/* ************************************************************************************ */
/* The grammar                                                                          */
```

```
/* Message syntaxe */
message                       ::= LBRACK message_list RBRACK
                                     System.out.println("Expression bien formee");:}
;
/* different Message types */
message_list                  ::= welcome_interface_list
                                  |
                                  welcome_server_list
                                  |
                                  request_list
                                  |
                                  load_list:l
                                  |
                                  response_list
                                  |
                                  error_list
                                  |
                                  events_list
                                  |
                                  take_back_session_list:t
                                  |
                                  end_session_list
;

/* types & contents */
welcome_interface_list        ::= HANDSHAKE_INTERFACE:hi LPAREN welcome_interface_content RPAREN
                                  {: /* Put code here if interface can receive a
                                        HANDSHAKE_INTERFACE message */ :};

welcome_server_list           ::= HANDSHAKE_SERVER:hs LPAREN welcome_server_content RPAREN
                                   {: Parser.MessProto.setType(hs);
                                    WelcomeServerLst.setCodesList(CodeLst);
                                    CollectionLst.setCollAlgorithme(CollAlgo);
                                    WelcomeServerLst.setCollectionList(CollectionLst);
                                    WelcomeServerLst.setAlgorithmeList(AlgorithmeLst);
                                    Parser.MessProto.setWelcomeServerList(WelcomeServerLst);:};

request_list                  ::= REQUEST:r LPAREN request_content RPAREN
                                  {: Parser.MessProto.setType(r);
                                     RequestLst.setElementList(ElementLst);
                                     Parser.MessProto.setRequestList(RequestLst);:};

load_list                     ::= LOAD:l LPAREN load_content RPAREN
                                  {: Parser.MessProto.setType(l);
                                     Parser.MessProto.setLoadList(LoadLst);:};

response_list                 ::= RESPONSE:r LPAREN response_content RPAREN
                                  {: Parser.MessProto.setType(r);
                                     ResponseLst.setElementList(ElementLst);
                                     Parser.MessProto.setResponseList(ResponseLst);:};

error_list                    ::= ERROR:e LPAREN error_content RPAREN
                                  {: Parser.MessProto.setType(e);
                                     Parser.MessProto.setErrorList(ErrorLst); :};

events_list                   ::= INTERFACE_EVENTS:ie LPAREN events_content RPAREN
                                  {: Parser.MessProto.setType(ie);
                                     Parser.MessProto.setEventList(EventLst); :};

take_back_session_list        ::= TAKE_BACK_SESSION:tb LPAREN take_back_session_content RPAREN
                                  {: Parser.MessProto.setType(tb);
                                  Parser.MessProto.setTakeBackSessionList(TakeBackSessionLst); :};

end_session_list              ::= END_SESSION:es LPAREN end_session_content RPAREN
                                  {: Parser.MessProto.setType(es);
                                     Parser.MessProto.setEndSessionList(EndSessionLst);:};

/* content of each type */
/* -------------------- */

/* welcome interface content */
/* version = "MREP1.0" */
```

```
welcome_interface_content     ::= VERSION:v user_name:u
                                  {: /* Put code here if interface can receive a
                                         HANDSHAKE_INTERFACE message */ :};

user_name                     ::= STRING:s;


/* welcome server content */
welcome_server_content        ::= VERSION:v session codes collection algorithme
                              {: WelcomeServerLst.setVersion(v); :};

…
```

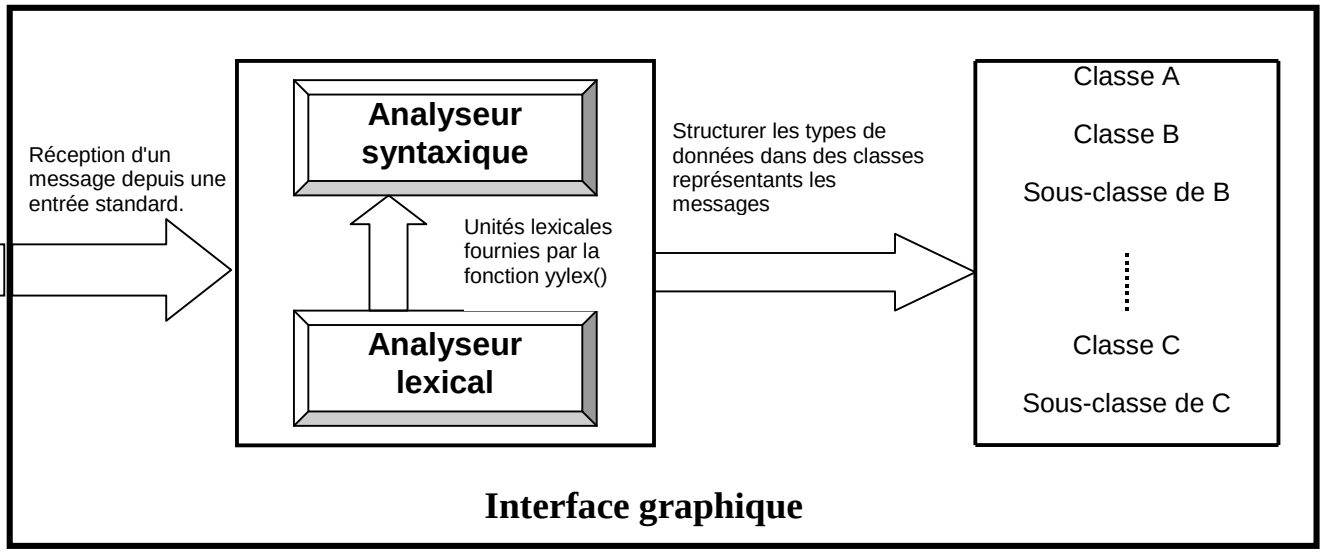

Figure 12

Le fichier de spécification de *JavaCUP* doit s'occuper du côté syntaxique de la grammaire du protocole de communication. Dans cette spécification, nous devons définir les actions à exécuter lorsqu'un message est syntaxiquement correct. Ces actions sont définies en code *Java* dans la section de la grammaire du fichier de spécification. En effet, l'interface graphique n'est pas uniquement implantée avec une seule classe *Java* comme nous allons voir dans le chapitre 6, mais il existe un ensemble de classes Java qui représentent les différents messages. Les contenus des messages sont aussi des sous-classes des classes représentants les messages. Dans les actions de la grammaire, l'analyseur syntaxique remplit au fur et à mesure ces classes en analysant le message venant du serveur de la base de données comme nous le montre la figure 12. Si le message est syntaxiquement correcte, toutes les classes et sous-classes représentants les messages seront affectées. Dans le cas contraire un message d'erreur est généré. La fonction *yylex()* permet de fournir à l'analyseur syntaxique les unités lexicales en provenance de l'analyseur lexical.

Comme dans le paragraphe 5.3.4, les étapes qui suivent décrivent la manière d'utiliser et de compiler avec *JavaCUP* de la version 0.10g de mars 1998:

1) Chargement d'un fichier de type *tar* (*java_cup_v10g.tar* pour la version 0.10g) depuis le site Internet référencé au chapitre 5.2.2.
2) Extraction des fichiers de l'archive téléchargé dans le répertoire créé pour l'outil *JavaLEX* (par exemple *Java/Jlex*) et qui doit exister dans le *CLASSPATH*.
3) Compilation de tous les fichiers Java extraits dans le point 2 dans le répertoire de *JavaLEX* (*Java/Jlex/java_cup*):

```
java java_cup/*.java java_cup/runtime/*.java
```

4) Enfin pour exécuter *JavaCUP* avec le fichier de spécification, il faut lancer l'interpréteur Java:

```
java java_cup.Main -parser Parser -symbols Sym <Parser.cup
```

où *Parser.cup* est le nom du fichier de spécification, *-parser* une option pour changer le nom de la classe de l'analyseur syntaxique et *–symbols* une option pour changer le nom de la classe des symboles terminaux générés.

6 L'interface graphique

6.1 *Introduction à l'interface de l'utilisateur*

L'interface graphique est l'outil qui va permettre à un utilisateur de s'identifier et de lancer les requêtes de recherche d'images similaires. Son rôle est de recevoir les commandes de l'utilisateur à travers des menus, boutons, champs de texte etc. pour les envoyer au serveur par le protocole de communication. Son rôle est également d'intercepter des résultats et des données du serveur par le protocole de communication pour les présenter graphiquement à l'utilisateur. L'interface est construite à l'aide d'une *Applet Java* version 1.0.2.

Pour construire l'interface graphique, il est important d'étudier le type de message qu'elle peut recevoir, ainsi que l'ordre des commandes lors de l'envoi du premier message de bienvenue avec le serveur. La figure 13, montre la séquence d'envoi des messages et les types de message que l'interface peut échanger avec le serveur.

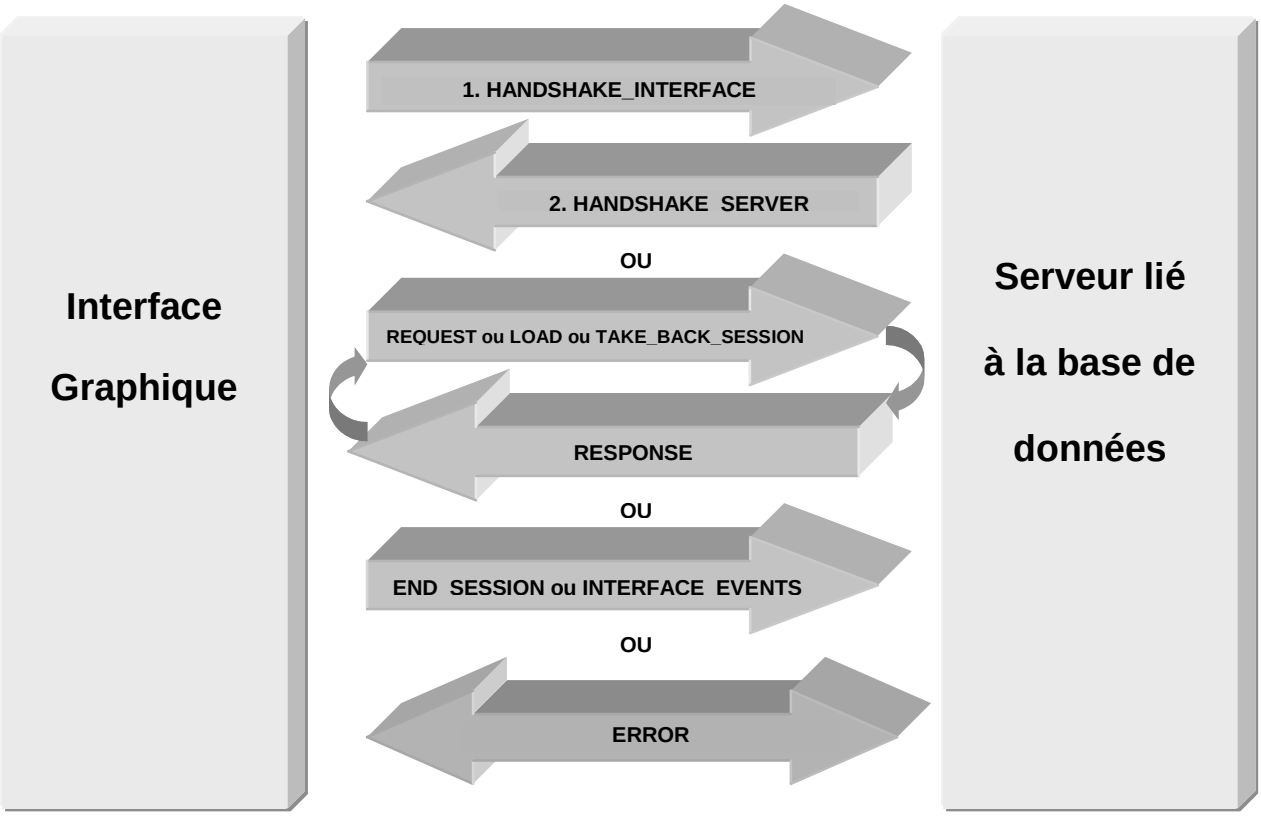

Figure 13

Figure 14: Fenêtre d'identification de l'utilisateur

Au premier contact de l'interface avec le serveur de la base de données, l'interface envoie en premier son message de bienvenue en se connectant au serveur. En deuxième étape, le serveur répond par son message de bienvenue à l'interface. Les messages suivants peuvent être échangés dans n'importe quel ordre.

L'utilisateur peut accéder à l'interface[1] à travers un navigateur sur Internet *(Netscape ou Internet Explorer*). Une première fenêtre de bienvenue à l'utilisateur, comme nous le montre la figure 14, apparaît pour lui permettre de saisir son nom l'identifiant. L'identification de l'utilisateur permet au serveur de construire pour chacun d'eux un profil des événements réalisés sur l'interface. La partie inférieure de la fenêtre de la figure 14 montre une section pour afficher les messages d'erreur et les messages du déroulement des actions. En cliquant sur le bouton "*Start a session*", l'utilisateur peut accéder sur une deuxième interface de la même *Applet*, qui lui permet de commencer une session de recherche d'images. Cette deuxième interface est divisée en quatre sections comme nous allons le voir dans le chapitre 6.3.

[1] l'URL de l'interface: http://cuiwww.unige.ch/~viper/demo/

6.2 *description de l'utilisation de l'interface par un diagramme*

Avant de décrire les différentes sections de l'utilisation de l'interface graphique, nous pouvons mieux comprendre les différentes utilisations de l'interface avec le diagramme de la figure 15. En commençant par la fenêtre de bienvenue permettant d'identifier l'utilisateur, le diagramme montre le parcourt de l'interface jusqu'au lancement des requêtes au serveur de la base de données.

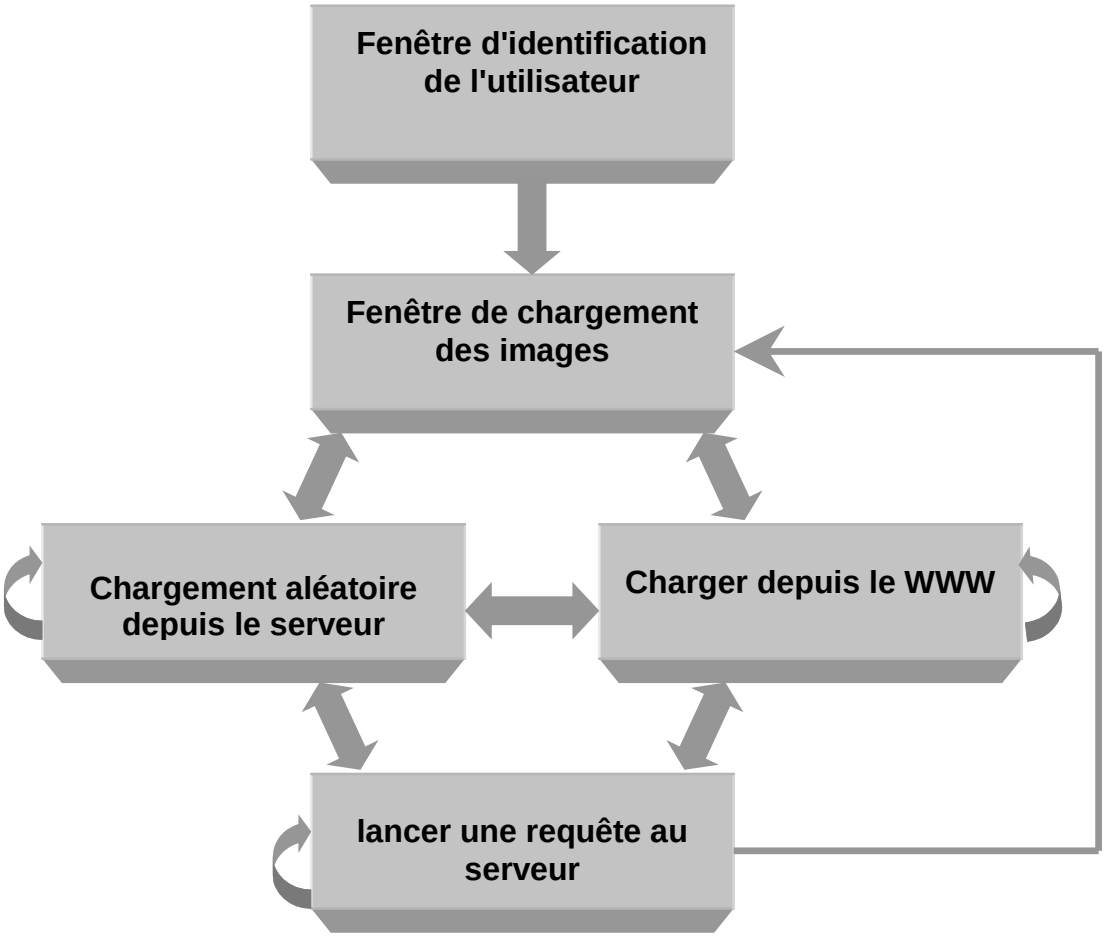

Figure 15: Diagramme d'utilisation de l'interface graphique

6.3 *Les différentes sections de l'interface*

Après l'identification de l'utilisateur, la fenêtre de bienvenue de la figure 14 est remplacée par la fenêtre de la figure 16. Cette fenêtre montre trois sections nommées "*Enable section*", "*Load section*" et "*display section*". En plus de ces sections, il existe une autre section nommée "*Request section*" qui est invisible sur l'interface mais qui est activée dès qu'une ou plusieurs images sont chargées dans la section d'affichage "*display section*". Nous allons décrire les

différentes sections de l'interface en expliquant leurs fonctionnalités dans les paragraphes qui suivent.

Figure 16

6.3.1 La section "*Load*"

Figure 17

Cette section permet à l'utilisateur de charger une ou plusieurs images sur l'interface pour lancer des requêtes de recherche d'images similaires. Le type de chargement des images peut être aléatoire ou fixe grâce aux cases à cocher (*Checkbox*) "*Load type:*" avec les deux choix "*Random*" et "*From WWW*". Dans le cas d'un chargement aléatoire comme le montre la figure 16, l'utilisateur demande un certain nombre d'images dans le champ "*Elements number*". Ensuite

l'interface demande par un message du type "LOAD" au serveur de la base de données le nombre d'images voulu.

Dans le cas d'un chargement fixe comme nous le montre la figure 18, l'utilisateur saisit l'*URL* de l'image demandé et l'interface la charge depuis le *World Wide Web*. En même temps, un message du type "*LOAD*" est envoyé au serveur pour l'informer de la présence de l'image sur l'interface afin d'en extraire les caractéristiques et les stocker dans la base de données.

Figure 18

Lorsque l'un des deux types de chargement est choisi, l'utilisateur peut appuyer sur le bouton "*Ok for Load*" pour charger les images sur l'interface. Ensuite cette section est remplacée par une autre section nommée "*Request section*" comme le montre la Figure 19.

6.3.2 La section "*Enable*"

Cette section est désactivée au départ quand l'interface est chargée, comme on peut le constater sur la figure 15. Elle est active au moment où l'utilisateur charge des images sur l'interface et quand "*Load section*" est remplacée par "*Request section*". Dans ce cas, "*Enable section*" permet à l'utilisateur de basculer du "*Load section*" au "*Request section*" avec les deux cases à cocher "*Enable load section*" et "*Enable request section*" comme on peut le voir sur la Figure 19. Ce choix permet à l'utilisateur de recharger d'autres images sur l'interface ou de lancer une requête de recherche d'images similaires.

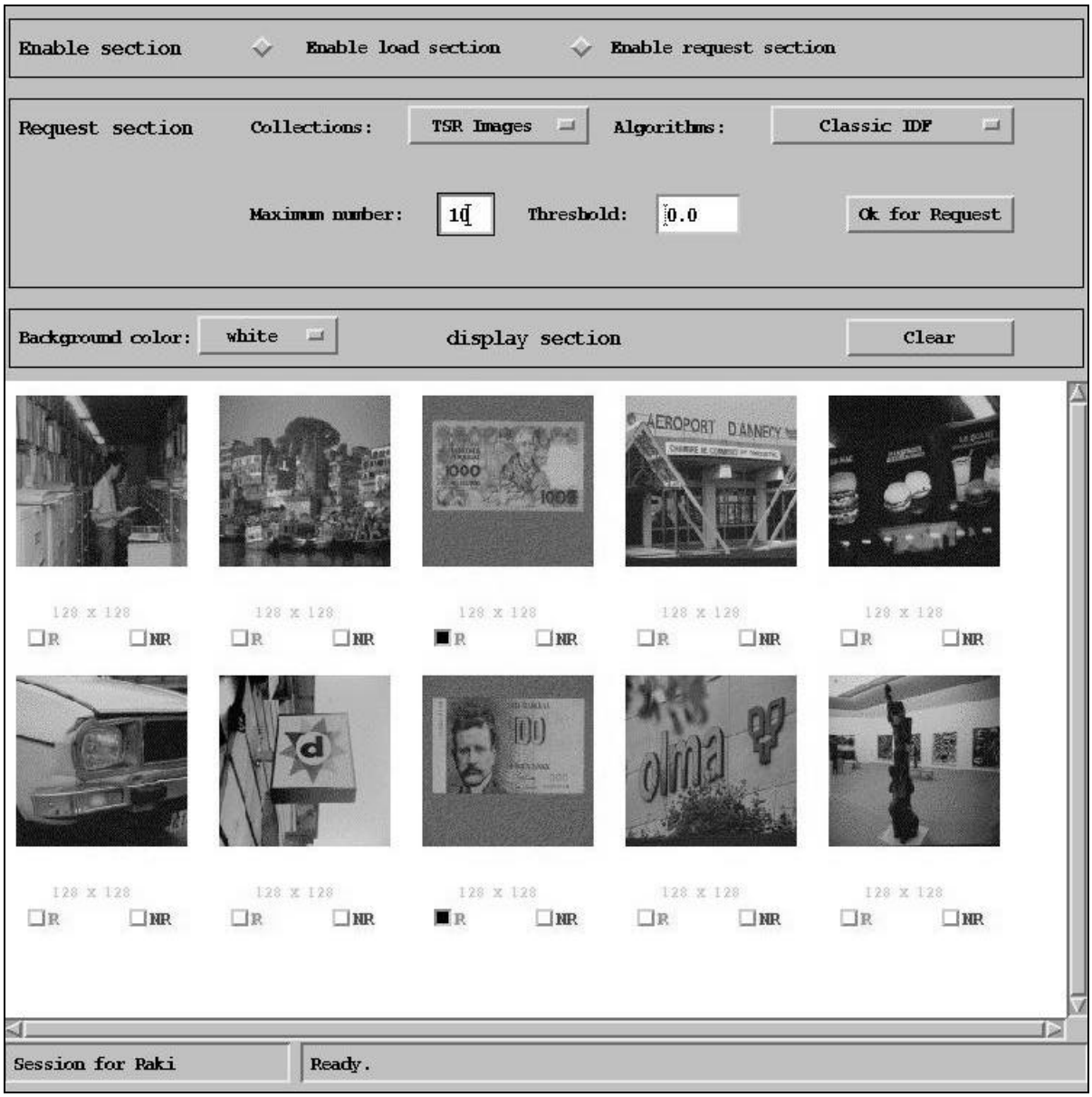

Figure 19

6.3.3 La section "*Request*"

C'est la section qui permet à l'utilisateur de lancer une recherche des images similaires. Avant de lancer une requête, l'utilisateur doit impérativement cocher l'une des deux cases d'au moins une image pour préciser laquelle des images est significative ou non dans la section "*display section*" (voir la Figure 20). Si aucune case n'est cochée, la signifiance de l'image est neutre. Ensuite l'utilisateur peut choisir dans la section "*Request section*" la **collection** d'images, **l'algorithme de recherche** à utiliser, le **nombre d'images**, et enfin le **seuil** de similarité voulu:

- **Collection d'images**[1]: c'est une des collections d'images à envoyer à l'interface dans le type de message "*HANDSHAKE_SERVER*" du serveur. Ces collections sont disponibles dans la base de données. La figure 20 montre qu'il n'y a qu'une collection de la TSR[2] qui est disponible actuellement.

- **Algorithmes**: ce sont les algorithmes de recherche disponibles pour la collection d'images comme on peut le voir à la Figure 20. Ces algorithmes sont également reçus par l'interface dans le message de bienvenue du serveur.

- **Nombre d'image**: c'est le champ de texte "*Maximum number*" où l'utilisateur peut saisir le nombre maximum d'images voulu pour la requête. Sa valeur par défaut est de 5 images.

- **Seuil**: c'est le champ "*Threshold*" de la Figure 20. C'est un champ alternatif à celui du nombre d'images. L'utilisateur peut décider que les images similaires aient un seuil de similarité correspondant au seuil saisi dans ce champ si le nombre d'images n'est pas précisé. Dans le cas où le seuil et le nombre d'images sont déclarés ensemble c'est le nombre d'images qui est pris en considération.

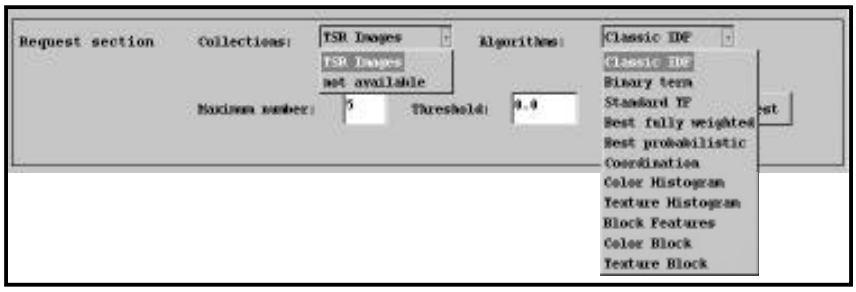

Figure 20

A la fin, l'utilisateur peut cliquer sur le bouton "*Ok for request*" pour lancer la requête et attendre les résultats.

6.3.4 La section "Display"

C'est la section réservée à l'affichage des images comme on peut le voir sur la figure 19. Il y a deux genres d'affichage: l'affichage des images que l'utilisateur charge avant de lancer une requête et l'affichage des images similaires suite à une requête. La figure 19 montre un affichage des images chargées aléatoirement depuis le serveur. On remarque que les tailles des images sont affichées en dessous de leurs images respectives. Chaque image possède deux cases à cocher pour choisir le niveau de signification de chacune. La Figure 21 montre quinze images affichées suite à une requête lancée pour trouver des images similaires. On remarque que dans cet affichage, le niveau de similarité de chaque image par rapport à l'image de référence de la requête est affiché

[1] une collection d'images disponible actuellement est celle des images de la Télévision Suisse Romande.
[2] Télévision Suisse Romande

également en dessous de chacune d'elle. Dans cette section d'affichage, l'utilisateur peut changer la couleur de l'arrière plan comme le montre la figure 21.

Le bouton "*Clear*" permet d'effacer l'affichage et revenir à l'interface de la figure 16. Ce bouton est utile dans la mesure où l'utilisateur charge des images sur l'interface avec la section "*Load section*", et additionne les images aux images déjà existantes sur l'affichage. Ce bouton permet donc d'initialiser l'affichage et de recommencer avec d'autres images.

Figure 21

Figure 22

6.3.5 La section des messages.

Sur la fenêtre initiale de bienvenue de la figure 14 ou celle de la figure 16, on constate en bas des fenêtres une section réservée au messages destinés à informer l'utilisateur sur le déroulement des événements et sur les erreurs qui peuvent se produire dans le protocole de communication, que ce soit pour l'interface elle-même ou du côté du serveur.

6.4 *L'implantation de l'interface*

L'interface graphique est implantée en langage Java en utilisant la version 1.0.2 afin d'être fonctionnelle sur les navigateurs Internet des versions 3 ou supérieur de *Netscape* et *Internet Explorer*.

L'implantation de l'interface doit assumer trois tâches essentielles:

1. Connexion avec le serveur pour envoyer et recevoir des messages du protocole de communication.

2. Construction de l'interface graphique pour l'utilisateur et gestion des événements.

3. Analyse et traitement des messages qui proviennent du serveur.

6.4.1 Connexion avec le serveur

La tâche de connexion au serveur de la base de données est assurée par une classe ***Client.java***. Cette classe comprend quatre méthodes:

1. ***Connect(String host)***: cette méthode fait une connexion du type *socket* au serveur connecté à la base de données qui attend une connexion appelante sur un numéro de port. Elle retourne un entier positif si la connexion s'est bien passé et crée un canal de sortie pour envoyer des données au serveur et un canal d'entrée pour recevoir les données envoyées par le serveur.

2. ***Send(byte[] Bytes, int NbBytes)***: cette méthode envoie un tableau *Bytes* de type *byte* avec un nombre d'octets *NBBytes* au serveur à travers le canal de sortie de type *socket*. Le choix de transmettre des *bytes* au lieu des chaînes de caractères est justifié par la différence de représentation de codage des caractères qui existe entre les deux différents langages *Java*, et *C++* (16 *bits* en *Java* et 8 *bits* en *C++*).

3. ***Receive()***: cette méthode reçoit des octets correspondants à un message du serveur par le canal de transmission d'entrée et retourne la chaîne de caractères formée de ces octets. La lecture des octets se fait caractère par caractère jusqu'à la rencontre du dernier caractère du message ']'.

4. ***Close()***: ferme la connexion de type *socket* avec le serveur.

Pour communiquer des messages avec le serveur, la classe *Interface.java* utilise ces méthodes, qui utilisent la classe *Client.java*:

- ***SendMessage(String str)***: cette méthode se connecte au serveur et envoie le message *str*.

- ***ReadMessage()***: cette méthode lit un message en provenance du serveur et l'expose directement à la classe de l'analyseur lexicale *Parser.java*. L'entrée de la connexion *socket* avec le serveur est dirigée vers l'entrée standard de l'analyseur lexical. Les deux déclarations suivantes permettent d'utiliser les outils *JavaLex* et *JavaCup*.
```
Parser parser = new Parser(new Scanner(in), MessProto);
Symbol parse_tree = parser.parse();
```

- ***SendReadMessage(String message)***: utilise les deux méthodes décrites ci-dessus pour envoyer un message et recevoir un autre message.

6.4.2 Construction de l'interface graphique

La construction de l'interface consiste à placer les différents composants graphiques sur une *Applet Java* et à gérer les événements captés par ces composants. Elle est réalisée par la classe *Interface.java* qui est le corps principal du code source de l'interface. Nous décrivons les méthodes importantes en permettant de construire l'interface graphique:

- ***addComponent(Panel p, Component c, int x, int y, int w, int h, Color fgcolor, Color bgcolor, String font, int size, boolean PutColor)***:
  cette méthode place un composant *c*, qui peut être un bouton, un menu, etc. dans un conteneur (panneau) *p* en position (*x,y*) avec une largeur *w* et une hauteur *h*. La fonte du texte et la taille sont définies par les paramètres *font* et *size*. Si la valeur booléenne *PutColor* vaut "*true*", les paramètres *fgcolor* et *gbcolor* sont pris en compte pour changer l'arrière plan et l'avant plan de la couleur du composant.

- ***DrawWelcomeInterface()***: permet de dessiner l'interface de bienvenue pour l'identification de l'utilisateur.

- ***RemoveWelcomeComponents()***: Supprime les composants de l'interface de bienvenue. Cette méthode est appelée lorsque l'utilisateur est identifié pour charger le deuxième interface.

- ***DrawInterface()***: c'est la méthode qui dessine la deuxième interface.

- ***DrawEnablePanel()***: cette méthode dessine la boîte de dialogue de la section "*Enable*". Elle est appelée par la méthode *DrawInterface()*.

- ***DrawLoadPanel()***:cette méthode dessine la boîte de dialogue de la section "*Load*". Elle est appelée par la méthode *DrawInterface()*.

- ***DrawRequestPanel(String[] ArrayStr1, String[] ArrayStr2)***: cette méthode dessine le conteneur de la section "*Request*". Les tableaux de chaînes de caractères, *ArrayStr1* et *ArrayStr2*, correspondent aux noms de collections et aux algorithmes à placer dans les menus. Cette méthode est appelée par la méthode *DrawInterface()*.

- ***DrawHeadPanel()***: cette méthode dessine l'en-tête de section "*display*" où se trouve le menu de choix des couleurs de l'arrière plan et le bouton "*Clear*".

- ***DrawElementsPanel()***: cette méthode dessine la partie réservée à l'affichage des images.

- ***action(Event e, Object arg)***: cette méthode permet de gérer tous les événements survenus sur les composants de l'interface. A chaque événement (cliquer sur un bouton, choisir un élément dans un menu, cocher une case à choix etc.), une action est reproduite comme par exemple envoyer un message au serveur ou changer l'arrière plan de l'affichage etc. Le paramètre *e* correspond à l'événement, et *arg* correspond à l'objet provoquant cet événement.

6.4.3 Traitement des messages

Les outils de compilations, *JavaLex* et *JavaCup*, génèrent respectivement l'analyseur lexical et syntaxique pour la grammaire du protocole de communication. Ces analyseurs sont associés au code source de l'interface afin de vérifier la forme correcte des messages envoyés par le serveur à l'interface. Pour stocker les différents termes du message envoyés par le serveur, des classes et des sous-classes sont créées et sont appelées dans le fichier de spécification de l'analyseur lexical. Après le stockage du message, l'interface appelle ces classes et modifie l'*Applet* graphique en fonction du message. Les classes et les sous-classes sont formées de méthodes qui stockent l'information (***set****NomMethode(…))* et des méthodes qui retournent l'information (***get****NomMethode*()) stockée. Nous allons décrire les différentes classes mais en indiquant seulement les méthodes qui stockent l'information puisque les autres méthodes jouent le rôle de récupérer cette information:

- ***MessageProtocol.java***: cette classe stocke le type de message du protocole de communication. Ces types sont expliqués dans le chapitre 4.2. La classe contient cinq méthodes (dix méthodes en comptant les méthodes ***get****NomMethode*):
  1. ***setType(Integer type)***: stocke l'entier correspondant au type du message.
  2. ***setWelcomeServerList(WelcomeServerList welcome_server_list)***: stocke le message *"HANDSHAKE_SERVER"* dans la sous-classe du type *WelcomeServerList*.
  3. ***setResponseList(ResponseList response_list)***: stocke le message "**RESPONSE**" dans la sous-classe du type *ResponseList*.
  4. ***setErrorList(ErrorList error_list)***: stocke le message "ERROR" dans la sous-classe du type *ErrorList*.
  5. ***setEndSessionList(EndSessionList end_session_list)***: stocke le message "END_SESSION" dans la sous-classe du type *EndSessionList*.

- ***WelcomeServerList.java***: c'est la sous-classe appelée par la classe-mère *MessageProtocol.java*. Elle contient six méthodes:
  1. ***setVersion(String version)***: stocke la version du protocole de communication (*MREP1.0* étant la version actuelle).
  2. ***setSessionNumber(Integer session_number)***: stocke le nombre de sessions de l'utilisateur.
  3. ***setSessionName(String[] session_name)***: stocke la liste des noms des sessions.
  4. ***setCodesList(CodeList code_list)***: stocke les codes des messages d'erreurs dans une sous-classe du type *CodeList*.

5. ***setCollectionList(CollectionList collection_list)***: stocke les collections d'images dans une sous-classe du type *CollectionList*.
6. ***setAlgorithmeList(AlgorithmeList algorithme_list)***: stocke les algorithmes des collections dans une sous-classe du type *AlgorithmeList*.

- ***CodeList.java***: c'est la sous-classe appelée par la classe-mère *WelcomeServerList.java*. Elle permet de stocker les codes des messages d'erreurs. Elle contient trois méthodes.
  1. ***setCodesNumber(Integer code_number)***: stocke le nombre de codes d'erreurs.
  2. ***setErrorCodes(Integer[] error_codes)***: stocke la liste des entiers correspondants aux codes d'erreurs.
  3. ***setErrorMessages(String[] error_messages)***: stocke la liste des messages d'erreurs correspondants aux codes.

- ***CollectionList.java***: C'est la sous-classe appelée par la classe-mère *WelcomeServerList.java*. Elle permet de stocker les collections d'images disponibles dans la base de données du serveur . Elle contient trois méthodes.
  1. ***setCollectionNumber(Integer collection_number)***: stocke le nombre de collections.
  2. ***setCollAlgorithme(CollAlgorithme[] coll_algorithme)***: stocke les algorithmes de recherche qu'on peut appliquer à chaque collection. Les algorithmes sont stockés dans un tableau de sous-classes du type *CollAlgorithme*.
  3. ***setCollectionName(String[] collection_name)***: stocke la liste des noms des collections.

- ***CollAlgorithme.java***: C'est la sous-classe appelée par la classe-mère *CollectionList.java*. Elle permet de stocker les algorithmes d'une collection. Elle contient deux méthodes.
  1. ***setAlgorithmeNumber(Integer algorithme_number)***: stocke le nombre d'algorithmes.
  2. ***setAlgorithmeIndex(Integer[] algorithme_index)***: stocke la liste des index[1] des algorithmes.

- ***AlgorithmeList.java***: C'est la sous-classe appelée par la classe-mère *WelcomeServerList.java*. Elle permet de stocker les algorithmes de recherche disponibles dans la base de données. Elle contient trois méthodes.
  1. ***setAlgorithmeNumber(Integer algorithme_number)***: stocke le nombres d'algorithmes.
  2. ***setAlgorithmeName(String[] algorithme_name)***: stocke la liste des noms d'algorithmes.
  3. ***setRelevanceType(Integer[] relevance_type)***: stocke le type de signification accepté par l'algorithme.

- ***ResponseList.java***: c'est la sous-classe appelée par la classe-mère *MessageProtocol.java*. Le message de cette classe peut être une réponse au message "REQUEST", au message "LOAD" ou au message "TAKE_BACK_SESSION". La classe contient six méthodes:
  1. ***setHighSimilarity(Integer high_similarity)***: stocke le niveau de similarité maximale observé entre une image de la requête et les images trouvées. Ce champ est utile quand le message est une réponse au message "REQUEST" de l'interface.
  2. ***setElementList(ElementList element_list)***: stocke la liste des éléments (images) correspondants à une requête dans une sous-classe du type *ElementList*.

[1] Les algorithmes sont indexés dans la classe *AlgorithmeList.java*. On utilise dans cette classe les index au lieu des noms.

- ***ElementList.java***: C'est la sous-classe appelée par la classe-mère *ResponseList.java*. Elle permet de stocker les éléments. Elle contient six méthodes.
  1. ***setElementNumber(Integer element_number)***: stocke le nombre d'éléments.
  2. ***setElementType(Integer[] element_type)***: stocke le type des éléments. Ce champ est utile si le protocole de communication est étendu pour traiter d'autres éléments multimédia.
  3. ***setUrls(String[] Urls)***: stocke l'adresse *URL* de chaque élément.
  4. ***setRelevanceLevel(Integer[] relevance_level)***: stocke le niveau de signifiance de chaque élément. Ce champ est neutre dans ce message, mais il est utilisé dans le message "REQUEST" envoyé par l'interface au serveur après que l'utilisateur ait choisi le type de signifiance des images sur l'interface.
  5. ***setSimilariry(Float[] similariry)***: stocke le niveau de similarité par rapport à l'image de la requête.
  6. ***setExtension(String[] extension)***: stocke une extension concernant chaque élément. Ce champ existe dans le protocole de communication mais il n'est pas encore utilisé.

- ***ErrorList.java***: c'est la sous-classe appelée par la classe-mère *MessageProtocol.java*. Elle contient une méthodes:
  1. ***setErrorCode(Integer error_codes)***: stocke l'entier correspondant au code de l'erreur.

Dans la classe *Interface.java*, ces classes sont instanciées et une classe de type *MessageProtocol* est passée en argument à l'analyseur lexical pour analyser et stocker le message dans cette classe.

Dans ce chapitre, nous avons décrit les éléments essentiels qui ont permis d'implanter l'interface graphique, ainsi que la communication avec le serveur lié à la base de données. Dans le chapitre 7, nous allons examiner la part du serveur afin de décrire son implantation, et la manière dont les caractéristiques des images sont stockées dans la structure du fichier inversé.

7 Implantation de la base de données

7.1 *Structure de fichier inversé*

Dans le domaine de recherche de textes, un index est un mécanisme qui permet de localiser les textes contenant un terme. Il existe beaucoup de moyens d'archiver les textes. Dans un environnement statique contenant une quantité importante de textes, les trois structures qui conviennent le plus sont: *inverted files*; *signatures files*; et les *bitmaps*. Les structures de **fichier inversé** (*inverted file*) et "*bitmap*" ont besoin d'une liste de termes qui apparaissent dans la base de données alors que dans la méthode "*signatures file*" ce n'est pas le cas. Dans ce chapitre, nous n'allons parler que de la structure du **fichier inversé** utilisée pour stocker les caractéristiques des images pour le système *Viper*.

Pour comprendre la signification d'un index du fichier inversé, supposons un ensemble de documents contenant du texte. Un index du fichier inversé contient, pour chaque mot du texte, une entrée qui stocke une liste de pointeurs à toutes les occurrences de ce mot dans les documents. Chaque pointeur est en fait le numéro de documents où le mot apparaît. C'est probablement la méthode d'indexage la plus fréquente que nous pouvons rencontrer dans la partie ***index*** des livres.

La table 1 donne un exemple où chaque document correspond à une ligne de texte en anglais. Le fichier inversé généré pour ce texte est montré dans la table 2 où tous les mots du texte sont indexés. Une colonne des documents indique les numéros des documents où le mot apparaît. Pour faire une requête pour un seul mot par exemple, la réponse à la requête se fait en analysant son entrée du fichier inversé et en recherchant chaque document qui contient le mot. On peut effectuer une requête en combinant les mots avec les opérateurs logiques *AND* et *OR*. Dans ce cas, l'intersection et la réunion des entrées du fichier inversé sont formées pour trouver le résultat. Comme par exemple le cas où on veut trouver les documents qui contiennent "*some AND hot*" dans la collection de la table 1. La réunion des entrées respectives des deux mots (4,5) et (1,4) donne la liste des documents qui ont ces deux mots en commun. Dans ce cas le résultat est un seul document (4).

Table 1

Document	Texte
1	*Pease porridge hot, pease porridge cold,*
2	*Pease porridge in the pot,*
3	*Nine days old,*
4	*Some like it hot, some like it cold,*
5	*Some like it in the pot,*
6	*Nine days old,*

Table 2

Nombre	Mots	Documents
1	cold	1, 4
2	days	3, 6
3	hot	1, 4
4	in	2, 5
5	it	4, 5
6	like	4, 5
7	nine	3, 6
8	old	3, 6
9	pease	1, 2
10	porridge	1, 2
11	pot	2, 5
12	some	4, 5
13	the	2, 5

7.2 *Stockage des caractéristiques d'images*

Dans le chapitre 3, nous avons expliqué les types de caractéristiques que nous avons extraites des images. Chaque image peut avoir un nombre de caractéristiques de l'ordre de $O(10^3)$. En augmentant le nombre d'images, le nombre de caractéristiques possibles dans la collection peut augmenter considérablement. Une structure du fichier inversé a la possibilité de stocker un très grand nombre de données et peut servir de base de données pour ces caractéristiques. Notons surtout que la finalité de ces caractéristiques est d'être éparses. Une image peut avoir $O(10^3)$ caractéristiques, mais c'est $O(10^3)$ sur $O(10^5)$ caractéristiques possibles.

7.2.1 Stockage préalable dans des fichiers

Avant de stocker les caractéristiques des images dans la structure de données du fichier inversé, il est important de connaître la manière dont ces caractéristiques sont organisées et stockées dans les différents fichiers. Les caractéristiques extraites des images sont indexées et stockées préalablement dans un fichier. La table 3 montre un exemple du stockage dans le fichier. Ensuite, pour chaque image nous avons un fichier qui contient toutes ses caractéristiques avec la fréquence d'une caractéristique dans cette image. La table 4 montre un exemple de structure d'un fichier associé à une image. La fréquence de chaque caractéristique dans l'image est comprise entre 0 et 1.

Table 3: Liste de toutes les caractéristiques

caractéristique	description des caractéristiques
0	blocs rouges dans le coin supérieur gauche
1	blocs rouges dans le coin supérieur droit
…	…

Table 4: Liste des caractéristiques dans une image

caractéristique	fréquence de la caractéristique dans l'image
5	0.008
17	0.091
…	…

7.2.2 Correspondance entre URL et fichier

Dans le protocole de communication entre l'interface graphique et le serveur de la base de données, les images sont identifiées dans les messages par leurs *URLs*. En plus des deux fichiers de caractéristiques d'images, un troisième fichier est construit. Il comporte une liste de correspondances entre les *URLs* des images et leurs fichiers de caractéristiques. La table 5 montre un exemple d'un tel fichier.

Table 5: Liste des URLs des images et de leurs fichiers de caractéristiques

image	URL de l'image	nom du fichier des caractéristiques de l'image
0	*http://cuiwww.unige.ch/~squire/image0*	image0.fts
1	*http://cuiwww.unige.ch/~squire/image1*	image1.fts
…	…	…

7.2.3 Archivage dans le fichier inversé

A partir des différentes informations extraites des images et de leurs caractéristiques, nous pouvons construire la structure de fichier inversé correspondante. Cette structure comporte toutes les images avec leurs caractéristiques. Pour chaque caractéristique, la structure comporte la fréquence de la caractéristique dans toute la collection d'images notée ***cf***, ainsi que la liste des images qui contient cette caractéristique avec leurs fréquences notées ***df*** de la table 4. Il ne faut pas confondre entre les deux fréquences *cf*, et *df*; *cf* est la fréquence d'une caractéristiques par rapport à la globalité de la collection d'images, alors que *df* est la fréquence d'une caractéristique dans une image.

Un programme implanté en langage *C++* permet de stocker les caractéristiques dans la structure de fichier inversé à partir des différents fichiers des caractéristiques cités dans l'exemple de la table 4 du chapitre 7.2.1. En lisant tous les fichiers des caractéristiques, nous pouvons connaître le nombre d'images et le nombre de caractéristiques total. Ces valeurs ne sont pas connues à l'avance dans le programme. Le programme comprend deux structures de données: `_FEATURE_DATA` et `_IFS`:

```
 typedef struct _FEATURE_DATA {
  int id;
  unsigned char frequency;
} FEATURE_DATA;

typedef struct _IFS {
  int url_freq_number ;
  FEATURE_DATA* url_freq;
}IFS;
```

La structure `_FEATURE_DATA` permet de stocker l'ensemble des identificateurs `id` des caractéristiques avec les fréquences `frequency` des images lues dans leurs fichiers correspondants. Le nombre d'images contenant une fréquence donnée `url_freq_number`, la liste `url_freq` des identificateurs des images, et leurs fréquences dans l'image sont stockés dans la structure `_IFS`. Quand cette structure est complétée avec toutes les caractéristiques existantes, elle est copiée dans un fichier qui est utilisée par le serveur pour rechercher des images similaires à des requêtes qui peuvent contenir plusieurs images.

Pour faciliter l'accès à la structure de fichier inversé, un autre fichier est construit. Dans ce fichier, l'offset de chaque caractéristique dans le fichier inversé est stocké. La table 6 présente une partie de la structure *du fichier inversé* de la collection utilisée de 500 images de Télévision Suisse Romande. De ces images, le nombre total de caractéristiques possibles est 56'605. La table 7, montre la structure du deuxième fichier contenant les offset des caractéristiques dans le fichier inversé. Cette table est extraite également du résultat du programme. Les valeurs des offset correspondent à la position du pointeur dans le fichier inversé pour chaque caractéristique.

Table 6: structure du fichier inversé

caractéristique	fréquence *cf*	nombre URL	liste des URL et fréquences	
			URL	fréquence *df*
0	1	2	261	255
			433	255
1	0	0		
2	13	25	27	255
			96	255
			112	255
			114	255
			135	255
			…	…
…	…	…	…	…

Table 7: les offset des caractéristiques dans le fichier inversé

caractéristique	offset
0	0
1	28
2	40
3	252
...	...

7.3 *Implantation du serveur*

Le serveur à implanter doit assurer l'interaction entre l'interface graphique et la structure de fichier inversé de la base de données. L'interaction avec l'interface graphique est établie grâce au protocole de communication. Le serveur est implanté de telle sorte qu'il attend toujours une demande de connexion venant d'un client, qui n'est autre que l'Applet *java* de l'interface. En effet, quand le serveur est lancé, il faut s'assurer que son processus est toujours en "vie". Une manière de réaliser ceci est de créer un *crontab*[1] qui exécute un script *C-Shell* toutes les minutes afin de relancer le processus du serveur s'il s'est arrêté.

Comme l'interface graphique, le serveur assume trois tâches essentielles dans le système *Viper*:

1. Attendre une connexion de l'interface graphique et utiliser le protocole de communication pour envoyer et recevoir des messages avec l'interface.

2. Effectuer l'analyse lexical et syntaxique, grâce à *Lex* et *Yacc* des messages, et les stocker dans une structure de classes *C++* similaire à celle des classes *Java* de l'interface.

3. Utiliser le fichier inversé pour trouver les images similaires à une requête (voir l'équation EQ4 de la section 3.5).

7.3.1 Connexion avec l'interface

Le code source du serveur (*Server.cc*) est implanté en *C++*. Lorsqu'il est lancé, il attend une demande de connexion de l'interface sur un numéro de port défini. Comme nous l'avons expliqué dans le chapitre 6.1, après l'acceptation de la connexion, le serveur reçoit le message "*HANDSHAKE_INTERFACE*" pour reconnaître l'utilisateur de l'interface, ainsi que la version du protocole de communication. Le serveur envoie à son tour le message *"HANDSHAKE_SERVER"* à l'interface pour lui faire connaître la version du protocole, et les données relatives à l'utilisateur. Le serveur se remet ensuite en attente d'une nouvelle demande de connexion pour traiter d'autres messages.

[1] un *crontab* est un processus sous UNIX qui permet de planifier l'exécution d'autres processus à des moments définis.

Comme dans le cas de l'interface, le code du serveur possède des fonctions pour communiquer avec l'interface:

- ***ReadMessage()***: cette fonction lit, par le canal de communication de type *socket*, les octets envoyés par l'interface jusqu'à la rencontre du caractère '**]**' de fin de message.

- ***SendMessage(const char *str)***: cette fonction envoie la chaîne de caractères *str* à travers le canal de communication de type *socket* à l'interface.

7.3.2 Traitement des messages

La manière dont l'analyseur lexical (*flex*) reçoit les caractères en entrée pour effectuer l'analyse lexicale, peut être contrôlée en redéfinissant la ***macro*** *YY_INPUT*. Cette *macro* est déclarée dans le fichier de spécification de l'analyseur lexical et dans le code source du serveur. La déclaration est faite de la manière suivante:

```
{
     char c =  ProtocolMessage[gCount++];
      result = (c == EOF) ? YY_NULL : (buf[0] = c, 1);
}
```

Chaque message reçu de l'interface par la fonction *ReadMessage()*, est stocké dans le tableau de chaîne de caractères *ProtocolMessage[]*. Le compteur *gCount*, initialement à zéro, permet de pointer le prochain caractère du message à analyser avec l'analyseur lexical. Après la réception du message, la fonction *yyparse()* de l'analyseur syntaxique est appelée. L'analyseur lexical envoie les unités lexicales à l'analyseur syntaxique pour vérifier la grammaire du message. Si le message est incorrect, un message du type "*ERROR*" est envoyé à l'interface. Dans le cas contraire, l'analyseur syntaxique place les données du message dans des classes et des sous-classes *C++*, pour être récupérées et traitées par le serveur. Comme dans le cas de l'interface, les classes sont formées de fonctions *setNomFonction(…)* pour stocker une information et *getNomFuction(…)* pour récupérer l'information. Les différentes classes avec leurs méthodes *setNomFonction(…)* sont:

- ***WelcomeInterfaceList.cc***: c'est la classe qui stocke le message "*HANDSHAKE_INTERFACE*". Elle contient deux fonctions:
  1. ***setUserName(const string& str)***: stocke le nom utilisateur de l'interface graphique dans la chaîne de caractères *str*.
  2. ***setVersion(const string& str)***: voir chapitre 6.3.3.

- ***RequestList.cc***: cette classe stocke le message du type "*REQUEST*". Elle contient six fonctions:
  1. ***setCollectionName(const string& str)***: stocke le nom de la collection des images de la requête.
  2. ***setAlgorithmeId(const int intValue)***: stocke l'index de l'algorithme de recherche choisi par l'interface.

3. ***setMaximumNumber(const int intValue)***: stocke le nombre maximum voulu d'images similaires aux images de la requête.
4. ***setThreshold(const float floatValue)***: si le nombre maximum n'est pas précisé, cette fonction stocke le seuil de similarité souhaité avec les images de la requête. La valeur du seuil est un réel entre 0.0 et 1.0.
5. ***setElementList(const ElementList& objectValue)***: stocke la liste des images dans une sous-classe du type *ElementList*.

- ***ElementList.cc***: voir la classe *ElementList.java* du chapitre 6.3.3.

- ***LoadList.cc***: cette classe stocke le message du type "*LOAD*". Elle contient quatre fonctions:
  1. ***void setCollectionName(const string& strValue)***: stocke le nom de la collection d'images à charger sur l'interface.
  2. ***setLoadType(int intValue)***: stocke le type de chargement des images. Le type peut être aléatoire ou fixe. Dans le cas du type fixe, l'interface charge une image depuis le *World Wide Web*.
  3. ***setElementNumber(int intValue)***: stocke le nombre d'images à charger aléatoirement sur l'interface s'il s'agit du type aléatoire sinon le nombre d'images est fixé à 1.
  4. ***setUrl(const string& Urls)***: stocke l'adresse *URL* de l'image chargé depuis le *World Wide Web* sur l'interface.
  5. ***setElementType(int intValue)***: stocke le type d'éléments à charger. Dans cette version du protocole nous utilisons des images. Mais ce champ peut servir à charger d'autres types d'éléments multimédia (vidéo, texte, etc.).

- ***ErrorList.cc***: voir la classe *ErrorList.java* du chapitre 6.3.3.

- ***EventList.cc***: cette classe stocke le message du type "*INTERFACE_EVENTS*". Ce message n'est pas encore utilisé dans le protocole de communication. La classe contient deux fonctions:
  1. ***setEventCode(int intValue)***: stocke le code correspondant à l'événement survenu sur l'interface. Dans ce cas, nous supposons que le serveur possède une base de données avec une liste de codes et d'événements correspondants.
  2. ***setExtension(string strValue)***: stocke une chaîne de caractères correspondant à une information additionnelle concernant l'événement.

- ***TakeBackSessionList.cc***: cette classe stocke le message "*TAKE_BACK_SESSION*". Elle contient une fonction:
  1. ***setSessionName(string strValue)***: stocke le nom de la session que l'utilisateur de l'interface voudrait reprendre.

- ***EndSessionList.cc***: cette classe stocke le message "*END_SESSION*". Ce message n'est pas encore utilisé dans le protocole de communication. La classe contient deux fonctions:
  1. ***setSessionFlag(bool floatValue)***: stocke une valeur booléenne pour informer le serveur si la session que vient de terminer l'utilisateur doit être sauvegardée ou non.
  2. ***setSessionName(string strValue)***: stocke le nom de la session à sauver. Si ce champ est omis par l'utilisateur, c'est la date courante qui est prise en considération.

La manière de traiter les messages du protocole de communication dans ce chapitre est similaire à celle utilisée par l'interface, sauf que chaque partie traite les messages qu'elle reçoit.

7.3.3 Utilisation du fichier inversé

Dans le code source du serveur, la déclaration de la structure de fichier inversé est faite par les constructeurs des classes `CinvertedFileAccessor` et `CInvertedFileQuery`. Dans les arguments du constructeur de la première classe, les noms des différents fichiers qui ont servi à la construction de la structure de fichier inversé sont définis. Cette partie de la déclaration est faite de la manière suivante:

```
CInvertedFileAccessor lAccessor (   "mInvertedFile.db",
                                    "mIFOffSet.db", "urlfts",
                                    "mIFFeatureDescription.db");
CInvertedFileQuery lQuery(lAccessor);
```

La classe `CinvertedFileAccessor` permet de définir les fichiers suivants:

- Le fichier `mInvertedFile.db` correspond à la structure du fichier inversé.
- Le fichier `mIFOffSet.db` correspond à la liste des offset des caractéristiques dans le fichier inversé.
- Le fichier `urlfts` fait la correspondance entre les *URL* des images et leurs fichiers des caractéristiques.
- Le fichier `mIFFeatureDescription.db` correspond à la liste des index des caractéristiques avec leurs descriptions.

La classe `CinvertedFileQuery` permet de récupérer la liste des *URL* des images significatives du résultat de la requête.

Lorsque le serveur reçoit un message du type "*REQUEST*", les *URL* des images significatives et non significatives sont stockées dans une classe *CrelevanceLevelList* de la manière suivante:

```
// déclaration de la classe
CRelevanceLevelList lRLL;

// stockage de l'URL de l'image non significative d'index i dans la classe
lRLL.push_back(CRelevanceLevel(UrlsStr[i],0.0));

// stockage de l'URL de l'image significative d'index i dans la classe
lRLL.push_back(CRelevanceLevel(UrlsStr[i],1.0));
```

Quand toute la liste des images de la requête est stockée dans la classe, l'appel de la classe `lQuery.query()`en passant la liste des URL des images significatives de la requête en argument:

```
CRelevanceLevelList* lOutList = lQuery.query(lRLL);
```

A la fin, les fonctions de chaque élément de la liste `lOutList` sont appelées:

`getRelevanceLevel()` et `getURL()`

permettant de retourner respectivement le niveau de signification d'une image et son *URL* similaire aux images significatives de la requête. Pour plus de détails sur le calcul de la similarité, il faut consulter le paragraphe 3.5 du chapitre 3.

8 Quelques résultats de recherche d'images

8.1 *Recherche de billets de banque*

Dans ce chapitre, l'exemple proposé pour tester le système *Viper* est la recherche de billets de banque. Avant de montrer les différentes étapes de cet exemple, il faut connaître quelques types d'images dont les caractéristiques sont stockées dans la structure de fichier inversé. La figure 23 montre quelques variétés d'images de la base de données. Toutes ces collections d'images proviennent de la Télévision Suisse Romande et leur nombre utilisé pour tester le système *Viper* est de 500.

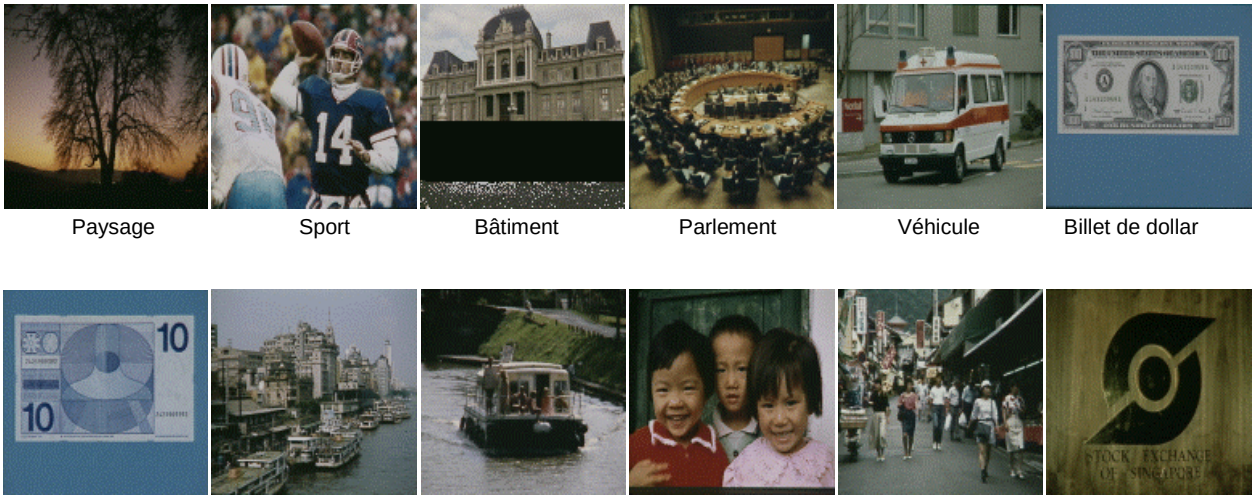

Figure 23

Le but de notre exemple est de rechercher des images qui contiennent des billets de dollar, en sachant que la base de données en contiennent quelques-uns. La première étape est de charger aléatoirement des images sur l'interface graphique pour lancer une première requête. La figure 19 donne une liste de dix images aléatoires fournie par le serveur. Deux images correspondant à des billets de banque sont désignées comme significatives pour lancer la requête. Après avoir lancé cette requête, nous avons obtenu les résultats de la Figure 24. Ces images présentent toutes des billets de banque et parmi ces billets, nous avons quatre billets de dollar qui sont choisis comme images significatives pour lancer une deuxième requête.

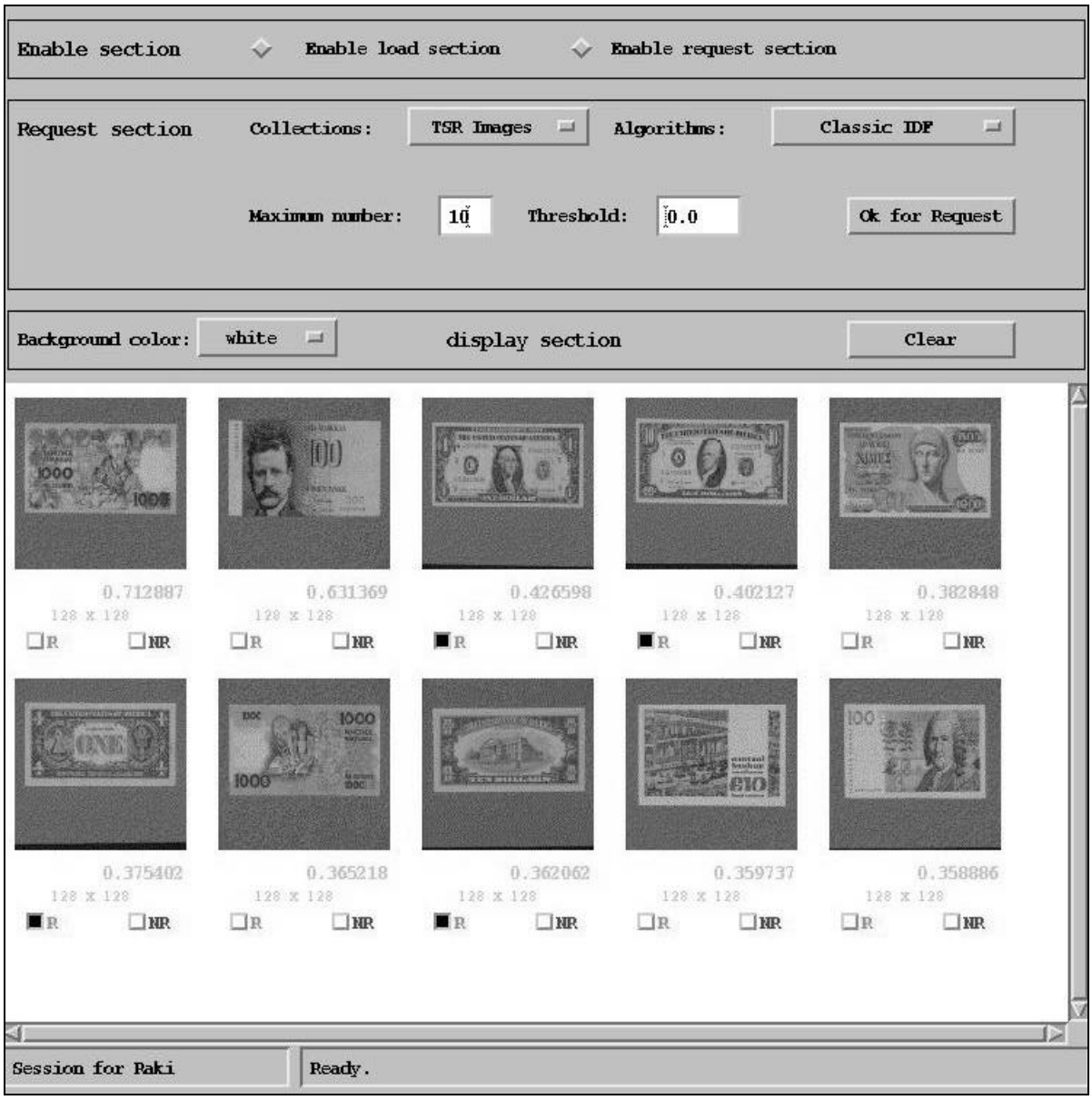
Enable section
Enable load section
Enable request section
Request section
Collections:
TSR Images
Algorithms:
Classic IDF
Maximum number:
10
Threshold:
0.0
Ok for Request
Background color:
white
display section
Clear
0.712887
0.631369
0.426598
0.402127
0.382848
0.375402
0.365218
0.362062
0.359737
0.358886
128 x 128
R
NR
Session for Raki
Ready.

Figure 24

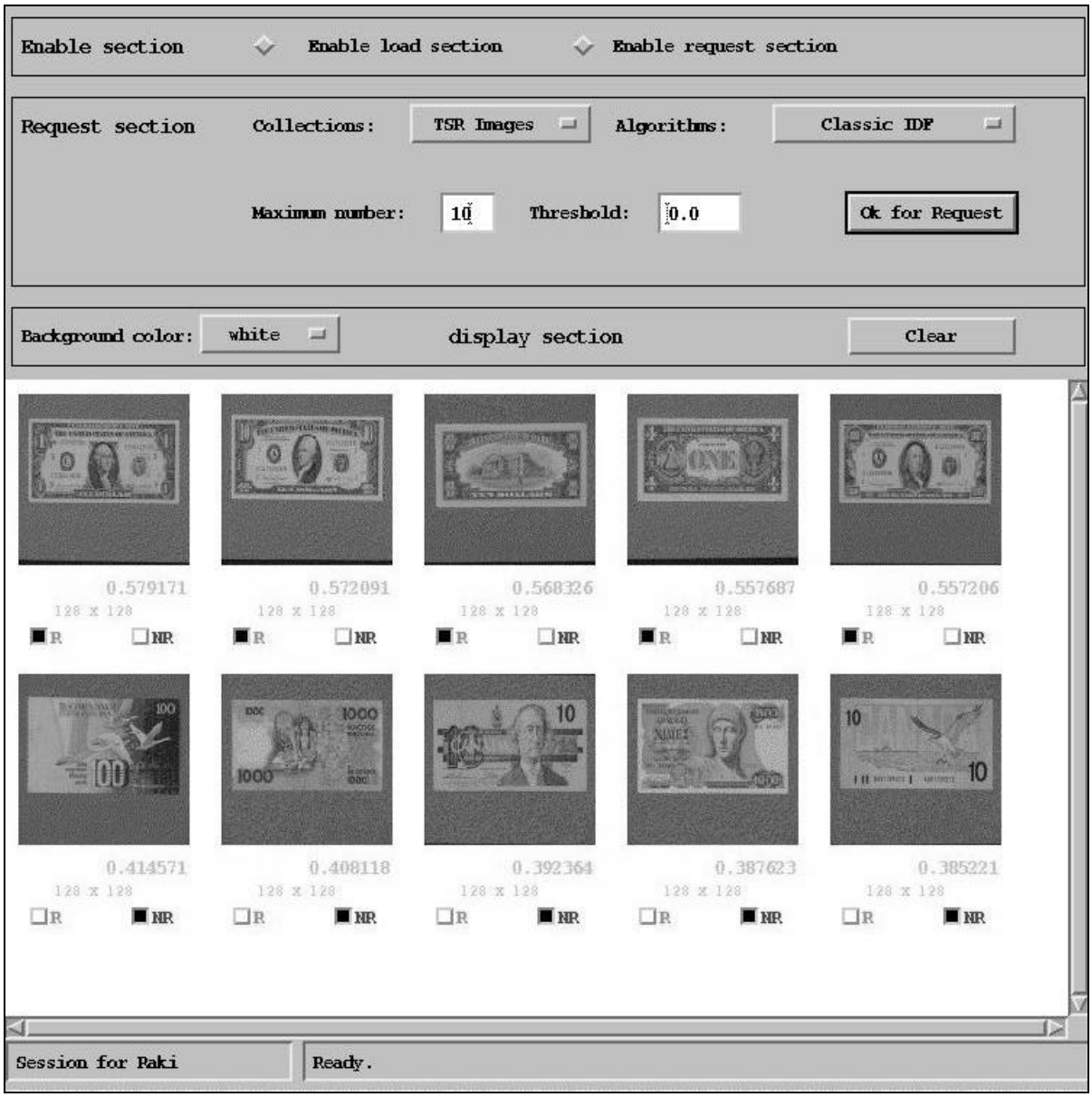

Figure 25

La figure 25 présente les résultats de la deuxième requête. Nous remarquons d'ores et déjà que les cinq premières images correspondent à des billets de dollar plus d'autres billets présentant une certaine similarité avec les images de la requête. Nous n'allons pas nous contenter de ces résultats, nous allons utiliser la non-significativité des autres images pour lancer une dernière requête et savoir s'il existe d'autres images de billets de dollar. Les images des billets de dollar sont choisies comme étant significatives.

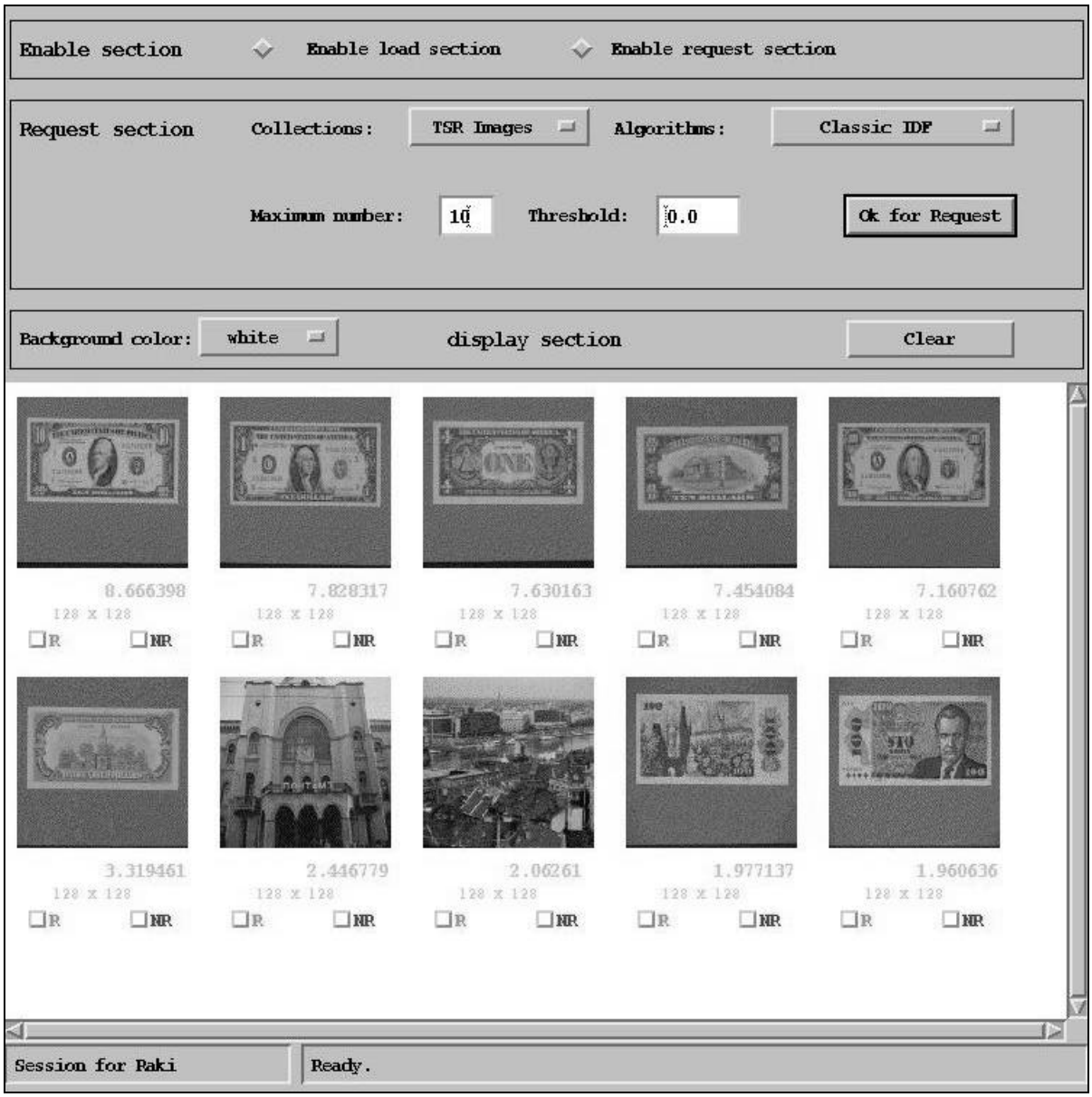

Figure 26

La figure 26 montre un autre billet de dollar supplémentaire que la requête a permis de trouver. En vérifiant dans la base de données le nombre exacte de billets de dollar, nous en avons trouvé six. Ces résultats nous montrent de l'efficacité du système *Viper* à trouver des images similaires dans le cas des billets de banque.

8.2 *Comparaison avec un autre système*

L'exemple du chapitre 8.1 montre l'efficacité du système pour la recherche de billets de dollar américains. Cette expérience ne permet pas d'évaluer les performances du système. Par comparaison avec un autre système de recherche d'images, le système *Viper* peut montrer ses qualités et ses performances. Le système de recherche d'images choisi pour comparaison avec le système *Viper* est le système *Vector Space*, plus traditionnel, et qui utilise un espace de caractéristiques vectoriel de 16 dimensions. Ce type de système est très communément utilisé pour la recherche d'images. On calcule exhaustivement la distance euclidienne entre chaque image de la base de données et chaque image de la requête.

Dans cette expérience, six images sont sélectionnées pour le lancement de requêtes, le but étant de déterminer l'ensemble des images significatives pour chaque requête avec chacun des deux systèmes. La figure 27 montre quatre des images utilisées comme requêtes. Les ensembles des images significatives étaient choisies par cinq utilisateurs et varient suivant leurs nombres différents. Le plus petit nombre d'images significatives d'un ensemble est de trois images alors que le plus grand nombre d'images significatives d'un ensemble est de 19 images. Le degré de similarité varie également d'un ensemble à un autre, allant d'un degré de similarité très élevé dans le cas de l'ensemble des billets de banques "*German Bank-note*" qui possède 7 images véritablement similaires (la collection contient 37 billets de banques de différents pays), à un degré plus faible dans le cas d'ensembles différents d'images pris dans des bibliothèques plus variées.

Parliament *Cityscape* *German Bank-note* *Magic Show*

Figure 27

Pour chaque requête, chaque image est présentée initialement lors du chargement des images aléatoires, sur l'interface. Le nombre d'images demandé pour chaque requête est de 20. Les images retournées après la requête peuvent être choisies comme significatives ou neutres. La non-significativité n'est pas utilisée dans cette expérience.

Les performances des deux systèmes sont comparées en utilisant les deux quantités, *Precision* et *Recall* qui sont définis de la manière suivante:

$$Precision = \frac{r}{N}$$
$$Recall = \frac{r}{R}$$

EQ 7

où *N* étant le nombre total d'images recherché, *r* le nombre d'images significatives recherchées et *R* le nombre total d'images significatives dans la collection.

Les valeurs de *Precision* et *Recall* sont souvent présentées dans un graphe avec la représentation des valeurs de *Precision* en fonction de celles de *Recall*. Ce graphe montre la manière dont la précision décroît au fur et à mesure que la fonction de collection retrouvée augmente. Un graphe *Precision = f(Recall)* est idéal si la précision vaut 1 pour toutes les valeurs de *Recall*, ce qui veut dire que toutes les images significatives suite à une requête sont trouvées avant de rencontrer les autres images non significatives. Les figures 28, 29, et 30 montrent les performances des deux systèmes avec les six images des requêtes données plus haut.

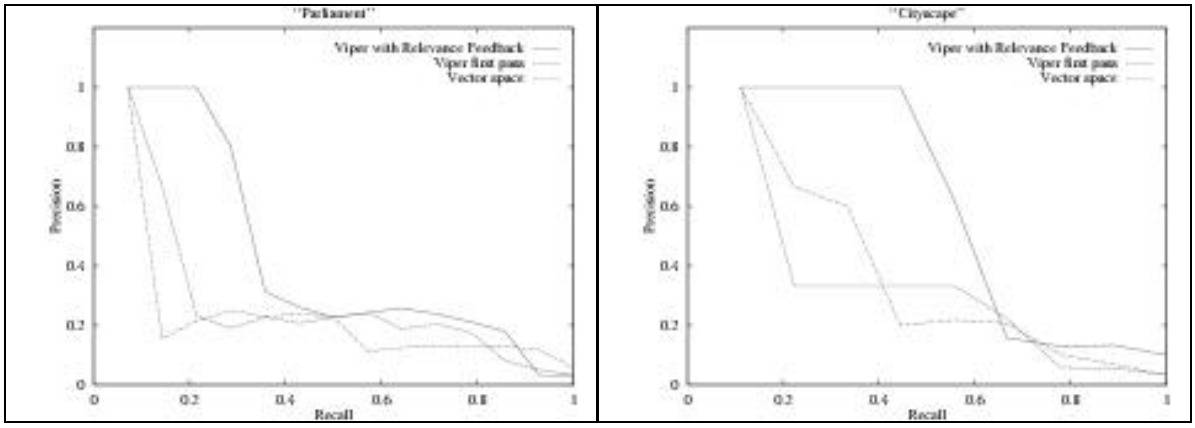

Figure 28

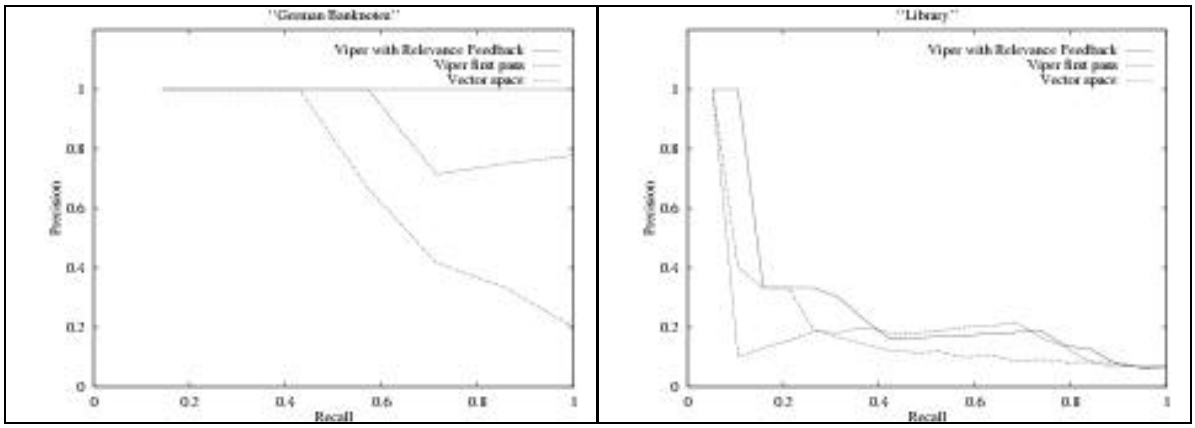

Figure 29

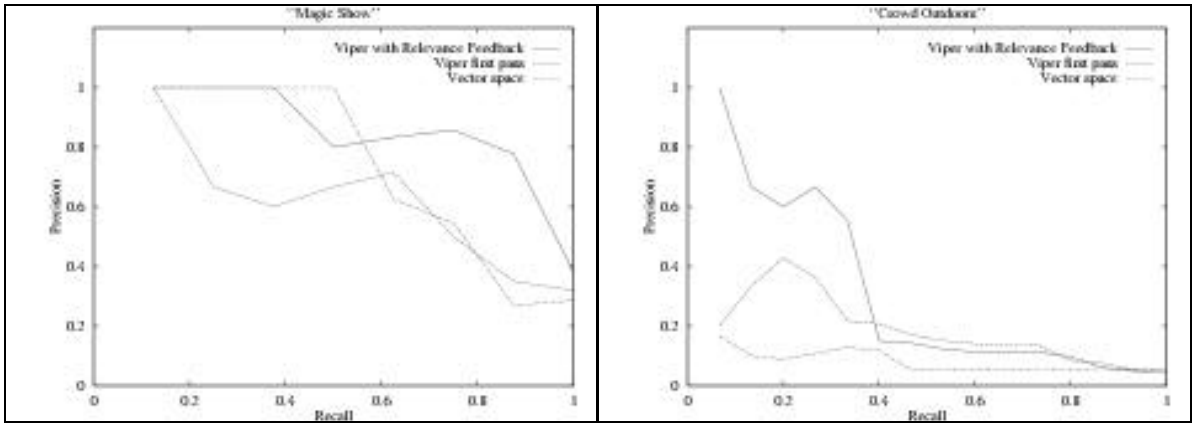

Figure 30

Sur chaque graphe, trois courbes *Precision = f(Recall)* sont tracées. Deux courbes correspondant au système *Viper*, indiquant les performances avant et après les étapes de *relevance feedback*. Il faut rappeler que l'utilisateur ne voit que les 20 premières images après une première passe, il est alors impossible en général d'inclure tous les images significatives dans la requête de retour.

Les graphes indiquent clairement l'énorme influence de la *relevance feedback*. Dans tous les cas excepté pour celui de la requête "*Crowd Outdoors*", l'utilisation de la *relevance feedback* améliore nettement la *Precision*, qui reste à 1 sur une gamme importante des valeurs de *Recall*. La *Precision* des requêtes avec la *relevance feedback* reste plus grande que la première passe du système *Viper*, ou bien celle du système *Vector Space* pour la plupart des valeurs de *Recall*. La performance de la première passe du système *Viper* est également meilleure que le système

Vector Space pour la plupart des requêtes à l'exception des catégories d'images variées comme "*Cityscape*" et "*Magic show*".

9 Conclusion

L'objectif initial d'un système de recherche d'images est de stocker leurs caractéristiques dans une base de données et d'y effectuer des recherches basées sur la similarité aux images de référence des requêtes. L'objectif de notre système était d'introduire une nouveauté par rapport aux autres systèmes. Cette nouveauté était d'appliquer à notre système de recherche d'images la technique de fichier inversé (*inverted file*) pour le stockage de l'information utilisée dans les systèmes de recherche de texte. L'utilisation de la *relevance feedback*, avec notre protocole de communication, nous a permis de rechercher des caractéristiques de manière efficace dans la structure de fichier inversé. Les résultats obtenus avec les billets de banque et, par la comparaison avec un autre système, ont montré un net progrès dans le domaine de recherche d'images, qui reste un domaine très demandé. La demande est particulièrement forte dans le monde du *WWW*, qui utilise de plus en plus de moyens multimédia pour les communications.

L'interface graphique peut être améliorée en construisant d'autre versions, ne modifiant que la partie graphique du code source. La fonctionnalité de pouvoir introduire une nouvelle image depuis le *WWW* sur l'interface est réalisée, mais malheureusement, le serveur ne permet pas encore d'accumuler les caractéristiques des nouvelles images employées. Il s'agit d'une amélioration ultérieure à apporter au système afin que l'utilisateur puisse lancer des requêtes avec ses propres images.

La spécification complète du protocole de communication comporte neuf types de messages. Cependant, la version actuelle de l'interface graphique n'en utilise que cinq types. L'objectif principal de l'interface était avant tout de montrer que la réponse de ce système à une requête d'images donne de bons résultats grâce à la structure de fichier inversé. Dans les futurs travaux sur le système *Viper* actuel, d'autres types d'éléments multimédia peuvent être introduits dans le protocole de communication. La version actuelle du protocole est prévue pour permettre l'introduction de ces nouvelles fonctionnalités. Nous pourrions envisager un système qui donne la possibilité à l'utilisateur de rechercher du texte, de la vidéo ou des images. Une autre amélioration au niveau de la recherche d'images est d'introduire le moyen d'envoyer des régions d'images comme requêtes, ce qui permettrait d'affiner la recherche. L'introduction de nouveaux moyens multimédia, et l'augmentation du nombre d'images rendront les caractéristiques très volumineux. Ceci n'empêche pas l'utilisation de la structure de fichier inversé prévue pour stocker et pour gérer des gigabytes d'information.

10 Remerciements

Après avoir effectué mon travail de licence dans le domaine de recherche d'images de filigranes, qui était un travail très intéressant, je remercie encore une fois le professeur Thierry Pun de m'avoir donné sa confiance pour la réalisation de ce travail de diplôme, qui se situe dans le même domaine. Je remercie également le docteur David Squire pour ses efforts et de ses conseils concernant la mise en place du protocole de communication et de l'interface graphique, Wolfgang Müller pour son aide dans la programmation en langage C++ et sur la manière de trouver une grammaire pour le protocole de communication, Henning Müller pour ses précieuses informations sur la manière de construire la structure de fichier inversé et je remercie également tous ceux qui ont contribué à la réalisation de ce travail de grand intérêt.

11 Bibliographie et références

[I] Karen Spark Jones and Peter Willett eds., Readings in information retrieval, Morgan Kaufmann Publishers, Inc., 1997. (surtout pp. 323--338, pp. 355--364).

[II] Ian H. Witten, Alistair Moffat and Timothy C. Bell, Managing gigabytes: compressing and indexing documents and images, Van Nostrand Reinhold, 115 Fifth Avenue, New York, NY 10003, USA, 1994.

[III] Thierry Pun and David Squire, Image archival and retrieval for multimedia databases based on exploratory statistics and geometrical invariants, (internal document), 1997.

[IV] M. Flickner, H. Sawhney, W. M. Niblack et al. Query by image and video content: The QBIC system. *IEEE Computer*, pp. 23-32, September 1995.

[V] Excalibur Visual Retrival Ware SDK 2.1 Technical summary. Web page, 1997.

[VI] A. Gupta and R. Jain. Visual information retrieval. *Communications of the ACM*, 40(5), May 1997.

[VII] S. Sclaroff, L. Taycher and M. La Cascia. ImageRover: a content-based browser for the world wide web. In *IEEE workshop on Content-Based Access of Image and Video Libraries*. San Juan, Puerto Rico, June 1997.

[VIII] C. Carson and V. E. Ogle. Storage and retrieval of feature data for a very large online image collection. *IEEE Computer Society Bulletin of the Technical Committee on Data Engineering*, 19(4):19:27, December 1996.

[IX] A. K. Jain and A. Vailaya. Image retrieval using color and shape. *Pattern Recognition*, 29(8):1233-1244. August 1996.

[X] J. R. Smith and S.-F. Chang. Tools and techniques for color image retrieval. In I. K. Sethi and R. C. Jain, editors, *Storage & Retrieval for image and Video Databases IV*, volume 2670 of *IS&T/SPIE Proceedings*, pages 426-437, San Jose, CA, USA, March 1996.

[XI] W. Ma and B. Manjunath. Texture features and learning similarity. In CVPR'96 [XXI], pages 447-452.

[XII] F. Liu and R. Picard. Periodicity, directionality, and randomness: Wold features for image modeling and retrieval. IEEE Transactions on Pattern Analysis and Machine Intelligence, 18(7):722-733, July 1996.

[XIII] A. Pentland, R. W. Picard, and S. Sclaroff. Photobook: Tools for content-based manipulation of image databases. International Journal of Computer Vision, 18(3):233-254, June 1996.

[XIV] W. Niblack, R Barber, W. Equitz, M. D. Flickner, E. H. Glassman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. QBIC project: querying images by content, using color, texture, and shape. In W. Niblack, editor, *Storage and Retrieval for Image and Video Databases*, volume 1908 of SPIE *Proceedings*, pages 173-187, April 1993.

[XV] R. Zarita and S. Lelandais. Wavelets and high order statistics for texture classification. In Frydrych et al. [XXII], pages 95-102.

[XVI] S. Sclaroff. Deformable prototypes for encoding shape categories in image databases. Pattern Recognition, 30(4):627-642, April 1997. (special issue on image databases).

[XVII] A. D. Bimbo and P. Pala. Shape indexing by multiscale representation. In Smeulders and Jain[XVIII], pages 43-50.

[XVIII] A. W. M. Smeulders and R. Jain, editors. *Image Databases and multi-Media Search*, Kruisslan 403, 1098 SJ Amsterdam, The Netherlands, August 1996. Intelligent Sensory Information Systems, Factury of Mathematics, Computer Science, Physics and Astronomy, Amsterdam University Press.

[XIX] S. D. Cohen and L. J. Guibas. Shape-based image retrieval using geometric hashing. In Proceedings of the ARPA Image Understanding Workshop, pages 669-674, May 1997.

[XX] David McG. Squire, Wolfgang Müller, Henning Müller and Jilali Raki*, Content-based query of image databases, inspirations from text retrieval: inverted files, frequency-based weights and relevance feedback*, In The 11th Scandinavian Conference on Image Analysis, Kangerlussuaq, Greenland, 7-11 June 1999.

[XXI] *Proceedings of the 1996 IEEE Conference on Cumputer Vision and Patern Recognition (CVPR '96)*, San Francisco, California, June 1996.

[XXII] M. Frydrych, J. Pakkinen, and A. Visa, editors. *The 10th Scandinavian Conference on Image Analysis (SCIA '97)*, Lappeenranta, Finland, June 1997. Pattern Recognition Society of Finland.

[XXIII] Jacques Menu, *Compilation: Concepts et exemples en C++*. Tome1. Pages 121-141, Novembre 1993,