Design and implementation of a flexible Content Based Image Retrieval framework

The GNU Image Finding Tool

THÈSE
présentée à la Faculté des sciences de Université de Genève
pour obtenir le grade de Docteur ès sciences, mention informatique

par
Wolfgang T. E. Müller

de
Tübingen (Allemagne)

Thèse N° 3287

GENÈVE
2001
## Contents

Acknowlegdments .................................................. 7
Résumé .................................................................. 9

1 Introduction ......................................................... 21
   1.1 Challenges in content-based image retrieval ........... 22
      1.1.1 Semantic gap ....................................... 22
      1.1.2 Feature Extraction, Indexing, Learning .......... 22
      1.1.3 Knowledge engineering ......................... 24
      1.1.4 Interaction and Visualization .................... 24
      1.1.5 Software engineering ............................ 27
   1.2 An outline of this thesis .................................... 28
   1.3 Contributions in this thesis ............................... 28

2 The architecture of the GNU Image Finding Tool ............ 31
   2.1 Client-server communication layer ....................... 31
   2.2 Session management ...................................... 32
   2.3 Query engine ............................................... 33
      2.3.1 Query processor .................................... 34
      2.3.2 Accessor of the indexing structure ............... 34
      2.3.3 Two examples for query processor/accessor separation 34
   2.4 Feature extraction, indexing .............................. 35
      2.4.1 Examples for feature extraction and indexing .... 35
   2.5 MRML, plug-ins, and the GIFT: a walkthrough ........... 35
      2.5.1 Configuring a session ............................. 36
      2.5.2 Querying ............................................. 36
   2.6 Configuring the GIFT ...................................... 37
   2.7 Summary and future work .................................. 38

3 MRML, a protocol for communication in content-based retrieval systems ........................................ 39
   3.1 Examples .................................................... 39
   3.2 The design of the MRML query shipping framework .... 40
      3.2.1 The structure of XML and “graceful degradation” 41
      3.2.2 A walk through of MRML ......................... 41
      3.2.3 Logging onto a CBIR server ....................... 42
      3.2.4 Interface configuration ............................. 42
      3.2.5 Query Formulation ................................... 45
   3.3 Extending MRML ............................................ 45
      3.3.1 A region query extension for MRML ............... 44
   3.4 State and future of implementations using MRML ....... 47
      3.4.1 Content-based image retrieval systems ............ 47
3.4.2 GUI Clients ........................................ 47
3.4.3 Benchmarking harness ............................ 47
3.5 Future work in the specification of MRML ......... 48

4 Processing queries by example using inverted file indexes 49
4.1 The need for flexible indexing structures ............ 50
4.2 Inverted files and probabilistic weighting .......... 51
4.3 From text to features ................................ 55
4.4 The feature set used in the Viper system ............ 59
4.5 Coping with correlations ............................ 62
  4.5.1 Complete containment: hypernym/hyponym structure ... 62
  4.5.2 Augmenting the index, not the query ........... 64
  4.5.3 Partial containment .............................. 65
4.6 Viper, a query engine using inverted files ......... 66
  4.6.1 Data access in Viper ............................ 66
  4.6.2 Separate normalization ............................ 67
4.7 Summary ............................................ 67

5 Towards unified retrieval of MPEG-7 and other structured multimedia document formats 71
5.1 Conceptual graphs: a primer .......................... 72
5.2 Retrieving conceptual graphs ........................ 73
5.3 Greedy matching for fast graph similarity retrieval ... 74
  5.3.1 Weighting features using node types ............ 76
  5.3.2 Weighting relations .............................. 76
5.4 featurenet .......................................... 77
  5.4.1 Describing features in featurenet ............... 79
  5.4.2 Relating features ............................... 82
  5.4.3 Using featurenet ................................ 84
5.5 bothrops, a query engine for structured queries .... 85
  5.5.1 Tests ............................................ 87
5.6 Conclusion ........................................... 91

6 A Bayesian image browser tracking the user’s goals 95
6.1 Bayesian methods for browsing queries .............. 96
  6.1.1 PicHunter: a static target searching system .... 96
  6.1.2 tracker: modeling the changing mind ............ 97
6.2 Evaluation ........................................... 99
  6.2.1 Simulation ....................................... 100
  6.2.2 User Experiments ............................... 100
6.3 Conclusion ........................................... 103

7 Automatic benchmarking of image browsers 107
7.1 Defining the goal of image browsers ................ 109
  7.1.1 CBIRS using low level features ................ 109
  7.1.2 Formal description of the browsing problem .... 109
  7.1.3 Requirements for a good browser benchmark .... 109
7.2 Ranked QBE on structured annotation ............... 110
  7.2.1 Differences with classical text retrieval ....... 111
  7.2.2 The structure of the annotation ................. 112
  7.2.3 Performance of the annotation .................. 114
  7.2.4 snakemeter A benchmarking harness with pluggable components 115
  7.2.5 Benchmarking an imitation of PicHunter ......... 120
7.3 Conclusion .......................................... 120
## CONTENTS

8 Conclusion 123

8.1 Summary .................................................. 123
8.2 Putting the results into context .......................... 124
  8.2.1 GIFT and MRML: software engineering ................. 124
  8.2.2 Diverse results for a common goal ..................... 124
  8.2.3 Beyond Viper ........................................... 125
8.3 Achievements ............................................. 126
  8.3.1 Query by example ...................................... 126
  8.3.2 Browsing ............................................... 127
  8.3.3 The GIFT framework: combine query engines .......... 127
  8.3.4 Benchmarking of browsers ............................ 127
  8.3.5 MRML: putting the pieces together ................... 128
8.4 Future work ................................................ 128
  8.4.1 Software issues ........................................ 128
  8.4.2 Image retrieval issues ................................. 128
  8.4.3 Towards the intelligent desktop ....................... 129

A Further documentation of MRML 131
  A.1 State machine of MRML client–server communication .... 131
  A.2 query-paradigms ......................................... 133
  A.3 Algorithms .............................................. 133
  A.4 MRML property sheets ................................... 133
    A.4.1 A simple example .................................... 134
  A.5 The DTD of MRML ....................................... 139

B Two simulated moving target tests 147
  B.1 quickhunter ............................................ 147
  B.2 tracker .................................................. 150

C Sample images and sample queries 153
  C.1 Example images .......................................... 153
    C.1.1 20 randomly drawn images from the TSR500 test collection 153
    C.1.2 20 randomly drawn images from the TSR2500 test collection 155
  C.2 Test queries ............................................ 156
    C.2.1 The 10 TSR500 test queries .......................... 156
    C.2.2 The 14 TSR2500 test queries ........................ 157
    C.2.3 The moving target test images for chapter 6 .......... 158
Acknowledgments

This work was supported by the Swiss National Foundation for Scientific Research under grant number 2000-052426.97.

I want to thank Professor Thierry Pun for providing us with excellent working conditions, and for his friendly way of discussing matters, as well as showing understanding for the many choir rehearsals. In short, for making things you don’t get everywhere seem plain normal.

It was Dr. David Squire who formulated this irresistible job offer which made me come to Geneva. Viper is his idea. For this and 18 pleasant months sharing the same office I am very grateful. I thank him very much for soon giving me the freedom to pursue my own ideas while continuing to give suggestions and guidance. I am especially thankful for kicking me to bring the tracker chapter to publication and for being experimental subject in these experiments.

I thank Dr. Stephane Marchand-Maillet, successor of David Squire, for his quiet style of posing the right questions at the right moment and thus changing the focus of this thesis, as well as for help and encouragement in the annotation.

I would like to thank Dr. Nozha Boujemaa, head of the IMEDIA group at INRIA Rocquencourt, as well as Professor Christian Pellegri, head of the AI group at the CUI, for having accepted to be member of the jury.

Henning Müller collected the user data used in this thesis. He greatly contributed to the stability of the GIFT by spotting bugs and glitches as well as fixing some. I am also thankful for PR work regarding MRML.

I thank Zoran Pećenovic for some moments of incredibly fast programming, and for making SnakeCharmer a GNU package, thus greatly helping the GIFT, and thus MRML.

Dr. Shelby Perina and Dr. Sviatoslav Voloshynovskyi helped Stephane in encouraging me to look more closely at what happens within Viper. Again, I thank all of them for controversial discussions which were real fun.

I thank Felix Balado, Dr. Gabriela Czurka, Frederick Degailllame, Nadia Lasri, Maribel Madueno, Lori Petrucci, Patrick Roth, Dr. J.J.K. O’Ruanaidh and all the ones I forgot for contributing to the “vision group spirit”.

Jilali Raki, was our first Diploma student, who programmed the client for the precursor of MRML, MREP. His ability to produce code under pressure, as well as his friendly style remains appreciated and earned him [88]. Our license students Cornelia Luoni (Viper inverted file accessor for MySQL instead of the file system) and Nicolas Chabloz (PHP client for the GIFT) also contributed to the success of the GIFT and were also fun to work with.

Thanks to Carsten Pfeiffer for good discussions about KDE and the GIFT, and of course for kmrml and insights into the free software world.

Dr. Christoph Giess from the Krebsforschungszenrum Heidelberg greatly improved the GIFT framework, as well as the MRML design by his code audit.

I thank Dr. Arjen de Vries for giving MRML some direction by asking tough questions in a friendly way during some mail discussions. I also thank him for some nice days in Enschede and for improving my Dutch pronunciation.
I thank Dr. Audrey Tam, for requesting a GIFT/Perl interface (viPerl) in order to integrate Andy Bagdanov’s demonstration code for her MPEG-7 proposal into the GIFT, and for demonstrating that on the MPEG-7 Geneva meeting. Audrey, Professor Corinne Jörgensen and Dr. Frank Nack intensified my interest into annotation and fast retrieval of annotation. I am very thankful for these discussions over some plates of great spaghettis in the Geneva youth hostel, and some pancakes in the Ethiopian restaurant.

La première version de MRML a été conçu chez mes beaux-parents, Louis et Elisabeth Liénart, qui ont aidé ce processus par de la bonne nourriture française, de la nage dans la Marne et beaucoup de compréhension pour le gendre qu’on ne voyait pas beaucoup.

I am thanking my parents Prof. Dr. Wolfgang F. Müller and Gerhild Müller, as well as my sister Gerhild Beneke for scientific and especially the non-scientific discussions, the music and the support.

Lastly, thanks go to my wife Dr. Sandrine Müller for making everything possible.
Résumé

Introduction

Cette thèse traite des aspects théoriques ainsi que pratiques de la recherche des images par leur contenu visuel (Content-Based Image Retrieval, CBIRs). Dans cette thèse nous décrivons le GNU Image Finding Tool (GIFT, l'outil GNU de recherche d'images), MRML (Multimedia Retrieval Markup Language, langage pour les requêtes multimedia) ainsi que des méthodes utilisées dans le GIFT pour la recherche des images.

Le développement de méthodes pour la recherche des images par leur contenu visuel est motivée par les insuffisances de l'indexation manuelle des images: pour indexer des images manuellement, une personne doit regarder chaque image, et associer l'image avec, par exemple, quelques mots-clés ou d'autres formes de description d'image. Ces mots-clés vont être classés dans une base de données, et les requêtes opèrent uniquement sur ces mots-clés. Cette approche semi-automatique (indexation manuelle, traitement des requêtes automatique) présente des faiblesses: premièrement, annoter des images constitue une charge de travail non-négligeable. Pire, ce qui n'a pas été vu par l'annotateur lors de l'annotation est perdu pour les requêtes. De plus, l'annotation des caractéristiques visuelles d'une image est très difficile.

À l'inverse, un système CBIR indexe des images automatiquement, et il permet de les retrouver à partir d'une requête, qui, dans la plupart des cas, est constituée d'une ou de plusieurs images exemplaires. Il paraît évident qu'une telle approche pour classer des milliers d'images exige des algorithmes de traitement d'image rapides et robustes, ce qui constitue un domaine de recherche très intéressant.

La recherche des images par leur contenu visuel est rendue difficile par le fait que les utilisateurs vont — dans la plupart des cas — chercher des images avec un contenu sémantique ainsi que visuel. Donc, dans le cas idéal, l'utilisateur aurait la possibilité de donner des mots-clés en plus d'une image d'exemple. Dans le cas d'un système CBIR "pur", ceci n'est pas possible, car pour le moment, un tel système n'est pas capable de reconnaître des concepts sémantiques dans un contexte général. On est donc limité aux seules caractéristiques visuelles pour trouver des images. Comme il n'est pas possible de décider à partir d'une seul image ce qui intéresse l'utilisateur dans l'image, un tel système doit prévoir des moyens d'interaction. Trouver des bons moyens d'interagir avec une base de données multimedia constitue également un domaine actif de recherche.

Dans cette thèse, on se concentre sur deux méthodes d'interactions. Par la première, appelée contrôle de pertinence (relevance feedback), l'utilisateur a la possibilité d'exprimer pour plusieurs images, si elles sont pertinentes pour la requête, indifférentes, ou non-pertinentes pour la requête. Ceci permet à l'utilisateur de raffiner la requête initiale. Au moyen de la seconde méthode d'interaction, appelée image browsing le système présente à plusieurs reprises un sous-ensemble de la base d'images, permettant à chaque fois de sélectionner les images qui correspondent plus
à l'image recherchée *que les autres* images de la même sélection.

**Chapitre 2: L’architecture du GIFT**

Après l’introduction, le deuxième chapitre donne une description succincte de l’architecture du GIFT. Dès le début de notre travail il était clair que le GIFT allait être utilisé dans un contexte de recherche, avec des tailles de collections d’images modérées, mais avec des fortes exigences sur la flexibilité du système pour en faire un bon outil de recherche.

![Diagram of GIFT architecture](image)

*Figure 1: Une vue structurale du système GIFT.*

Une de ces exigences est une approche client/serveur, pour rendre possible une démonstration immédiate de nos résultats de recherche sur le world wide web. De plus, grâce à cette architecture on bénéficie automatiquement d’une séparation données/interface très nette. L’utilisation d’un protocole de communication client/serveur visible par l’humain facilite l’implementation de l’interface client/serveur dans des langages de programmation différents avec un processus de débogage facile. Un
nouveau protocole est présenté ici, appelé MRML. MRML est une DTD (définition d’un type de documents) XML, c.a.d. une grammaire qui définit un langage.

MRML a influencé tout le développement du GIFT. Très vite, il est devenu clair, que la flexibilité gagnée en utilisant MRML va de pair avec son utilisation à l’extérieur du GIFT. Comme cela, les changements à l’intérieur du GIFT n’affectent pas du tout ses couches de communication. De même il, a rapidement été réalisé qu’il était utile de configurer le GIFT en utilisant une variante de MRML, utilisant le même code pour cette configuration et pour la communication client/serveur.

Le GIFT permet que plusieurs utilisateurs soient connectés sur le GIFT, chacun ayant sa propre “session” et sa propre configuration. Ceci suggère une architecture qui sépare les données qui décrivent une session (l’état de la session et de la requête), des données qui peuvent être partagées entre les sessions (accesseurs). Cette séparation a été très utile pour le développement du GIFT. Pour rendre l’ajout de nouveaux algorithmes au système le plus facile possible, les processeurs de requête ainsi que les accesseurs ont été réalisés comme greffons (plug-ins).

Un corollaire de cette architecture est le fait de pouvoir configurer des processeurs de méta-requêtes (c.a.d. des requêtes qui sont distribués à d'autres processeurs de requêtes) pendant l'exécution du système.

Chapitre 3: MRML, un protocole de communication pour des systèmes de recherche d'images par leur contenu

MRML est un langage de markup pour des requêtes multimédia. Le but de son design n’était ni de concevoir “le” langage pour des requêtes multimédia parfait, ni le langage le plus adapté à un certain domaine de problèmes. Par contre, le but du développement de MRML était un langage qui découpe la recherche pour un bon langage de requête, de la recherche pour des bons algorithmes ou de la recherche pour de bonnes méthodes d’interaction CBIRS/utilisateur. Le but était de créer un outil pour rendre une évolution indépendante possible, tout en gardant un maximum d’interopérabilité.

XML (eXtensible Markup Language, langage de markup extensible) est très similaire à HTML (HyperText Markup Language, langage de markup pour les hyper-textes), ce qui est dû a leur ancêtre commun, SGML (Standard Generalized Markup Language). Un texte XML contient des éléments, dont le début et la fin est indiquée par des balises. Chaque élément peut contenir soit des autres éléments, soit du texte. Un élément peut aussi être vide.

Une balise de début a la forme <balise>. La balise fermante correspondante a la forme </balise>. La balise du début peut contenir des attributs <balise attribut="valeur">.

L’idée de base de MRML est d’enfermer les requêtes entre des balises XML ou d’en faire des attributs XML. Ceci permet à l’humain qui veut lire les échanges de communication, de mieux comprendre l’interaction client/serveur, car ces balises servent comme commentaire (aspect auto-documenté de XML). Plus important, les balises permettent au processeur de requêtes de décider si il est capable d’utiliser le contenu de la balise. Le processeur acquiert donc la capacité de choisir les parties utiles de la requête.

MRML fournit des balises pour ouvrir/fermer et configurer des sessions, ainsi qu’un langage pour spéciﬁer des feuilles de propriétés (property sheets). MRML permet au concepteur d’un processeur de requête d’envoyer au client une feuille de propriétés pour chaque algorithme, donnant ainsi à l’utilisateur la possibilité de régler des paramètres et de mieux conﬁgurer la requête.
Comme seul langage de requête, MRML se révèle performant. Par exemple le message qui transmet une requête pour l'image exemple positif 1.jpg et l'image exemple negatif 2.jpg est le suivant:

```xml
<mrml session-id="1" transaction-id="44">
  <query-step session-id="1"
    resultsize="30"
    algorithm-id="algorithm-default">
    <user-relevance-list>
      <user-relevance-element
        image-location="http://viper.unige.ch/1.jpg"
        user-relevance="1"/>
      <user-relevance-element
        image-location="http://viper.unige.ch/2.jpg"
        user-relevance="-1"/>
    </user-relevance-list>
  </query-step>
</mrml>
```

Ici, la requête query-step contient une liste (user-relevance-list) d'URLs d'images (image-location) dont la pertinence (user-relevance) pour la requête a été décernée par l’utilisateur (user-relevance-element).

Dans ce chapitre, on donne également plusieurs exemples d’extensions de MRML. Pour concevoir, et pour traiter des extensions, on donne les consignes suivantes:

- Si le client ou serveur ne reconnaît pas une balise XML (ou un attribut), il doit ignorer la balise (l’attribut).
- Les extensions doivent être telles que les parsers qui ne comprennent pas l'extension peuvent utiliser le plus d’information possible.

On demande de plus que les extensions soient soumises à http://www.mrml.net, pour que ces extensions soient le plus facilement réutilisables par d’autres groupes de recherche. Plusieurs personnes ont déjà développé des clients MRML, ou sont en train de le faire. Une liste se trouve à la fin du chapitre 3. Il y a également des systèmes automatiques d’évaluation de systèmes CBIR qui se basent sur MRML pour la communication entre système CBIR évalué et système évaluant.

### Chapitre 4: traiter des requêtes par l’exemple en utilisant des fichiers inversés

Les requêtes par l’exemple (query by example, QBE) sont la méthode la plus simple de spécification de requête pour des systèmes multimédia. Ici, l’utilisateur donne une ou plusieurs images exemple pour décrire la ou les images recherchées. Comme elle ne nécessite pas beaucoup de développement coté interface utilisateur, cette méthode pour spécifier des requêtes est très souvent utilisée.

Par contre, pour le traitement des requêtes, QBE pose des problèmes: il faut définir une mesure de similarité entre des images, et il faut indexer les images en utilisant cette mesure. Pire, une seule mesure de distance ne suffit pas, car la similarité entre images n’est pas bien définie. Même deux humains ne sont pas toujours d’accord sur la similarité de deux images. Par exemple, considérons trois images: (1) une femme indienne avec son enfant, (2) une femme anglaise avec son enfant, et (3) un homme anglais. Une personne qui cherche des images d’une femme avec son enfant, va classer les images 1, 2, et puis 3. Une personne, qui cherche des images des personnes anglaises va classer les images 1, 3, et puis 2.
RÉSUMÉ

Ce simple exemple montre que, en général, une seule image exemple ne peut pas suffire comme requête. L’utilisateur doit avoir la possibilité de compléter sa requête par d’autres images. Il doit aussi avoir la possibilité de marquer des images comme étant négatives, c’est à dire de les exclure de la recherche. L’utilisateur va donc lancer une requête initiale, et ainsi obtenir des images résultats, puis marquer quelques images de ce résultat pour les ajouter à la requête initiale.

Les systèmes classiques indexent des images en extrayant un ensemble de caractéristiques de chaque image. L’ensemble des caractéristiques constitue un vecteur de nombres réels. Pour des raisons d’efficacité, la dimensionnalité des vecteurs de caractéristiques est réduite en utilisant une méthode comme la PCA (Principle Components Analysis qui trouve une base orthogonale des vecteurs classés par variance). Les vecteurs ainsi obtenus sont indexés en utilisant une structure d’indexation spatiale ainsi que les arbres k-d (à k dimensions).

Ceci génère deux problèmes : premièrement, ces structures d’indexation perdent de leur efficacité avec l’augmentation de la dimension des vecteurs classées; deuxièmement, les structures ne sont pas conçues pour un changement de mesure de distance après indexation. Or, dans la plupart des cas, les systèmes utilisent l’interaction avec l’utilisateur pour obtenir une nouvelle mesure de distance qui soit mieux adaptée aux buts de l’utilisateur.

Viper, le proceuseur de requêtes présenté dans ce chapitre, adapte des techniques conçues pour la recherche des textes aux cas de la recherche d’images. La recherche de textes (Information Retrieval, IR, recherche d’information) est confrontée à des problèmes similaires aux nôtres : les vecteurs à classer (c.a.d. des textes) sont très longs (chaque mot correspond à une colonne), et la mesure de similitude entre vecteurs doit être modifiable pendant la requête. Comme solution à ce problème, la structure du fichier inversé a été proposée dans le domaine du IR.

Dans le contexte IR, on va traiter un texte comme une liste de mots. Donc, le fichier texte relie chaque document à une liste de mots. La similitude entre deux textes va être établie en comptant les mots qu’ils ont en commun. En général, une telle comparaison prend un temps proportionnel au nombre de mots du document le plus court. Donc, sans fichier inversé, une requête va prendre un temps qui est proportionnel au nombre de documents multiplié par le nombre de mots dans la requête.

Pour améliorer les performances, le fichier inversé va relayer chaque mot à une liste contenant tous les documents qui contiennent ce mot. Pour traiter une requête (une autre liste des mots), on va, pour chaque mot \( m_i \) de la requête \( R \), visiter tous les documents \( D_{ij} \) qui contiennent \( m_i \), et pour chaque \( j \) compter \( m_i \) pour \( D_{ij} \). La complexité baisse considérablement, elle devient maintenant proportionnelle au nombre total des mots comptés.

En fait, le comptage est pondéré : un mot qui est contenu dans peu de documents porte plus d’information qu’un mot qui est contenu dans beaucoup de documents de la collection. Dans ce chapitre on donne une fonction de pondération des mots qui a été dérivée par van Rijsbergen et al.

Pour utiliser des fichiers inversés dans un système d’indexation d’images, il suffit de quantiser les caractéristiques. Au lieu de stocker des nombres réels (par exemple : “20.3% des pixels ont la couleur \( H = 34°, S = 0.1, V = 0.8\)”), on va stocker le fait qu’une caractéristique d’image est contenu dans une intervalle (par exemple “20-25% des pixels ont une couleur comprise dans l’intervalle \( H = 20...40°, S = 0...1, V = \frac{2}{3}...1\)”).

La Fig. 3 montre une partie des caractéristiques, celles qui décrivent les couleurs et leur positionnement (color layout). L’image d’entrée est transformée en une taille de 256 x 256 pixels, puis divisée en 16 x 16 blocs de taille 16 x 16 pixels. Pour chaque bloc, l’intervalle le couleur le plus fréquent est calculé. La même chose est répétée pour 8 x 8 blocs, de taille 32 x 32 pixels etc.
Figure 2: Une image qui visualise comment le fichier inversé lie caractéristiques avec des images. Les points rouges symbolisent des caractéristiques, les tiges vertes symbolisent les caractéristiques qui sont présentes dans la première image.

De manière similaire, des caractéristiques de texture sont calculées en utilisant des filtres de Gabor. Finalement, une image est décrite par Viper en utilisant environ 80 000 caractéristiques, comprenant aussi un petit nombre de caractéristiques globales: des histogrammes de couleur et de texture.

Une grande partie du chapitre est consacrée à l'indépendance statistique des caractéristiques. Les résultats théoriques qui justifient l'utilisation des fichiers inversés ainsi que les fonctions de pondération ont été obtenus sous condition de l'indépendance statistique des caractéristiques. Par construction, les caractéristiques visuelles de Viper sont fortement corrélées, donc ne sont pas statistiquement indépendantes.

On présente dans ce chapitre des corollaires des résultats de van Rijsbergen sur la corrélation des caractéristiques en général. On propose de construire des ensembles de caractéristiques tel qu'ils se contiennent parfaitement l'un dans l'autre. Deux caractéristiques sont contenues parfaitement l'une dans l'autre si un document contenant l'une caractéristique va forcément contenir l'autre aussi. On propose une méthode qui permet de calculer exactement la pondération pour les caractéristiques qui contiennent l'une dans l'autre.

En fin de ce chapitre on donne quelques graphes precision-recall qui permettent l'évaluation de Viper. Pour obtenir un tel graphe, on lance quelques requêtes test. Pour chaque requête, on présente la précision (c.à.d. le nombre d'images pertinentes trouvées divisé par le nombre d'images trouvées) en fonction du recall (c.à.d. le nombre d'images pertinentes trouvées divisé par le nombre total d'images pertinentes pour la requête, ce quotient pouvant varier entre 0 et 1).

N.B. Viper utilise des fichiers inversés, mais n'utilise pas la correction de la
Chapitre 5: vers des méthodes unifiées pour la recherche de documents MPEG-7 et autres formats de documents multimédia

Ce chapitre introduit un nouveau moteur de recherche, appelé *bothrops*, qui apporte des extensions importantes à *Viper*.

*Viper* utilise peu d’information sur les caractéristiques. Pour chaque caractéristique, *Viper* stocke seulement un numéro d’identification ainsi qu’un identificateur de groupe de caractéristiques. Comme on a vu dans le chapitre précédent, il est utile de pouvoir stocker les relations entre des caractéristiques, ainsi que le fait que ces caractéristiques sont corrélees ou non. De même, il est souhaitable de relier des caractéristiques afin de pouvoir mieux modéliser et répondre aux souhaits de l’utilisateur. Un autre problème est que l’on voudrait être capable d’ajouter et de retirer des images pendant que l’exécution du système. Dans le cas général (p.ex. si on veut ajouter des textes aux images), ajouter une image veut dire aussi ajouter des nouvelles caractéristiques. Pour ajouter des caractéristiques à l’ensemble des caractéristiques, il faut d’abord savoir quelles caractéristiques sont déjà présentes dans le système. Il faut donc pouvoir faire des requêtes sur les caractéristiques elle-mêmes. *featurenet* présenté dans ce chapitre, permet une classification des caractéristiques, d’effectuer des requêtes pour savoir si une caractéristique est présente ou non dans l’ensemble de caractéristiques, ainsi que retrouver des relations entre des caractéristiques. *featurenet* est utilisé pour définir un nouvel ensemble de caractéristiques, inspiré par *Viper*, et permet de corriger la pondération de ces caractéristiques. Ces caractéristiques ont été utilisées avec le moteur de recherche pour obtenir (entre autres) les résultats donnés en Fig. 4.

Il faut souligner que nos résultats ne sont pas tout à fait concluants. Avec une
collection de 2500 images, on obtient des résultats qui indiquent que la correction de la pondération améliore les résultats. Par contre, les résultats obtenus sur une collection de 500 images étaient dégradés par la correction de la pondération. Il n’est pas surprenant que les méthodes statistiques marchent mieux sur les grandes collections, mais nous attendons avec impatience que peut-être grâce au Benchathlon (voir http://www.benchathlon.net), une mesure de qualité pour des CBIRS soit établie, pour en tirer les conclusions.

Le deuxième thème abordé dans ce chapitre est le fait de créer des requêtes structurées. Implicitement ce thème était déjà abordé un peu plus haut: classer des documents dans un fichier inversé fait perdre la structure. “Un homme mord un chien devient équivalent à “Un chien mord un homme”. Le fichier inversé ne donne pas de moyen de distinguer ces deux phrases. Or, pour des requêtes multimédia, on a besoin de structure: relations spatiales entre des segments dans un image, relations temporelles entre des images dans une vidéo, etc. MPEG-7 est maintenant l’exemple par excellence pour des données multimédia structurées. MPEG-7 est un projet de standard international pour créer un format commun de description des données multimédia. Cette description contient des descripteurs visuels ainsi que sémantiques. Il existe une large gamme de descripteurs simples ainsi que structurés.

Dans ce chapitre on présente un algorithme qui traite des requêtes de similitarité entre graphes attribués. La structure des documents traités est similaire aux graphes conceptuels simples: un document est représenté par des sommets. Chaque sommet a des caractéristiques non-structurées (dites “plates”) ainsi que des relations nommées avec des autres sommets. Pour calculer la similitarité entre deux documents $D_1$ et $D_2$, on va d’abord associer les sommets les plus similaires entre ces deux documents. Après, on va vérifier que les relations à l’intérieur de $D_1$ ont la même structure que dans $D_2$. Cette approche permet la requête approximative des graphes, tout en restant rapide.
Chapitre 6: un browser d’image qui suit les changements de but de l’utilisateur

Jusqu’ici on a parlé des requêtes par similarité. Ce chapitre aborde le problème du browsing d’images. Dans le browsing, l’utilisateur entame la requête sans apport initial: le système propose une sélection d’images, l’utilisateur choisit les images les plus proches et les plus éloignées de la requête dans cet ensemble, et demande une autre sélection. Un bon système permet à l’utilisateur d’atteindre le but de la recherche dans un nombre d’itérations minimal. On essaye de minimiser le nombre d’images que l’utilisateur doit regarder avant qu’il atteigne le but de sa recherche.

PicHunter avait été évalué par un target test (test de recherche de cible). Dans un tel test, on présente une image à l’utilisateur, et il lui est demandé de trouver cette image le plus rapidement possible en utilisant le browser. On compte le nombre d’itérations.

Dans ce chapitre, on étend le système PicHunter conçu par Cox et al. dans les laboratoires de NEC à Princeton. Ce système maintient une distribution de probabilité qui indique pour chaque image si cette image est l’image recherchée. A chaque pas de l’interaction, cette distribution de probabilité est mise à jour.

A notre avis, ce système présente deux faiblesses principales: premièrement, le système ne modifie pas son modèle utilisateur pendant la requête. Si le modèle utilisateur utilisé pour la mise à jour de la distribution de probabilité est erroné, il ne va pas être change durant le processus de requête. Deuxièmement, le système est incapable de détecter si l’utilisateur change d’avis, donc si l’utilisateur change l’image cible durant le processus de recherche.

Dans le chapitre 6, on définit une approche qui permet de détecter si l’utilisateur entre des données incohérentes, ceci permet ainsi “d’oublier” sélectivement des anciennes données entrées par l’utilisateur qui seraient incohérentes avec des données nouvelles. Ce système, tracker, est testé en utilisant une modification du test de recherche de la cible, le moving target test, le “test à cible mobile”. Ici, on demande à l’utilisateur de visiter un nombre d’images dans une séquence donnée.

Nos tests indiquent l’utilité de cette approche, mais il nous semble que le changement du modèle utilisateur est nécessaire pour obtenir une bonne performance du système. Nos tests ont également montré que le problème d’évaluation des browsers d’images n’est pas du tout trivial. Ce qui nous a amené à traiter ce problème dans le chapitre 7.

Chapitre 7: évaluation automatique des browsers d’images

Evaluer des browsers d’images est rendu difficile par le fait que l’utilisateur apprend pendant les tests. Ceci rend difficile d’évaluer l’évolution d’un système pendant le développement, ainsi que la comparaison des systèmes qui utilisent des paradigmes de browsing différent. Par exemple, il est très difficile de comparer des browsers qui préparent une fois une hiérarchie d’images avec des systèmes stochastiques comme PicHunter, qui préparent une nouvelle sélection d’images pour chaque pas de la requête pour chaque utilisateur. La difficulté vient du fait que dans le premier cas, l’utilisateur test va pouvoir apprendre à chaque utilisation quelle image est où dans l’hierarchie. Dans le deuxième cas, l’utilisateur ne peut pas faire un tel apprentissage. Donc, il devient impossible de comparer l’utilité du système pour un utilisateur qui découvre une nouvelle collection d’images, sans changer le groupe d’utilisateurs à chaque test, ou changer la collection d’images. Notre but est donc de concevoir une méthode d’évaluation automatique pour des browsers d’images, et
RÉSUMÉ

donc, de trouver une méthode pour simuler l'utilisateur d'un browser.

Pour trouver une mesure pertinente pour comparer des browsers d'images, on propose d'abord de regarder pourquoi on utilise un browser d'images. Le browser doit nous aider à exprimer quelle image est recherchée. On suppose que l'utilisateur cherche une image qui exprime quelques concepts sémantiques.

De cette observation on tire deux conséquences: premièrement, une approche qui simule un utilisateur en utilisant des caractéristiques visuelles devient insuffisante. Par contre, on va pouvoir simuler l'utilisateur en utilisant une annotation sémantique d'images. Ceci n'est sûrement pas parfait, mais constitue une façon de simuler la différence entre l'index qui est basé sur les caractéristiques visuelles, et la pensée de l'utilisateur.

Comme base d'une telle mesure de distance entre des images, on a conçu une méthode d'annotation structurée. L'avantage d'une telle structuration d'annotation est qu'on peut introduire de la connaissance a-priori. Si, dans l'annotation, la différence est faite entre des acteurs et les actions, entre attributs et noms, il est possible de mettre plus de poids sur l'action, et moins sur le lieu, par exemple.

Notre annotation structurée est formulée comme un texte “pseudo-prolog”:

```
actor(man,$man1).
actor(man,$man2).
action(ask).
performs($man1,$ask). % read: "man1 performs ask"
isPerformedOn($ask,$man2). % read: "ask is performed on man1"
```

Ce pseudo-prolog est traduit dans un graphe, et ce graphe est complété, comme indiqué Fig. 5, pour guider la recherche. Nos expériences encouragent une utilisation de cette annotation pour notre outil d'évaluation des browsers.

![Figure 5: La structure augmentée de la phrase "Un homme demande quelque chose à un autre homme.". Voir pages 18 ou 75 pour plus d'explications.](image)

Pour tester notre idée, l'annotation à été intégrée dans le contexte *snakemeter* d'évaluation (Fig. 6). Ce framework constitue une base pour le développement des systèmes d'évaluation de CBIRS.

*snakemeter* utilise MRML pour la communication. En fait, MRML est utilisé des deux cotés: *snakemeter* communique en MRML avec le système qui constitue la mesure de similarité de référence, et *snakemeter* communique en MRML avec le système qu'il est en train d'évaluer, comme montré à la Fig. 7.

Nos expériences montrent qu'en l'état actuel des recherches, le benchmark proposé est très difficile. Notre système de test (une variante simple de PicHunter)
Figure 6: *Un cadre pour évaluer des systèmes CBIR, snakemeter est un prototype de ce cadre.*

Figure 7: *L’annotation dans le benchmark automatique des browsers.*

obtenait un score moyen qui était seulement 30% meilleur que ce qu’aurait été un tirage successif d’images par pur hasard. On utilise ce résultat comme motivation pour développer un browser qui est capable de modifier son modèle d’utilisateur pendant la recherche.

**Chapitre 8: Conclusion**

En conclusion, cette thèse a abordé et proposé des solutions à plusieurs problèmes liés au CBIR:

- Le développement d’une architecture flexible pour des moteurs de recherche multimédia;
- la communication client/serveur dans le contexte des requêtes multimédia;
- l’indexation d’images en utilisant des fichiers inversés;
- l’indexation des données semi-structurées, en utilisant des fichiers inversés;
- le browsing d’images dans un contexte où l’utilisateur change d’avis;
- et finalement une approche originale pour évaluer des browsers d’images.

Pour chaque partie nous avons obtenu des résultats encourageants, et qui ouvrent de nouvelles questions, difficiles et intéressantes.
Chapter 1

Introduction

This thesis covers both theoretical and practical aspects of one of the most challenging tasks of modern computer science: content-based image retrieval (CBIRS).

While being extremely challenging and ultimately impossible, it becomes interesting by the practical usefulness even of imperfect solutions.

Like many of the difficult tasks in sciences, the task of a CBIRS is easy to state:

Help a user in finding the image(s) he wants from a collection of images.

However, this task is far from simple. True image understanding is still impossible in the case unconstrained image collections, so our problem resembles closely to what one would face in the following scenario:

Imagine, for example, you were replacing during two weeks a librarian in a library for scientific images, let’s say: images of rocks. Imagine most of your clients don’t speak your language. Your clients know that there is this strange guy who is not really fit for the job, and who does not speak nor read their language, so they usually come with examples. You tend to bring images that you regard similar, and people will tell you (by pointing on the images and nodding or shaking the head), what images they consider particularly interesting or particularly uninteresting. You will refine your search according to these additional examples.

The above story of a librarian’s nightmare is a rather simplified\(^1\) example of the problems a designer of a CBIRS is facing: finding images without understanding them.

It is evident that the computer’s lack of image understanding causes “imperfect” retrieval results. Even if the query was perfectly “understood” by the software, the query results would be imperfect\(^2\). On the other hand, query formulation is rendered difficult because only the querying user has understood the image, but not the query processing software. They do not operate or think based on the same notions. The gap between the semantic view by image–understanding humans and the purely visual view of non–image–understanding CBIRS is called the semantic gap. Coping with the semantic gap is the main challenge of CBIR, and the motivation for all the research in the field.

In this thesis we will show how one CBIRS, the GNU Image Finding Tool, was designed, and in this process we will present the reader with the interesting problems that a CBIRS designer is facing:

---

\(^1\)What makes this example simpler than the CBIRS problem is the fact that in scientific imagery the concept of similarity is much better defined than in unconstrained photo collections.

\(^2\)We shall show that in our context even the notion of a perfect retrieval result is ill-defined.
• computer vision: find a similarity measure on images;
• knowledge engineering: combine textual and visual knowledge;
• computer learning: modify the similarity measure on images by learning what the user wants;
• complexity considerations: index the images to make a fast and useful tool;
• human-computer interaction: tell your user about the similarity measure by visualizing what different choices mean in terms of your similarity measure. Use computer learning for giving the user the possibility to move consciously in feature space;
• evaluation problem: guide the quest for perfection by defining what’s good performance;
• software engineering: build a framework that is flexible enough for research and standardized enough interchange and reuse of CBIRS components.

All bulletpoints, except maybe for the last in this list are related to the semantic gap. This relation will be pointed out in the next section. This thesis contains original contributions to some of these fields.

1.1 Challenges in content-based image retrieval

1.1.1 Semantic gap

In the previous section we have stated that image retrieval is difficult, and ultimately impossible. What makes this task so difficult, is the semantic gap. The semantic gap is the discrepancy between the user’s semantic oriented representation of the image and the CBIR’s low-level representation. “Semantic” means here that a human thinks in terms of what items are on the image, and how they relate to each other, which topics they express, and so on [45]. The CBIR’s low-level representation, however, captures the location and distribution of visual features without being able to relate them enough for eliciting the full semantics of an image. Even the much simpler problem of recognizing all objects on an image (i.e. recognizing only a small part of the full semantics) remains unsolved. This might be surprising at first. However, it is quite evident that object recognition heavily depends on experience and education, as becomes more clear if we look at pictures that are out of our normal context.

Take for example the image of a man wearing a plain ring on the left hand. In some part of Germany the ring on the left hand would be a sign that the wearer is engaged to be married, in Anglo-Saxon countries the same type of ring, worn on the left hand, would be a sign that the wearer is already married. This is just a small example, the interested reader can find many more such examples when reading literature about symbols in foreign countries and diverse cultures.

What is worse, there have been experiments showing that the relevance of two images to each other depends on what other images are shown at the same time, or on the mood of the searcher [94].

1.1.2 Feature Extraction, Indexing, Learning

In most CBIRS, the following steps will be performed:
1.1. CHALLENGES IN CONTENT-BASED IMAGE RETRIEVAL

- Some computer-vision algorithms will extract characteristics from an image. These characteristics are stored as a feature vector of fixed (most systems, like e.g. [15, 68]) or varying length (Viper [60, 87]). This process is called feature extraction.

- Then, items are indexed using spatial indexing techniques, ranging from techniques specialized on vectors or lists [16] and techniques that need just that a metric be defined on the space of items in the database such as [104]. Smellders et al. give a more detailed overview in their review article [78].

- During the query process (i.e. a sequence of queries performed by the user to reach one goal), the system will perform learning in order to improve the system’s performance.

![Image of visualizing information loss](image_url)

Figure 1.1: Visualizing the information loss when calculating a color histogram, a popular feature extraction technique. The parts in which form is recognizable were derived from the original image using quantization within the HSV color-space. The parts in which no form is recognizable are random dots drawn from the local color distribution. For a matching method that is based on the color histogram only, the squares without form would match perfectly with the original squares.

For a long time, the search for the “best” feature sets governed the research in CBIRS (e.g. [43]). Now, as it becomes clear that there is no “best” feature set, the main interest has turned to the right combination of learning method, indexing method and feature set, leading to a system that learns the best feature set for a given query situation.

While query paradigms and visualization techniques (described in one of the following sections) might change, one has to keep in mind that all content-based retrieval techniques boil down to a representation of images and image features that enables a sufficient first-shot performance, and that allows short-term or long-term learning for improving the performance.
1.1.3 Knowledge engineering

Recently, it has been realized that the problem of combining textual image annotation and visual features is far from trivial. At the same time it became clear that some of the potential applications of CBIRS systems are out of reach if one does not resort to partial annotation of images and other multimedia data. This lead to standardization efforts such as MPEG-7 [57] (MPEG is the Motion Picture Experts Group, part of the ISO, ISO/IEC JTC1/SC29/WG11 effort and RDF (Resource Description Framework, an effort of the World Wide Web Consortium [100]).

In our context, MPEG-7 is the most interesting, as it deals with the annotation of multimedia data, as well as the representation of visual and audio features for exchange of such data. Solving the problem of transporting data MPEG-7 does not address nor solve the problem of finding a good indexing/learning structure.

Finding an indexing and querying method that provide meaningful and useful interaction between multimedia appliances and their users is a new challenge posed by MPEG-7.

Annotation, as provided by MPEG-7 helps narrowing the semantic gap. However, a complete closing of the semantic gap will not always be possible: there is always the gap between what the annotator has seen in the image, and what the user sees in the image (this has been stated many times throughout the literature, as e.g. in [78]).

1.1.4 Interaction and Visualization

Due to the semantic gap, query formulation is a far more difficult problem than in the area of (purely text) Information Retrieval (IR). The user interface, and the query process will have to be adapted to query processes that last many interaction steps. It is desirable to guide the user in his choices. Some systems also favor direct input like the drawing of outlines of segments or describing the image by a sketch.

In the following we will introduce the main query paradigms. Almost all of them can be combined with an initial text query on annotation provided with the images to narrow the search.

Query by example: Query By Example gives the user the possibility to give example images from which the system is to infer the properties of the desired query result. This method is very popular, as it is possible to do QBE using very simple user interfaces [60, 67, 87]. Moreover, it is much simpler to evaluate than the other query methods that will be described in the following.

However, it is clear that QBE is a very coarse method of formulating a query. Blobworld [2] is a classic refinement for QBE: The user is presented with a segmented version of the original example, and he is given the possibility to choose one or two segments which correspond most or least to the user’s wishes.

In QBE systems, the user is presented with the query results that allows direct modification of the query. This relevance feedback [76] will be evaluated in the next step. However, this method of refining the query is not comparable to image browsing.

Query by sketch Some groups have attempted to provide query by sketch [15, 34, 40, 41, 92]. However, it is evident that this problem is very hard. There are several reasons:

- In the age of photography, drawing has gotten lost as a cultural technique. People without a very pronounced interest in arts will be unable to draw items in a recognizable fashion.
1.1. CHALLENGES IN CONTENT-BASED IMAGE RETRIEVAL

- Similarity measures developed for photo collections will not work without extension, as even a skilled human drawing an item will subconsciously apply transformations to the drawn items which do not correspond to the transformations performed by cameras. An example is given in Rosalind Picard’s paper on the visual thesaurus [69].

- Except for very skilled drawings, even humans won’t always agree on how to interpret a drawing. This is exploited by games such as Pictionary [71]. Here one player of a team has to draw a concept (in a given time), while his partners are trying to guess which concept is being drawn.

- As we have stated a human’s view on image similarity depends on his educational and cultural background as well as the immediate information need. In query by sketch, these cultural differences are even more pronounced than in other retrieval methods. An example is given in Fig. 1.2. Fig. 1.3 illustrates the author’s view that query systems that are able to answer general queries by sketch have to detect the semantics of images. However, already current systems are able to retrieve photos from drawings that are visually very similar, as seen in Fig. 1.4.

![Image](image.jpg)

Figure 1.2: Example of a query by sketch that can be interpreted in two very different ways. An image like the one above was shown at Visual 2000 in Lyon by Marcel Worringer (for illustrating [103]). He gave it as example for a query by sketch. While the author of this thesis (currently living in Geneva enjoying daily the view to the Alps) instantly thought of a sunrise above mountains, Dr. Worringer from Amsterdam saw it as an example of a sunset at the beach.

There are also systems which soften the original query by sketch paradigm. They provide tools for making crude drawings that reflect the color/texture distribution on the image [81]. Such images are much easier to analyze than drawings, however, also here the use of the drawings will strongly vary with the user’s skill.

**Browsing** While query by example is easy to formulate once you are in the possession of an example, there still remains the problem of how to provide an example. This problem has been described in [97] as the page zero problem. Query by sketch is a method to solve this problem. Another approach is to provide the user with overviews of the whole image collection, and to let him move between overviews, refining and broadening the user’s view on the collection. For refining the view, the user chooses one or more images from the current overview, coarsening the view usually corresponds to clicking some “history back” button. Later in this thesis we
will present a system that automatically broadens the overview if the user feedback becomes inconsistent.

Browsing is very appealing in that it allows the user to interact without asking too much. However, even this method is not without problems, some of which are going to be addressed in this thesis.

It has been perceived that the automatic summarization of collections into few categories is very difficult: while high visual similarity is strongly correlated with high semantic similarity of images, semantic and visual similarity decorrelate more and more for lower ranks. Summarizing an image collection of 50000 images to a summary of 50 representative images, however, would correspond to a QBE system that provides a close-to-human ranking for at least the first 1000 items. However, such a ranking is difficult to achieve.

Recently, there have come up systems that try to give the user more detailed feedback capacities [77], giving the user the possibility to give both the classical relevant vs. non-relevant user feedback, and at the same to group images that are considered to be of the same class in the given situation. By this, the system has much more detailed information about how to weight its features.

At the same time, the research area of image retrieval has been reached by visualization scientists. This gives rise to a new class of image retrieval systems, those who let you explore virtual worlds.

**Virtual worlds** In the field of image retrieval systems that present the user with visual worlds, we can make the distinction between systems that try to mimic the way we live (visit a virtual museum or a virtual arts gallery [46]). While such systems provide a good visual experience, they do not enhance the user’s knowledge about the relationships of images.

In our view most promising is the work that uses the new possibilities computing is able to offer [11, 29,66]. All these systems let the user move in abstract three-dimensional spaces in which the user is able to move among images:

Pecenovic et al. [66] use distance-preserving projections of the multi-dimensional space of image features (feature space) onto the two-dimensional space. The user is enabled to navigate freely in this space. He flies from image to image. Furthermore, the user has the possibility to choose to view just the centers of clusters in order to unchutter the map.

Hiroike et al. [29] make use of the fourth dimension for giving an overview of the collection. They project the 20-dimensional feature space on the tree dimensional space in a crude and simple way (if \( \mathbf{f}_i := (f_{i1}, \ldots, f_{i20}) \) designates position of image \( i \) in the feature space, the projected position of the image \( \pi^{a,b,c} \) is \( \pi^{a,b,c} := (f_{i1}, f_{i6}, f_{ic}) \)). Clearly, this alone would not lead to satisfactory results. If \( a, b, c \) would be fixed,
one could start already with a three dimensional feature space without further degrading the results. What makes this system unique, however, is the fact that the dimensions on which the feature space is projected change over time. As a consequence, the images are constantly in movement. Thus, the system is using the human capability of quickly analyzing images. Now the user can interact with the system by adding images as attraction centers to the moving cloud of images. Images similar to the attraction center will be attracted to this center, thus creating a flow of images. The use of this system is exploiting the human’s pre-attentive, subconscious capabilities.

Chen et al. [11] are generating tree structures of related elements and visualizing them as a multikimensional graph structure which is automatically laid out in three dimensions. The result is visualized using VRML (Virtual Reality Modelling Language). The current system does not adapt to the user’s needs on-line. However, this system helps the user by providing a global view on the data and by selecting useful interconnections between pieces of data.

[20,105] provide additional information about information visualization.

1.1.5 Software engineering

As we have seen in the last sections, there are several blocks that have to be combined for obtaining a CBIR system. About all of them are very active fields of research. Large projects, like e.g. the MIA project in Amsterdam, are trying to provide a software infrastructure that lets people from different research areas work together on an integrated way of treating annotated multimedia information. Also
smaller projects, like the GIFT (GNU Image-Finding Tool) profit from an architecture that provides enough flexibility to change ideas without having to rewrite the system.

1.2 An outline of this thesis

Much work and many ideas have been devoted to the difficult task of providing a usable system that can serve both as a flexible prototype for research, and a sufficiently stable and fast test system for demonstrating our ideas at work. The resulting system is the GNU Image Finding Tool (GIFT). The GIFT’s design is based on the separation of a CBIRS system into several components. These components and their use are described in chapter 2.

The GIFT’s design is closely linked to the design of MRML, the multimedia retrieval markup language. We see this markup language as the key to flexibility in our context. The problem MRML is designed to solve, and MRML’s design are described in chapter 3.

Chapter 4 describes work on processing queries by example. Our approaches are based on the adaption of text retrieval techniques to image retrieval (this is first described in Squire et al’s paper about the Viper system [87]). We will describe the Viper approach, and we will justify this approach by comparing it to more traditional indexing techniques. Furthermore (in section 4.5) we adapt enhancements to this technique that are based on analysis of the correlation between features within the collection to the case of image retrieval.

More new work is presented in chapter 5. Here we develop an efficient method for similarity search on directed labeled graphs. This method is suitable for the integrated treatment of annotation and visual feature. To our knowledge this is one of the first approaches treating low-level multimedia features and semantical annotation data using the same formalism.

Chapter 6 describes and analyzes PicHunter, a browsing system invented by Cox et al [12] that is based on Bayesian learning and we present our contribution tracker: a system designed to follow the user’s changes of mind.

The work presented in chapters 4 through 6 resulted in the implementation of three query engines that can be used either separately, or together using the GIFT’s meta query capacities described in chapter 2.

During the work on tracker, it became apparent that comparing the performance of image browsers is an extremely challenging problem. We present a possible solution to this problem in chapter 7.2.5, along with a benchmarking harness framework, snakemeter that is an attempt to provide the flexibility needed for day-to-day work with a benchmarking tool.

1.3 Contributions in this thesis

Within this thesis we give both theoretical and practical contributions to the field of image retrieval.

MRML and the GIFT constitute a framework that enables fast development and fast demonstration of content based retrieval solutions. The programming overhead for moving from a working command line tool to a working demonstration that can be viewed from the world wide web is extremely small. The GIFT is freely downloadable at [21].

The GIFT framework achieves its purpose by providing services to query/retrieval engines, i.e. independent query-processing entities. Within the GIFT, query engines
1.3. CONTRIBUTIONS IN THIS THESIS

can either function independently or they can be used in combination to provide a new, more powerful query engine.

We designed and implemented most of GIFT's default retrieval engine Viper. To our knowledge Viper is the first image retrieval system that is using text retrieval techniques based on inverted files and probabilistic weighting for indexing low-level visual features. Further contributions are the adaptation of theoretical information retrieval results to the image retrieval case, and the formalization of the quantization process needed for inverted file indexing. Both these theoretical results motivate the feature description framework featurenet. Its usefulness is demonstrated in experiments.

Furthermore, we contribute an algorithm that is able to do fast similarity retrieval of graphs, capable of performing ranked query by example on graph-structured data.

This algorithm, as well as featurenet are included in the new query engine bothrops. Also this software package is a contribution of this thesis which will be released to the general public as part of the GIFT package.

We are presenting tracker, our extension to the PicHunter image browser. We demonstrate that our extension to the initial user model greatly improves performance if the user changes his query goals during the query process.

Finally, we present a benchmark for image browsers that simulates the semantic gap between the user's imagination and the low-level visual queries he can issue by another semantic gap: the semantic gap between textual annotation and low-level visual queries. Our method is an extension to schemes that use just categorization to generate ground truth data for testing, as we need ranking of images to have suitable ground truth for simulating a browser user.

Also this last part is demonstrated using a software package, the snakemeter benchmarking server.
Chapter 2

The architecture of the GNU Image Finding Tool

Throughout this thesis we are pursuing the goal of building a system with the following main characteristics:

We want a system that provides both reasonable desktop usability and a demonstration on the internet. Working on desktop scale ($O(10^4)$ images, as opposed to $O(10^9)$). While we are using our system for small collections only, we want our methods to scale well to large image collections. We want to use methods that carry over from images to semi-structured multimedia data, like e.g. HTML pages with graphics. We wanted to emphasize retrieval performance as opposed to retrieval/insert/delete performance. We assume that collections will be indexed rarely and queried very often after that.

Further, we want our system to have an architecture enabling several people to work on different parts of the project without having to discuss each and every change and we want an architecture enabling the collection and the sharing of interaction data. The central part of this flexible architecture is MRML, the Multimedia Retrieval Markup Language. Initially, MRML has been developed only for client-server communication, as described in the following section 2.1. However, it turned out that the flexibility was greatly enhanced by making MRML an integral part of the design, as we will describe in section 2.5.

The current version of the GIFT supports plug-ins. A plug-in is a dynamically linkable library that is found by its host program during runtime, and of whom only the interface is known to the host system. As a consequence, plug-ins can be added by copying a dynamically linkable library in a specified location, and the GIFT will find them without any need to change either source code or configuration of the GIFT. Plug-in developers need to know about the GIFT+plug-in interface, but not any implementation details of the GIFT.

2.1 Client-server communication layer

Many content-based image retrieval systems provide some form of client-server communication. Take, for example WebSEEk [82], BlobWorld [2], or IMEDIA [32]. They enable the user to connect via a WWW client (i.e. web browser), and to express queries, and to view the result of the queries on the browser.

In addition to the benefit in terms of visibility of the group’s work, such client/server communication is useful also for the development of the system itself: it automatically provides separation of GUI (graphical user interface) and server code, thus enforcing cleaner design.
In our system, we also followed the concept of client-server separation. However, instead of focusing on our own data model and query model, we developed an extensible protocol designed to make several such models coexist. The resulting client/server communication protocol is MRML, the Multimedia Retrieval Markup Language. Throughout this thesis we benefit from MRML. The basic ideas that lead to MRML, as well as the advantages of MRML will be described in the next chapter, chapter 3.

![Diagram of the GIFT system](image)

**Figure 2.1: A structural overview of the GIFT system.**

### 2.2 Session management

MRML has been developed with the idea of logging and learning in mind. In order to perform learning of user behavior, we want to know the name of the user, and we want to give him the possibility to choose between different sessions, in order to be able to give specialized services, as well as to collect user data by category. In the GIFT, session management currently is implemented in a non-persistent fashion. The GIFT server maintains sessions during runtime in memory. Clearly,
2.3 Query engine

Throughout this thesis, we will call query engine an entity within the GIFT that is capable of processing queries. As described in this section, the query engine is the combination of the implementation of a search method with the implementation of a suitable indexing structure.

Consider a browsing query in a stochastic image browser such as tracker, described in chapter 6. In tracker, the user receives overviews of the collection, each time rating some images from the overview, and sending them back as a query. Note that the user does not receive a best match to the query but rather a broad overview over the set of images still considered as potentially relevant for the user. The goal is to approach one target image in the collection, and ultimately, to find this target image.

At each step, the query engine has to maintain a state that expresses the probability that a given query image is the target image searched for by the user.

Take a different scenario: a long series of queries by example. You would like to extract behavior patterns from the user's feedback during runtime.

Figure 2.2: Within the GNU Image Finding Tool system, all functions that are likely to benefit from MRML's flexibility receive XML parse trees as parameters, and give back XML parse trees as results.
CHAPTER 2. ARCHITECTURE OF THE GIFT

Each of these scenarios need the query engine to maintain some state that describes the knowledge gained from the queries issued so far within the session. At the same time, the query engine will use data structures that will not be affected by the feedback. For the purposes of the query process they can be considered stateless.

This insight lead us to the separation of the query engine into two blocks: the so-called query processor (which remembers its state) and the stateless accessor of the indexing structure.

2.3.1 Query processor

The query processor holds all the information needed for processing the user's query. It holds the current state of the query process (e.g. past interactions), as well as it knows which indexing structure to access for processing the query.

On receiving a query, the query processor will parse the query, and translate it into a number of requests to the indexing structure accessor. The requests to the accessor will be generated using all information available to the query processor, e.g. the user's query itself, as well as the current state of the query processor. The results obtained from the indexing structure will then be assembled in order to produce a response, possibly again using previous interaction data.

2.3.2 Accessor of the indexing structure

As it has been said before, the accessor of the indexing structure provides function to access the indexing structure. These functions can be considered stateless.

In our view, the query processor/accessor separation not only forces a cleaner design, mainly we save initialization time and memory. The accessor can be shared among different incarnations of query processors. At no time we will need more than one accessor per indexed collection.

2.3.3 Two examples for query processor/accessor separation

Viper

Take, for example the Viper query processor, and the inverted file accessor. Viper (see chapter 4) processes image queries by example. Viper expects and processes queries that contain a list of positive and negative images (i.e. basic user-relevance-lists in MRML, as described in the next chapter). Viper's query processor will request from the accessor for each image a list containing features for this image.

The feature lists for all query images will be combined to a common feature list, which will be used as the basis to a sequence of queries to the accessor. We will call this list now qfl (query feature list). The query processor will now request for each feature \( \varphi \) contained in the qfl a weighted list of documents \( d \) from the accessor. The weights of \( \varphi \) and \( d \) will be combined by the query processor, yielding a score for each \( d \). Here again, we use informations about the features that are contained in the accessor.

The accessor for inverted files that was used throughout our work with Viper stores access information about the files used in main memory (Random Access Memory). While this is convenient for the implementation and for the performance, memory consumption can be high for large collections. 50000 images can take in the order of 20 megabytes of RAM, essentially URLs or file names of the images in the collection. Sharing inverted file accessors between different instances of Viper query processors thus saves a considerable amount of memory.
2.4. FEATURE EXTRACTION, INDEXING

tracker

While in Viper the query processor itself is stateless, tracker's query processor, as described in chapter 6 has to maintain a state over many queries. tracker's query processor receives queries that are syntactically equivalent to Viper's queries (a list of images with relevances), however, the semantics is different: tracker is an image browser that provides successive overviews of the image collection, allowing the user to express preferences by feedback and adjusting the overviews to these preferences. tracker needs to know previous feedback in order to provide a new overview.

For computing a new overview, tracker combines its current state with the new feedback given by the user. For this, tracker needs to know the inter-distances between all images that are part of the query. The tracker query processor retrieves these inter-distances from the distance matrix accessor. In the current implementation which was done for scientific purposes only, the distance matrices are simply stored on disk. However, distance matrix accessors with another underlying structure and the same interface are possible.

2.4 Feature extraction, indexing

As we have said before, our design is oriented towards fast retrieval. As a consequence, so far little efforts were made in building a framework for integrating the feature extraction with the retrieval program. Currently, the GIFT's feature extraction and indexing programs are completely or near-completely independent from the GIFT.

2.4.1 Examples for feature extraction and indexing

Viper

Viper's feature extraction uses still the initial command-line program gift-extract-features that was intended for use with UNIX shell scripts. gift-extract-features has been wrapped with a Perl script (gift-add-collection.pl) that automatically calls the feature extraction for a set of images, and then calls the inverted file generation. Finally, gift-add-collection will modify the GIFT's configuration files in order to make the new collections known to the GIFT. Currently, the GIFT needs to be restarted for taking these changes into account.

When Viper is queried using an image that is not part of the collection used in the current session, a Perl script is called by Viper that downloads the query image, and that calls gift-extract-features.

tracker

tracker needs a distance matrix (matrix of image inter-distances) for functioning. Such a matrix can be calculated using an indexed and well-configured Viper collection, by calling the GIFT using a special startup option. The user will have to modify the configuration file by hand.

Clearly there is much to be improved in terms of usability. Still, in combination with the rest of the framework, this system was simple and flexible enough for making it a good test platform.

2.5 MRML, plug-ins, and the GIFT: a walkthrough

All requests to the GIFT are formulated in MRML and received by the communication handler. This communication handler separates the communication
from the processing (i.e. query processing, but also configuration of query processors/accessors etc.). The communication handler analyzes the incoming request, and then calls the session manager. The session manager provides functions for building, and configuring queries, as well as functions that pass queries to a session.

The session handler knows a couple of helper classes that contain information about the system (e.g. where to find plug-ins, information about the collections used etc.). Communication handler, session manager and all related helper structures will be called kernel in the following.

We now will describe the main use cases of the GIFT.

### 2.5.1 Configuring a session

We want configuring a session within the GIFT to be both simple and flexible. The configuration process has to support MRML, and it should be flexible enough for bringing complex configuration data to query processor and accessor without changing the kernel. This is a necessary precondition for a successful plug-in mechanism.

During this thesis work, we experimented on quite a number of approaches. We finally settled with a very simple solution: query processors and accessors each receive an XML parse tree containing a fragment of an MRML configuration message.

So, on receiving the request for configuring the session s, the session-manager will first find s in the list of open sessions. Next, it will look at the configuration data received. These configuration data contain at least information which algorithm to use, and in general a more complex configuration document. The session manager will retrieve from the algorithm collection (a helper structure managing query engine configurations) the defaults d for the algorithm chosen, and it will augment c using d, implementing overloading (c overrides d).

This information will be used to build a query processor tree. During this process, the session manager (again through helper structures) will make use of the query processor factory in order to construct query processors of the right type.

As already noted, each query engine receives on construction an XML parse tree containing its configuration data. The query accessor will extract from these configuration data information on which collection to use, and will then open an accessor. Opening means here: requesting an accessor from the accessor administrator. The accessor administrator will either construct a new accessor using the accessor factory, or give back the reference to an already-open accessor.

### 2.5.2 Querying

For processing the query, the session manager will retrieve the appropriate session, and then feed the query to the session. The session will use its query processor tree for processing the query.

**Meta queries within the GIFT**

In the query processor tree, query processors be attached to other query processors, forming a tree of query processors. Most query processors, like Viper, bothrops\(^1\) or tracker, simply ignore their children in this tree (and for this reason, a sensible configuration will not assign children to them), however, currently there exists within the GIFT one query processor whose function is to be an inner node in the query tree: it dispatches each query it receives to all its children, and then merges the results sent back to its children by averaging for each result image the scores obtained by the children.

---

\(^1\)In the classification of biological species, *bothrops* is the latin name for a subgroup of the *viperidae*, the vipers.
2.6 Configuring the GIFT

Another example is the split screen query engine which does not merge the results, but sorts them by query engine. A possible use of this is shown in chapter 5. Currently, this is not part of the GIFT distribution.

Existing GIFT plug-ins

During this thesis we implemented several query engines that were used as plug-ins within the GIFT framework. We thus could verify the usefulness of our approach as a framework that lets multiple query engines coexist and which easily supports and incorporates new development.

- *Viper*, currently the default query engine within the GIFT, described in chapter 4

- *bothrops*, a new query engine which includes support for querying relations, described in chapter 5. As *bothrops* is inherently more powerful than *Viper*, eventually it will replace *Viper* within the GIFT.

- *tracker*, an image browser tracking the user’s goals, described in chapter 6. *tracker* is not part of the GIFT distribution, as we have not yet cleared if it infringes patents held by Cox et al. for PicHunter.

- A (nameless) simple hierarchy browser that enables browsing of fixed hierarchies.

- *viPerl*, a plug-in that interprets Perl scripts. This enables the use of Perl scripts as a query engine fully integrated with the GIFT. It is very useful for quickly testing new concepts.

2.6 Configuring the GIFT

The GIFT’s base configuration is done using one configuration file <algorithm>
gift-config.mrml. gift-config.mrml consists of fragments of MRML. It allows the specification of the defaults for all algorithms, and it allows specifying which image collections will be used by the system. The relation between configuration entry and the analysis in the program is simple to understand: A query processor will get as configuration data an XML parse tree which is comprised of the configuration given in gift-config.mrml overridden by the configuration message sent from the client: tags present in the configuration message sent from the client will replace the information given in the configuration file. For example, if the entry for the algorithm adefault in gift-config.mrml is

```xml
<algorithm>
    <algorithm-id>adefault</algorithm-id>
    <algorithm-type>adefault</algorithm-type>
    <algorithm-name>Classical IDF</algorithm-name>

    <collection-id>TSR500</collection-id>

    <cui-block-color-histogram>No</cui-block-color-histogram>
    <cui-block-color-blocks>No</cui-block-color-blocks>
    <cui-block-texture-histogram>No</cui-block-texture-histogram>
    <cui-block-texture-blocks>No</cui-block-texture-blocks>

    <cui-base-type>inverted_file</cui-base-type>
</algorithm>
```
cui-weighting-function="Classical IDF"
/>

And the client sends:

<algorithm
  algorithm-id="default"
  algorithm-type="default"
  collection-id="TSR2500"
  cui-block-texture-histogram="yes"
  cui-block-texture-blocks="yes"
/>

The query processor will get the configuration data.

<algorithm
  algorithm-id="default"
  algorithm-type="default"
  algorithm-name="Classical IDF"
  collection-id="TSR2500"
  cui-block-color-histogram="no"
  cui-block-color-blocks="no"
  cui-block-texture-histogram="yes"
  cui-block-texture-blocks="yes"
  cui-base-type="inverted_file"
  cui-weighting-function="Classical IDF"
/>

Of course, the configuration data is given to the query processor as a parse tree. The query processor does not need to know anything about XML parsing. More about the meaning of the MRML tags given here can be found in the next chapter, and in the developer documentation to the GIFT package.

An accessor will be configured using exactly the configuration data given in the configuration file. There is no need for overloading.

2.7 Summary and future work

We have given a quick tour of the architecture of the GNU Image-Finding Tool. We have described the main, and we have described how these are configured.

In addition to rendering the code more robust against attacks, in the future we would like to move towards a feature extraction framework which is as flexible as the query processing framework provided by the GIFT. bothrops and featurenet, described in chapter 5 are a step in this direction. We would also like to add persistent session management to the GIFT, thus enabling query engines to retrieve and store their state on startup and shutdown respectively.

Finally, we would like to carry over the meta query framework which is part of the GIFT towards distributing the system. We would like to be able to mix local query engines with query engines that access remote MRML-based retrieval systems. Similarly it would be imaginable to use the GIFT as an MRML-to-CORBA bridge. Some work has been undertaken by Dr. Christoph Giess to this end, however his code has been overtaken by a variety of developments.
Chapter 3

MRML, a protocol for communication in content-based retrieval systems

3.1 Examples

The Multimedia Retrieval Markup Language (MRML), is an XML-based markup language for encapsulation of query languages. It is designed for multi-paradigm image retrieval systems, and enabling them to send a “society of queries” to a “society of models”, receiving unified results. At the same time it permits evolution while maintaining a maximum of compatibility. This is important in the fast-paced multimedia retrieval community which experiments both with new interaction paradigms such as query-by-sketch, query-by-outline, and new query languages such as SQL [10] or MOQL [12].

During the last 18 months, MRML has been used intensively within the Viper project at the University of Geneva. It has shown its applicability as a communication protocol for web-based QBE of the GIFT system, as well as a tool for devising automatic benchmarks. Moreover, we experienced a strong gain in flexibility by the use of MRML which is described in the previous chapter.

The development of research can be described as moving in two directions:

- search for new, useful query and interaction paradigms;
- deeper research to improve the performance of systems that have adopted a given query paradigm.

The search for new better performance given a query paradigm has led to “clusters” of systems which are similar in their interaction with the user, and which give a certain set of interaction capabilities to the user. It is already visible that research will move towards systems which enable the user to formulate multi-paradigm queries in order to further improve results. As a consequence of the above, there is the need for a common mechanism for shipping multi-paradigm queries and their results. Such a mechanism ensures that the right query processor processes the right query. We also need for for each paradigm a common language, which allows to formulate queries according to the paradigm.

Fulfilling these needs would enable the different communities to share user interfaces. At present almost every group has its own interface suiting its purposes.
However, many interfaces are very much alike. Fulfilling these needs would also enable the different communities to share implementations without sharing the code. Comparing the resulting systems of different groups of a research domain would be easier, if one could make them accessible to outside scripts without publishing the actual code.

The query shipping–mechanism should be designed in a way that it does not constrain ongoing and future research regarding both the search for optimal query formulation for each paradigm and the research for new query paradigms for MMDB queries. In short, the query–shipping mechanism should separate the communication problem from the query formulation problem, letting research groups evolve freely to find good domain specific query formulation schemes.

MRML, as proposed in [34], provides such a shipping–mechanism. In addition to the requirements stated above it was designed for making the use of it as simple as possible, thus allowing groups with exotic development environments and little manpower to be able to use MRML.

MRML in the current version provides a complete solution for QBE in CBIRS, which was the initial use of MRML.

MRML is an XML based language for the communication between MMDB server and user interface. In this technical report we demonstrate the utility of the framework provided by MRML by giving the example how we use MRML in our Viper system.

The chapter is organized as follows:
First we describe the general framework with emphasis on extensibility. After this we give the first application of MRML which is the use in our CBIRS systems together with the SnakeCharmer interface written by Zoran Pečenovic.

3.2 The design of the MRML query shipping framework

Common positions always limit the freedom of the individual. However, in this case the design is easily extensible. In this section we propose a development strategy which will preserve the freedom of the individual research groups, while keeping the standard.

When designing MRML we were lead by the following goals:

**Extensibility** Our main concern was to provide a framework which permits independent growth of the products of different research groups (followed by periodical code merging).

**No preferred implementation language** We want to leave the developer the freedom of choice of the implementation language. A standard like this is unlikely to be adopted by the research community, if it works only with a given “mainstream” computing environment.

**Independence of third–party libraries** We want the use of the communication protocol to be as independent from third party libraries as possible. A group should be able to provide its own tools within finite time.

Our choice is to use an XML (eXtensible Markup Language) DTD (Document Type Definition – a grammar) for the specification of our communication protocol, together with specifications for the transmission of messages, and for extensions of the protocol.

When making this choice we saw mainly the alternatives of using EJB (Enterprise Java Beans), CORBA, and other methods of remote procedure calls. However, we feared strong links with languages (Java/EJB) or large program packages
3.2. THE DESIGN OF THE MRML QUERY SHIPPING FRAMEWORK

(CORBA). A further advantage of XML is that the use of an XML for communication directly implies a common human readable log file format of this communication.

The attractiveness of XML is further increased by the existence of free tools in numerous programming languages. XML has been designed explicitly for simplifying parser design. XML has to be parsable by deterministic parsers, thus it is simple to implement one’s own XML/MRML parsers.

3.2.1 The structure of XML and “graceful degradation”

The structure of XML is similar to that of HTML, which stems from their common ancestry, i.e. SGML (Standard Generalized Markup Language): an XML document can be seen as a tree of “elements” which themselves contain other elements. The content of each node of the document tree is a list of attribute-value pairs, as well as a sequence of nodes (possibly interleaved with text). This structure is encoded using so called “tags” for the elements. The “opening tag” of an element with type t and attribute anAttribute being set to x would be \(<t \text{ anAttribute}="x">\). The “closing tag” of an element t would be \(<</t>\). This free structure is constrained by a Document Type Definition (DTD) which is a grammar for the tree structure. The details can be found in [99].

Graceful degradation is the key to successful independent extension of MRML. The basic principles can be summarized as follows:

- servers and clients which do not recognize an XML element or attribute encountered in an MRML text should completely ignore its contents,
- extensions should be designed in such a way that all the standard information remains available to the generic MRML user (see examples in § 3.3). In other words, the modifications of the standard MRML should be as local as possible therefore not create global changes which, from the above would be dismissed by standard parsers (see the examples in section 3.3).

These principles provide guidelines for independent extensions of MRML. To avoid conflicts between differing extensions of MRML, we plan to maintain or promote a central database for the registration and documentation of MRML extensions. This would also facilitate the “translation” between user logs which contain extended MRML.

3.2.2 A walk through of MRML

MRML-based communications have the structure of a remote procedure call: the client connects to the server, sends a request, and stays connected to the server until the server breaks the connection. The server shuts down the connection after sending the MRML message which answers the request. This connectionless protocol has the advantage of easing the implementation of the server. To limit the performance loss caused by frequently reconnecting, it is possible to send several requests as part of a single MRML message. The extension of MRML to a protocol permitting the negotiation of a permanent connection is also planned. MRML, in its current specification (and implementation) state, supports the following features:

- request of a capability description from the server,
- selection of a data collection classified by query paradigm; it is possible to request collections which can be queried in a certain manner,
• selection and configuration of a query processor, also classified by query paradigm; MRML also permits the configuration of meta-queries during runtime,
• formulation of QBE queries,
• transmission of user interaction data.

The final feature reflects our strong belief that affective computing [70] will soon play a role in the field of content-based multimedia retrieval. MRML already supports this by allowing the logging of some user interaction data. In particular, this is the case for the history-forward and history-backward functionalities of the SnakeCharmer interface.

3.2.3 Logging onto a CBIR server

An MRML server listens on a port for MRML messages on a given TCP socket. When connecting, the client requests the basic properties of the server, and waits for an answer. Skipping standard XML headers, the MRML code looks like this:

```xml
<mrml>
  <get-server-properties />
</mrml>
```

The server then informs the client of its capabilities. This message is empty in the current version of MRML, but it allows for the extension of the protocol:

```xml
<mrml>
  <server-properties />
</mrml>
```

Using similar simple messages, the client can request a list of the collections available on the server, together with descriptions of the ways in which they can be queried.

The client can then open a session on the server, and configure it according to the needs of its user (interactive client) or its own needs (e.g. meta-query agents). The client can also request the algorithms which can be used with a given collection:

```xml
<mrml>
  <get-algorithms
      collection-id="collection-1" />
</mrml>
```

This request is answered by sending the corresponding list of algorithms. This handshaking mechanism allows both interactive clients and programs (such as meta-query agents or automatic benchmarkers) to obtain information describing the server.

In a similar simple manner, the client can open and close sessions for a user, and configure the algorithms chosen by the user. This enables multi-user servers and also on-the-fly learning by the query processor.

3.2.4 Interface configuration

The client can then request property sheet descriptions from the server. Different algorithms will have different relevant parameters which should be user-configurable (e.g. feature sets, speed vs. quality). Viper, for example, offers several weighting functions [75] and a variety of methods for, and levels of, pruning. All these parameters are irrelevant for CIRCUS. Thanks to MRML property sheets, the interface can adapt itself to these specific parameters. At the same time, MRML specifies the way the interface will turn these data into XML to send them back to the server. Here is short example of interface configuration:
<property-sheet
property-sheet-id="s1"

type="numeric"
numeric-from="1"
numeric-to="100"
numeric-step="1"

caption="\% features evaluated"

send-type="attribute"
send-name="cul-percentage-features" />

This specifies a display element which will allow the user to enter an attribute with the caption "\% of features evaluated". The values the user will be able to enter are integers between 1 and 100 inclusive. The value will be sent as an attribute e.g. cuiv-percentage-features="33". This mechanism allows the use of complex property sheets, which can send XML text containing multiple elements. The interested reader is referred to the appendix A for details.

3.2.5 Query Formulation

The query step is dependent on the query paradigms offered by the interface and the query engine. MRML currently includes only QBE, but it has been designed to be extensible to other paradigms.

A basic QBE query consists of a list of images and the corresponding relevance levels assigned to them by the user. In the following example, the user has marked two images, the image 1.jpg positive (user-relevance="1") and the image 2.jpg negative (user-relevance="-1"). All query images are referred to by their URLs.

<mrml session-id="1" transaction-id="44">
<query-step session-id="1"
result-size="30"
algorithm-id="algorithm-default">
<br user-relevance-list>
<br user-relevance-element
image-location="http://viper.unige.ch/1.jpg"
user-relevance="1"/>
<br user-relevance-element
image-location="http://viper.unige.ch/2.jpg"
user-relevance="-1"/>
</user-relevance-list>
</query-step>
</mrml>

The server will then return the retrieval result as a list of images, again represented by their URLs.

Queries can be grouped into transactions. This allows the formulation and logging of complex queries. This may be applied in systems which process a single query using a variety of algorithms, such as the split-screen version of tracker [55] or the system described by Lee et al. [39]. It is important in these cases to preserve in the logs the knowledge that two queries are logically related one to another.

3.3 Extending MRML

In order to demonstrate how easily MRML can be extended to other query paradigms, we give as an example QBE for images with user annotation. We assume that the
user is invited to associate textual comments with images he or she marks as relevant or irrelevant. Since a tag for this purpose does not yet exist in MRML, we add an attribute cui-user-annotation="..." to the element. The prefix cui- is added to avoid name clashes with extensions from other groups which use MRML.

```xml
<user-relevance-list>
  <user-relevance-element
    image-location="file://images/1.jpg"
    user-relevance="1"
    cui-user-annotation="Tropical fish"/>
</user-relevance-list>
```

It is important to note here that servers which do not recognize the cui-user-annotation attribute still can make use of the remaining information contained in the user-relevance-element element.

As an example of how not to extend MRML, we give an extension with the same semantics but which does not respect the principle of graceful degradation:

```xml
<user-relevance-list>
  <cui-user-relevance-element
    image-location="file://images/1.jpg"
    user-relevance="1"
    annotation="tropical fish"/>
</user-relevance-list>
```

Instead of adding an attribute to an existing MRML element (user-relevance-element), a new element was defined that contained the same kind of extension, namely cui-user-relevance-element. Consequently, servers which do not recognize this element will not be able to exploit any relevance information.

### 3.3.1 A region query extension for MRML

In the past, most CBIR systems concentrated on queries for complete images. One of the first systems to use regions was Blobworld [6]. Here, the user is presented with a segmented image. He/she can then specify which of the segments (blobs) is relevant to the query.

Here we give an example how this can be expressed in MRML, followed by an explanation:

```xml
<mrml session-id="1" transaction-id="44">
  <query-step session-id="1"
    resultsize="30"
    algorithm-id="algorithm-default">
    <user-relevance-list>
      <user-relevance-element
        image-location="http://v.unige.ch/banknote.jpg"
        user-relevance="1"/>
      <!-- interested in foreground -->
      <cui-relevance-region cui-region-start-x="21"
        cui-region-start-y="12"
        cui-region-outline="e216a137w216n137"
        cui-user-relevance="1"/>
      <!-- but not interested in background -->
      <cui-relevance-region cui-region-start-x="21"
        cui-region-start-y="12"
        cui-region-outline="e137w216n137w216"
        user-relevance="-1"/>
    </user-relevance-list>

    <user-relevance-element image-location="http://v.unige.ch/2.jpg"
      cui-user-relevance="-1"/>
  </query-step>
</mrml>
```
3.3. EXTENDING MRML

```xml
<query-step session-id="1" transaction-id="44">
  <query-step session-id="1"
    resultsize="30"
    algorithm-id="algorithm-default">
    <cui-prolog-query result-container="ResultList"
      cui-relative-weight="0.1">
      <![CDATA[ findAll(Image, contains(banknote, Image),
                  ResultList). ]]>
    </cui-prolog-query/>
    <user-relevance-list cui-relative-weight="0.9">
      <user-relevance-element image-location="http://v.unige.ch/banknote.jpg"
        user-relevance="1"/>
    </user-relevance-list>
  </query-step>
</query-step>
</mrml>
```

The example contains a QBE query augmented by segment information. The user-relevance-elements contain the usual information contained in a user-relevance-element plus, cui-relevance-regions. The benefit of this, is that clients that are not region-aware still receive and understand the main parts query (graceful degradation).

A cui-relevance-region contains either the region-id of a previously specified region, or a chain code [18] giving the outline of the region. Chain codes specify regions by giving a starting point and giving directions from this starting point. In the case of the first cui-relevance-region this is: 137 pixels south, 216 east, 137 north and 216 west, marking a rectangular region. Defining the region to the right of the moving direction as inside, we can also describe inverse segments as it is done in the second cui-relevance-region. Fig. 3.1 describes the banknote contained in the image as relevant, but the background as not relevant: requested are banknotes with a different background.

![Banknote Example](http://v.unige.ch/banknote.jpg)

**Figure 3.1:** Example of an MRML extension. The user has requested a banknote (marked white) with a background different from the one presented (marked black).

Classical query languages and MRML

Simply use a CDATA section and express the query using the query language chosen:

This example performs a QBE query using the banknote image shown in Fig. 3.1, but without regions. It also queries a base of prolog facts for images which contain
the Prolog atom banknote. The query results will be mixed, weighting the rank in the Prolog query with 0.1, and the rank obtained in the QBE query with 0.9. Of course, designers who want to share this use of MRML have to document the internals.

MRML and MPEG-7

MPEG-7, the multimedia content description interface, is a large collection of description schemes, aiming at describing a large range of multimedia data. Currently, MPEG-7 data descriptions are given as XML Schema (for a description of XML-Schema, see [101]). The XML Schema is considered as the replacement of the XML DTD (Document Type Definition), i.e. it specifies a grammar for the text. While the DTD is geared towards text markup, XML Schema is geared towards data transfer and allows the definition of type systems.

We did not choose XML-Schema as a basis for MRML as when we specified MRML, Schema-enabled XML parsers were not yet available. At the same time we wanted to keep the hurdle for XML-developers low. However, any text which contains XML-Schema is also well-formed XML. As a consequence, there is no problem in “piping” MPEG-7 multimedia content descriptions through MRML, for example in a query:

```xml
<mrml session-id="1" transaction-id="44">
  <query-step session-id="1"
    resultsize="30"
    algorithm-id="algorithm-default">
    <user-relevance-list>
      <user-relevance-element
        image-location="http://v.unige.ch/banknote.jpg"
        user-relevance="1"/>
      <mpeg-7>
        <!-- put some MPEG-7 text here -->
      </mpeg-7>
    </user-relevance-list>
  </query-step>
</mrml>
```

Again, a query processor able to process MPEG-7 will recognize the mpeg-7 tag and act accordingly. When not recognized, the tag is simply ignored. Using this method Leung and Tam’s proposal for the structured annotation DS [91] was demonstrated on the Geneva MPEG-7 meeting [52].

MRML and binary data

MRML’s preferred mechanism for transferring binary data is to send the URL where the data can be found. Binary data is then retrieved using the URL. As it is a primary goal of MRML to enable the sharing of logging data we suggest to transfer big chunks of data as follows.

**Binary data which stays constant** over several sessions (i.e. images and other media items contained in the queried collection) should be transferred using their URL, as described above. This keeps log files relatively small, yet data is accessible for everyone.

**Binary data which changes** during the query process (e.g. a file containing an example image for a QBE query which is not accessible by the web) should be
transferred using two attributes. One of the attributes should contain the base64-encoded binary data, the other one the corresponding MIME type.

However, in most cases, it is preferable to design proper extensions to MRML which provide the best accessibility and readability of the resulting logs.

3.4 State and future of implementations using MRML

3.4.1 Content-based image retrieval systems

The following content-based image retrieval systems are supporting or going to support MRML:

- the CIRCUS server framework [66] by the EPFL at Lausanne is being ported to MRML.
- the GNU Image-Finding Tool is currently the most complete implementation of MRML. It can be freely downloaded at [21].

As tools for developers who wish to integrate MRML support into their system with minimal efforts, there exists a minimal example MRML server written in Perl [51].

3.4.2 GUI Clients

Presently, the most complete implementation of MRML on the client side, is Zoran Pećenović’s JAVA applet Snake Charmer, which is also a GNU package. Except for the multiset property sheet element, the enforcement of subset size constraints, the visibility attribute (everything is popup), and the client side of enabling meta algorithms MRML is completely implemented in the SnakeCharmer client.

In addition to SnakeCharmer, Nicolas Chablon, a license student in our group implemented in PHP an MRML client that makes the GIFT more easily accessible via slow internet connections. Furthermore, we sidestep some limitations due to JAVA’s security system (applet and images need to be served by the same server).

Other Clients are under development under our guidance:

Deepika Sikri from the BITS University at Pilani, India, is working on the integration of an MRML client into the GNU Image Manipulation Tool. She is doing so as part of her Master’s project. The GNU Image Manipulation Tool [22] is currently the most popular piece of software of the GNU project.

Carsten Pfeiffer, a computer science student from Berlin, programs in his leisure time kmrml, a “KDE I/O slave” that makes MRML retrieval systems accessible by typing a simple URL within KDE’s web browser, Konqueror. KDE, the K desktop environment, is currently the most frequently used desktop environment on GNU/Linux systems [23].

3.4.3 Benchmarking harness

At the time of writing two prototypes capable driving an MRML-enabled benchmark that works over the internet are under development in the Viper group. They emphasize differing aspects of the task. One of these prototypes, snakemeter, designed and implemented by the author of this thesis is described in the chapter about browser Benchmarking 7.

These prototypes will help in choosing the design of the benchmarking harness for the international CBIRs benchmarking contest, the Benchathlon [3]. The Benchathlon is an international, interdisciplinary effort for rendering content-based image retrieval systems comparable. In it participate experts from the field of
content-based image retrieval, performance evaluation and information science. The Benchathlon is part of the SPIE Internet Imaging conference, chaired by Giordano Beretta and Raimondo Schettini. Stephane Marchand-Maillet is head of the Benchathlon initiative.

3.5 Future work in the specification of MRML

There are two main directions concerning further work on MRML and related goals.

1. Providing tools: In our opinion the best way of using the advantages created by MRML is to pool common tools which can be used and exchanged within the research community.

2. Enhancing MRML: it is already clear at the time of writing that MRML as is, very flexible, already very useful, but incomplete. We need to incorporate (at least): adding images to a collection, a more flexible region extension with clickable regions and allowing text in queries.

Here we are hoping for cooperation with working groups who are using these query techniques in their systems. We would like to make MRML a “living standard”, always keeping language specification and implementation date close together. Doubtlessly, MRML will benefit from its adoption as query shipping mechanism of the Benchathlon effort.
Chapter 4

Processing queries by example using inverted file indexes

As we have described in the introduction, Query By Example is the most basic query paradigm for content based retrieval systems: One or more images are fed to the system as positive or negative examples or list of images with attached real-valued positive or negative weights [60]. The CBIRS will learn from these images a suitable query on its internal representation.

The challenge in QBE is that the system usually has no additional information e.g. on which parts of the image are the most relevant, or which is the reason for the preference of a given image to another. As a consequence, the learning methods employed have to be very flexible. In the ideal case they should be able to discern from few examples which details of the image are important. This observation becomes interesting later in the discussion.

In this chapter we will describe which steps have to be taken to perform QBE in content–based image retrieval. This is related to the problem of (text–)information retrieval. We will argue that text-retrieval techniques have advantages in terms of flexibility and efficiency compared to other vector–based indexing techniques.

The main part of this chapter will be devoted to describing how to extend text retrieval techniques to index real–valued and discrete–valued vectors using inverted files.

In this chapter, we will describe a query engine based on inverted file indexing, Viper. Viper, initially implemented as proof-of-concept system, has been the basis of the Viper group’s work for more than two years. It implements only parts of the concepts described in this chapter, but evolutionary improvements have taken it to very high performance. bothrops, a new, more complex inverted file indexing framework that takes into account both the experience gained with Viper and the theoretical results presented in the following sections will be described in chapter 5.

The next section, section 4.1, explains the need for flexible indexing structures in image retrieval. In image retrieval we have very little a-priori knowledge about the quality of our distance measure, and we have to make up for this lack of knowledge by flexible indexing structures. Section 4.2 will explain one flexible indexing structure, the inverted file, and gives a derivation of a weighting scheme used for text retrieval using inverted files. Section 4.3 makes the step from text to features. Any type of multidimensional data can be indexed in an inverted file, and that section will describe how. Section 4.4 describes the features that are used in the Viper content-based image retrieval system. In contrast to text retrieval, when designing
a feature set for an image retrieval system we can already foresee that, and more importantly how terms in the index will be statistically dependent. Section 4.5 deals with such correlations. Lastly, section 4.6 describes Viper, a query engine using inverted files, implementing concepts up to the ones described in section 4.4. An implementation many of the concepts described in section 4.6 will be described in chapter 5, where we give an extension of inverted-file approaches to ranked retrieval of graph structures.

4.1 The need for flexible indexing structures

In content-based image retrieval, an image usually is interpreted as a point in a multidimensional space, the feature space. The first step in indexing an image is thus the translation of the image into a suitable vector representation. This is called feature extraction, as it involves the extraction and quantification of useful features in the image.

In most cases, the feature vector obtained for an image is very long, too long to be considered practical for frequent comparison of such feature vectors. PCA (principal components analysis, also known as Karhunen-Loeeve Transform, KLT) or similar techniques are used for identifying a subspace of the original feature space that captures most of the information contained in the original feature space.

The dimensionality-reduced vectors are then indexed using generic spatial indexing structures for vectors of real numbers. In low-dimensional spaces, these structures enable search with logarithmic time complexity: finding out if a given point in vector space is occupied, takes logarithmic time see [104] for the description of the VP-Tree (vantage-point tree), and its comparison to k-d-Trees.

The above-described class of image indexing techniques gives provably the best results given the initial feature vector is meaningful. However, in the context of image indexing, the feature vector corresponds rarely exactly to the needs of the user:

- There are no known distance measures that close the semantic gap in the context of unconstrained image collections.
- In particular, the user has not necessarily the same way of looking at images as the image processing specialist who devised the feature set [89],
- and the user’s similarity measure varies with his information need.

So, in general a CBIRS will need to interact with the user in order to find out his/her present information need. In other words, we will need to learn new distance measures during runtime. Changing the distance measure breaks down the advantages of the spatial indexing structure.

Berman et al. [4,5] propose to overcome this problem by proposing a large set of similarity measures to choose from. Feedback can be used for choosing between such models.

Another approach, first pursued by Ortega et al. [60] is the use of text retrieval techniques. Ortega et al propose in MARS the use of the boolean model for text retrieval for multimedia retrieval. In the Viper project, we went in the same direction, adapting Salton et al’s vector space model for text retrieval to the image retrieval problem [87].

Using text retrieval techniques is attractive, as text retrieval techniques have to cope with similar problems:

- As we will describe in the next section, text can be seen as a point in a multidimensional boolean vector space.
• Text retrieval is similarity retrieval on such a vector space.

• There is a semantic gap involved in query process, which exploits rather syntactic aspects of the query, and the desired result: a text that contains semantics interesting to the user.

In the last 30 years, the (text) information retrieval community has developed sophisticated techniques for coping with the high dimensionality of text, and for enabling effective and efficient user feedback. In the following section, we will describe the probabilistic model of inverted file text retrieval.

It is surprising that viewing low-level features as pseudo-texts is not yet a well-known technique. In their broad overview paper, Smeulders et al. [78] describe only the use of generic spatial indexing structures like k-d-trees [73], R-trees [27] and the like [35, 104].

4.2 Inverted files and probabilistic weighting

Currently, the most successful text retrieval systems are implicitly or explicitly probabilistic. They analyze the frequency of words in the texts in order to obtain a ranking of the documents given a query.

The systems that inspired Viper's query methods consider the text as a "bag of words". In this approach a collection of text documents (the file) is an unordered list of documents. Each document consists of an unordered bag of words, the words of the document. In a more formal notation, the file provides a mapping

\[
document \rightarrow \text{words in the document}
\]

The inverted file inverts the direction of the arrow. It provides a mapping from word to the documents that contain the word:

\[
\text{word} \rightarrow \text{documents containing the words}
\]

An inverted file contains for each word a list of documents containing this word.

In the following, we will use the symbol \( \varphi \) to designate a word, as later we will extend the approach to any kind of features. Documents \( D = \{ \varphi_1^d, \ldots, \varphi_n^d \} \) and queries \( Q = \{ \varphi_1^q, \ldots, \varphi_m^q \} \) are just sets of words. For a document \( D \) and a given query \( Q \) the query processor will calculate a score \( S(D|Q) \):

\[
S(D|Q) = \sum_{\varphi \in (Q \cap D)} f(\varphi, D, Q)
\]

(4.1)

where \( f \) is a function that determines a real-valued score for a word given the query and the document. Evidently, the weighting function determines strongly the performance of the query processor.

It becomes clear in the preceding equation 4.1 that for determining the score we need to look only at documents that have words in common with the query, so regarding every image is a waste of computing time. For calculating \( S(D_i|Q) \) for a given set of documents \( D = \{ D_1, \ldots, D_N \} \) it suffices to do the following:

• for each \( \varphi \in Q \) do

  – for each \( D_i \in D \) with \( \varphi \in D_i \cap Q \) do

  \( S(D_i|Q) \leftarrow S(D_i|Q) + f(\varphi, D_i, Q) \)
The inverted file contains precisely the information which documents \( D_i \) contain a given word \( \varphi \). Calculating the scores for each document amounts to reading \( |Q| \) lists and, calculating a value, and summing it to the score for each document in the list.

Till now the difficult problem of finding a good weighting function has been left out of the discussion. The rationale of a good weighting function can be described as follows:

- words that occur in fewer documents describe these documents better than words occurring in more documents.
- words occurring frequently in a given document/query are more important for the document/query.

Before going deeper into detail, we can state that for the described way of retrieving text, gross assumptions have been made.

- We lost all grammatical structure (“man bites dog” becomes the same as “dog bites man”).
- The usual weighting functions make the assumption of statistic independence, i.e. it is not taken into account that “illness” and “inflammation” occur more frequently together in documents than “fish” and “bicycle”.
- Related to the previous bullet-point, within our indexing structure we have lost semantic relation between words, like hypernym/hyponym relations (e.g. “animal” is a hypernym of it’s hypernym “dog”).

This being made clear, we will give a derivation of a weighting function that is motivated by statistic considerations. The derivation given here is a condensed version of what is given in van Rijsbergen’s book [95] about text retrieval:

We want to estimate the probability that a document \( D \) is relevant given a query \( Q \):

\[
P(\text{Degree of relevance of } D | D, Q) = P(\text{rel} | D, Q)
\]

We now assume that the document consists of words \( d_i \), and that words in \( D \) come from a set of all possible words \( \mathcal{D} \) (all words contained in at least one document of the database is a good and tractable approximation for \( \mathcal{D} \)).

\[
D = \{d_1, \ldots, d_n \} \subset \mathcal{D}
\]

We write \( \mathcal{D}^C \) for the words of \( \mathcal{D} \) that are not contained in \( D \): \( \mathcal{D}^C = \mathcal{D} - D \).

Using the Bayesian rule we can write:

\[
P(\text{rel} | D, Q) = \frac{P(D | \text{rel}, Q) P(\text{rel} | Q)}{P(D | Q)}
\]

with (because of \( P(a,b|\xi) = P(a|b,\xi) \cdot P(b|\xi) \))

\[
P(D | Q) = \int_0^1 P(D | \text{rel}, Q) \cdot P(\text{rel} | Q) \, d\text{rel}
\]

being the probability of observing \( D \) on a random basis given \( Q \). In the discrete case, this becomes

\[
= \sum_{\text{rel} = 0}^1 P(D | \text{rel}, Q) \cdot P(\text{rel} | Q)
\]
4.2. INVERTED FILES AND PROBABILISTIC WEIGHTING

Usually, we are interested in the case of two relevance levels: relevant or non-relevant. We want to minimize the expected error. If we assign \( \text{rel} = 0 \) to the case of the document being irrelevant, and \( \text{rel} = 1 \) to the case of the document being relevant, this minimization is achieved using the Bayesian decision rule:

\[
P(\text{rel} = 1|D, Q) > P(\text{rel} = 0|D, Q) \iff D \text{ considered relevant} \tag{4.8}
\]

and \( D \) is considered irrelevant otherwise.

If we now write

\[
p_i := P(\varphi_i \in D|\text{rel} = 1) \quad q_i := P(\varphi_i \in D|\text{rel} = 0)
\]

we can continue

\[
P(D|\text{rel} = 1) = \prod_{\varphi_i \in D} p_i \cdot \prod_{\varphi_i \not\in D} (1 - p_i) \tag{4.9}
\]

\[
P(D|\text{rel} = 0) = \prod_{\varphi_i \in D} q_i \cdot \prod_{\varphi_i \not\in D} (1 - q_i) \tag{4.10}
\]

if we assume probabilistic independence of the terms \( \varphi_i \). Applying the logarithm on both sides of the equation brings us to

\[
\log P(D|\text{rel} = 1) = \sum_{\varphi_i \in D} \log(p_i) + \sum_{\varphi_i \not\in D} \log(1 - p_i) \tag{4.11}
\]

\[
\log P(D|\text{rel} = 0) = \sum_{\varphi_i \in D} \log(q_i) + \sum_{\varphi_i \not\in D} \log(1 - q_i) \tag{4.12}
\]

Equation 4.4 tells us, how to calculate \( P(\text{rel}|D) \) from \( P(D|\text{rel}) \), equation 4.8 gives the decision rule we have to take in order to decide about the relevance of the document. Changing the form of this equation we obtain:

\[
\log P(D|\text{rel} = 0, Q) + \log P(\text{rel} = 0|Q) - \log P(D|Q) >
\]

\[
\log P(D|\text{rel} = 1, Q) + \log P(\text{rel} = 1|Q) - \log P(D|Q)
\]

\[
0 < \log P(D|\text{rel} = 1, Q) - \log P(D|rel = 0, Q) + \{\log P(\text{rel} = 1|Q) - \log P(\text{rel} = 0|Q)
\]

\[
+ \log P(D|Q) - \log P(D|Q)\}
\]

The part in \( \{\} \) stays the same for all documents. The other part becomes

\[
\log P(D|\text{rel} = 1) - \log P(D|\text{rel} = 0) = \sum_{\varphi_i \in D} \log \left(\frac{p_i(1 - q_i)}{q_i(1 - p_i)}\right) \tag{4.13}
\]

\[
+ \sum_{\varphi_i \in D} \log(1 - p_i) - \log(1 - q_i) \tag{4.14}
\]

Again, the second part, \( C \) does not depend on \( D \), but what is most important, is that the \( S(D|Q) \) depends only on the words that are part of both the query and the document (that means here we have the theoretical justification for the \( Q \cap D \) in equation 4.1). We calculated a weighted sum of the terms in the document.

We can derive the \( p_i \) and \( q_i \) by simple counting from a test collection of documents. The following table contains counts for the document element \( i \): How often \( \varphi_i \) occurs in relevant documents, how often in irrelevant documents etc.:
We can now express $S(D|Q)$ in terms of the values defined in the contingency table:

$$S(D|Q) = \sum_{\varphi_i \in D} \log \frac{r_i}{n_i - r_i} \cdot \frac{N - n_i - R + r_i}{R - r_i}$$

(4.15)

Under the assumption that $N, R \gg n_i, r_i$, this simplifies to

$$S(D|Q) \approx \sum_{\varphi_i \in D} \log \frac{N \cdot r_i}{R \cdot (n_i - r_i)}$$

(4.16)

For $n_i \gg r_i$ this converges to the well known idf (inverse document frequency) term weighting function.

$$S(D|Q) \approx \sum_{\varphi_i \in D} c \cdot \log \frac{1}{n_i}$$

(4.17)

The idf term weighting function has an interesting interpretation:

$S(D|Q)$ is proportional to the log probability of picking randomly a document $D'$ from the collection that has the same terms in common with $Q$ as $D$: $Q \cap D = Q \cap D'$.

**Multiplicity of terms:**

The above probabilistic formulation carries over to documents and queries containing multiple words. In this case, we treat a text as set of pairs

$$D = \{(\varphi_1, 1), \ldots, (\varphi_i, t_i), \ldots, (\varphi_n, 1), \ldots, (\varphi_n, t_n)\}$$

(4.18)

The second value of the pair representing the instance of $\varphi_n$ within the image. A document containing the word “cat” three times, will be expressed as a set that contains among others the pairs $(cat, 1)$, $(cat, 2)$ and $(cat, 3)$. These pairs can be treated in exactly the same fashion as words in the above derivation. Note that this means that each instance of a word in text and query is weighted differently. This, however, is deemed unnecessarily complex. Usually one assumes that the first instance of a word is weighted the same way as all other instances of the same word in the same document.

$$\text{weight}(\varphi, 1) = \text{weight}(\varphi, k) \quad \forall_{k=1}$$

(4.19)

Which using idf becomes the tf-idf weighting function

$$\text{weight}(\varphi) = t f_i \cdot n_i$$

(4.20)

Please note that, strictly speaking, this derivation asks from us to weight each term frequency separately, and to take into account the intersection of the term frequencies in document and query, although multiplication of term frequencies has been used successfully in text retrieval.

In this section, we have explained the concept of the inverted file. In addition to that we have given a derivation by van Rijsbergen [93] for probabilistic weighting of terms in the inverted file. We have extended this derivation to the case of multiple occurrences of the same word which is useful in the case of content-based image retrieval where features tend to arise multiple times within an image.
4.3. FROM TEXT TO FEATURES

Please note that throughout the information retrieval literature many term weighting approaches have been described and tested. At the current state of research it seems that in the absence of sufficient closed-form solutions, theory is often used rather as a base for empirical work and heuristics, than for providing a sufficient complete solution. Salton and Buckley [75] describe experiments where various weighting approaches have been used several combinations. Greiff and Ponte [25] justify some of these weighting functions using the maximum entropy approach.

4.3 From text to features

Glossary:
While in the previous section, there was only one type of features, in the following we have to deal with different roles of features. Depending on their role, the features will be designated using different notation. The notation used in the subsequent derivations is given in the following table:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I, J$</td>
<td>A collection $I$tem, i.e. a text document or a multimedia document</td>
</tr>
<tr>
<td>$\mathcal{S}$</td>
<td>The feature space</td>
</tr>
<tr>
<td>$\varphi$</td>
<td>The notation used above. Some discrete value, e.g. some word, or the label of a set. We will call this a <strong>binned feature</strong></td>
</tr>
<tr>
<td>$\text{set}(\varphi)$</td>
<td>$\text{set}(\varphi) \subset \mathcal{S}$. As it will turn out, the analogon to the word $\varphi$ in the above description of inverted file retrieval will be the label attached to a set. $\text{set}(\varphi)$ designates this set. We will speak of the <strong>bin of a feature</strong></td>
</tr>
<tr>
<td>$\varphi'$</td>
<td>Typically a value from the set from which the $\text{set}(\varphi)$ can be taken. In many popular cases $\varphi'$ is nothing else but a vector, hence the arrow. We will call these features <strong>unbinned features</strong></td>
</tr>
<tr>
<td>$\Phi_I$</td>
<td>The unbinned features of a collection item $I$.</td>
</tr>
<tr>
<td>$\Phi_B$</td>
<td>The binned features of a collection item.</td>
</tr>
<tr>
<td>$\varphi'(I)$</td>
<td>This is the value of continuous feature $\varphi'$ in collection item $I$</td>
</tr>
<tr>
<td>$\text{has}(I, \varphi)$</td>
<td>The collection item $I$ has binned feature $\varphi$</td>
</tr>
<tr>
<td>$x := \text{if}(\varphi, I)$</td>
<td>The binned feature $\varphi$ has the term frequency $x$ in collection item $I$.</td>
</tr>
</tbody>
</table>

The feature extraction into vectors usually corresponds to discretization, as the starting point is a multivariate function that maps points or point sets in color space to real values. Take for example the color histogram: We are interested in calculating the probability that a given pixel (picture element) $e$ has a color contained in a given region $R$ in color space:

$$P(c(e) \in R)$$  \hspace{1cm} (4.21)

As this, in general, is too complex to handle, color histograms are calculated by counting how many pixels within an image belong to one color from a discrete bin $\text{set}(\varphi_i)$, and then normalizing the sum of the entries to one. So, the color histogram entry contains the probability that a given pixel (picture element) $e$ of the image has a color $c$ from a given bin $\text{set}(\varphi_i)$, $i$ being a cardinal:

$$H(i) = P(c(e) \in \text{set}(\varphi_i))$$  \hspace{1cm} (4.22)
Calculating a histogram obliges us to choose suitable color bins \( \text{set}(\varphi) \). Usually one takes contiguous rectangular regions of a given feature space. From the histogram \( H(i) \) it is impossible to calculate \( P(c(c) \in \text{set}(\varphi_i)) \) unless (for some index set \( J \) \( B' = \bigcup_{i \in J} \text{set}(\varphi_i) \), i.e. \( \text{set}(\varphi_i) \) is the exact union of some of the existing bins.

In a similar way all the feature extraction techniques for indexing known to the author quantize in some fashion similar to the binning that has to be performed for histogramming. Still, the representation obtained is not suitable for indexing in inverted files, as every image will contain a value for each dimension of the feature vector. However inverted file indexing is most time efficient when each inverted file feature is contained only in a few images. So, what we do for obtaining a good inverted file feature set is to take a given vector representation and to quantize each of its vector components. This approach has been proven to be very successful using the \( \text{Viper} \) query engine.

An image \( I \) is seen as a set of points in a multi-dimensional space, the feature space \( S \):

\[
I \in 2^S
\]

The feature space is a Cartesian product of a number of sets, which we will call in the following dimensions. In fact, the dimensions of \( S \) usually can be grouped into groups of dimensions \( d \) with a common semantic meaning. These groups of dimensions we will call hyper-dimension, \( H \):

\[
S = \prod_{i=1}^{n} d_i = \prod_{k=1}^{o} H_k
\]

For example: a feature that captures the color contained in a rectangular region, will have the hyper-dimension "color" consisting of the real-valued dimensions of the color space chosen and the hyper-dimension "2d-space", consisting of the dimensions \( x \) and \( y \).

When looking for a distance function that relates two images \( I \) and \( J \), and which is easily represented using inverted files, we have to take into account that set intersection is the operation that suits best the inverted file indexing method. Thus, we are interested in useful functions, such that the distance function \( \Delta \) depends on the intersection of the feature sets of \( I \) and \( J \):

\[
\Delta(I, J) = f(I \cap J)
\]

Furthermore, we have to take into account that in a discrete-valued space with millions of values for each dimension (e.g. the single precision floating point representation) the probability for images to have the value in one dimension in common is very small:

\[
P(I \cap J \neq \emptyset) \approx 0
\]

As described above, the usual way of solving this problem is binning. That is, we store the image \( I \) rather a set of regions in feature space (in fact, all regions \( \text{set}(\varphi) \) that contain elements of \( I \)), together with the number of elements of \( I \) contained in \( \text{set}(\varphi) \). We call this set \( \Phi_I \):

\[
\Phi_I = \{ (\varphi_m, |\text{set}(\varphi_m) \cap I|) \mid \text{set}(\varphi_m) \subset S \land \text{set}(\varphi_m) \cap I \neq \emptyset \}
\]
4.3. FROM TEXT TO FEATURES

The \( \text{set}(\varphi_m) \) must be chosen such that \( \bigcup_{m\in M} \text{set}(\varphi_m) = \mathcal{S} \), and for tractability, we demand each \( \text{set}(\varphi_m) \) be a “rectangular” region in feature space. As you can see, again we use the letter \( \varphi \), because the \( \varphi \) are the features we are storing in the inverted file. We say, “Image \( I \) has or contains feature \( \varphi \)” if \( I \cap \text{set}(\varphi) \neq \emptyset \). We write \( \text{has}(I, \varphi) \), and we write \( \varphi(I) \) for expressing the feature value of \( \varphi \) for item \( I \) in continuous feature space.

Looking at our data representation, now an image is represented as a list of pairs \((\varphi_i, t_{i})\). The \( \varphi_i \), our features, are the identifiers for regions \( \text{set}(\varphi_i) \) in feature space, the \( t_i \) are the term frequencies, the number of times each feature is contained in a given image. This representation corresponds exactly to the way texts are represented in inverted files: the text is represented as an unordered list of pairs of words and their term frequencies.

We have limited ourselves to a class of distance measures between images that calculates inter-image distance as a function of the regions of feature space that are shared by the two images.

Imagine a collection consisting of the documents A through G which are points in a 2-dimensional feature space symbolized by the frame of the Fig. 4.1. The squares inside the frame in the image symbolize 2-dimensional bins. Assume that the user has queried for the document A. We can make the following observations:

- An \( L_2 \) (i.e, Euclidean) distance measure between the images would obtain the following ranking: A,G,E,C,D,B,F
- Using probabilistic weighting and only the innermost bin would retrieve: A,G
- Using probabilistic weighting and only the two innermost bins would retrieve: A,G,E.
- Using probabilistic weighting and the three innermost bins would retrieve: A,G,E,D;

Consider another collection, each item \( I \) of the collection containing one real-valued feature \( \varphi(I) \in [0;1] \). Assume the collection to be uniformly distributed over the feature space. Imagine that the probability that the user finds an item \( I \) relevant, \( P(I) \) is given by

\[
P(I) = a \cdot e^{\frac{(\varphi(I)-b)^2}{c}} + c \cdot e^{\frac{(\varphi(I)-d)^2}{e}}
\]
Assume an equidistant binning of the feature space using 128 bins \( \varphi_1 \), such that a collection item \( I \) has the term frequency \( tf(\varphi_1, I) \) for a given bin \( \varphi_1 \):

\[
  tf(\varphi_1, I) = \begin{cases} 
  1 & \varphi(I) \in \left[ \frac{1}{128}, \frac{127}{128} \right) \\
  0 & \text{otherwise}
\end{cases}
\]

In similar fashion we define binnings with 4 and 16 bins, \( \varphi_2 \) and \( \varphi_4 \).

We now assume that the user makes each relevant document part of the query, yielding a large set of images the query \( Q = \{I_1, \ldots, I_N\} \)

\[
  tf(\varphi_j, Q) = \frac{1}{N} \sum_{I \in Q} tf(\varphi_j, I)
\]

Assuming that the weight of each bin \( \varphi_j \) is given by \( tf \cdot idf = -tf \cdot \log df \), we obtain Fig. 4.2.

We see that we can approach the bi-modal distribution of relevance well, if we choose narrow bins. The approximation will be done by increasing weights for bins that occur that occur mostly in relevant items. Conversely, weights are reduced for bins that occur mostly in irrelevant items. By using average bins, we can obtain a tradeoff between too selective bins (for narrow bins, there will be only a relatively small number of items that contain the same bin), and too large bins (the true dependency between feature value and relevance will get lost). In the section 4.5 we will show how make both narrow and large bins coexist in the same feature set.

Please note that in a one dimensional, real-valued feature space, we would essentially have two alternatives to the above inverted-file based approximation: Fitting the relevance data to a model, or averaging. An example of fitting relevance data to a mono-modal Gaussian model is given in [59]. In the case of multi-modal relevance
4.4. The Feature Set Used in the Viper System

As far as we know, David Squire was the first to use inverted-file based text retrieval techniques in this way. For the proof-of-concept system Viper was intended to be, David Squire devised a feature set that contained both color and texture which is inspired by John R. Smith’s PhD thesis work [79].

In [86], the features were described as follows:

It is desirable that the color space used in an image retrieval system should be "perceptually" uniform, meaning that small changes in the color coordinates should correspond to small perceptual differences. The RGB space does not have this property [see Fig. 4.3]. The HSV color space offers improved perceptual uniformity, and is easier to compute and invert than systems such as CIE-LUV or CIE-LAB [80] [see Fig. 4.4].

Viper uses a palette of 166 colors, derived by uniformly quantizing the cylindrical HSV color space into 18 hues, 3 saturations, and 3 values. These are augmented by 4 grey levels. This choice of quantization means that more tolerance is given to changes in saturation and value, which is desirable since these channels can be effected by lighting conditions and viewpoint.

Two sets of features are then extracted from the quantized HSV image. The first is equivalent to a conventional color histogram, with the variation that bins containing zero pixels are discarded. There are thus 166 possible color histogram features, of which most images contain only about 40.

The second class of features represent local color properties of the image. The image is divided into square blocks at four scales, ranging from $16 \times 16$ through to $128 \times 128$. The mode color is calculated for each block. The occurrence of of a given color in a particular block is
treated as a binary feature. For our 256 × 256 images there are thus 56440 possible color block features, of which each image has 340.

If it were not for the weighting, one would expect from this feature extraction slightly less performance than from techniques that calculate local histograms over a range of scales, and then prune the histogram space. Tests of the retrieval performance of color histograms at varying degrees of reduction can be found in [79]. See Fig. 4.5 for a visualization of the color block features extracted.

In a similar way, a set of texture features was calculated:

Two dimensional Gabor filters have frequently been proposed as a framework for describing and understanding the orientation- and frequency-selective properties of neurons in the visual cortex [13], and banks of Gabor filters have often been applied to texture classification and segmentation [36, 102], as well as more general vision tasks [9, 33, 44]. We employ a bank of real, circularly symmetric Gabor filters, defined in the spatial domain by

$$f_{mn}(x, y) = \frac{1}{2\pi \sigma_m} e^{-\frac{x^2 + y^2}{2\sigma_m^2}} \cos(2\pi(u_0 x \cos \theta_n + u_0 y \sin \theta_n)),$$  (4.23)

where $m$ indexes the scales of the filters, and $n$ their orientations. The centare frequency of the filter is specified by $u_0$. The half peak radial
4.4. THE FEATURE SET USED IN THE VIPER SYSTEM

![Color cylinder diagram]

Figure 4.4: A quantized HSV (hue, saturation, value) color cylinder. In contrast to the RGB representation, only one perceptually meaningful dimension (hue, saturation, or value) changes when moving from one ball to the next. Among other reasons, this makes HSV and similar color spaces useful for color feature extraction that always involves quantization.

The bandwidth is given by

$$B_r = \log_2 \left( \frac{2\pi\sigma_m u_{0_m} + (2\ln 2)^{1/2}}{2\pi\sigma_m u_{0_m} - (2\ln 2)^{1/2}} \right)$$

(4.24)

(after [33]). $B_r$ is chosen to be 1 (i.e. a bandwidth of one octave), which then allows us to compute $\sigma_m$:

$$\sigma_m = \frac{3(2\ln 2)^{1/2}}{2\pi u_{0_m}}.$$  

(4.25)

The highest centre frequency is chosen as $u_{0_1} = \frac{0.5}{\pi \tan(1/3)} \approx 0.5$ so that it is within the discrete frequency domain. The centre frequency is halved at each change of scale, which implies that $\sigma$ is doubled (Equation 4.25). The orientation of the filters varies in steps of $\pi/4$, and three scales are used. These choices result in a bank of 12 filters which gives good coverage of the frequency domain, and little overlap between filters [33]. For practical implementation, filters are truncated at $3\sigma$, giving kernels of sizes $9 \times 9, 17 \times 17$ and $35 \times 35$.

The use of circularly symmetric filters means that Equation 4.24 is separable. The 2-D convolution can thus be computed using four 1-D convolutions, which reduces the number of computations required for an $N \times N$ kernel by a factor of order $N$ [33].
These filters are applied to the image, and the mean energy of each filter is computed for each 16 × 16 block in the image. The energy is then quantized into 10 bands, which were chosen by examining histograms of the filter energy at each pixel for 500 images. A feature is stored for each filter which has an energy in a band greater than the [0, 2) band. This means that there are 27648 possible such features for a 256 × 256 image, of which a given image may have at most 3072 (in practice this does not arise). Histograms of the mean filter outputs are also stored, giving a measure of the global texture characteristics of the image.

4.5 Coping with correlations

The feature weighting method derived above makes the assumption of conditional independence of all the words involved. We have already stated that this assumption is not justified. It is even less in the case of Viper’s feature set, as many between features are explicit in the process of the creation of the features: a color that is the most frequent color within one region will probably also be the most frequent color in one or more of its sub-regions. Unfortunately the correlation is not complete, we cannot say that an image having feature \( \varphi_a \) surely will have \( \varphi_b \).

4.5.1 Complete containment: hyponym/hyponym structure

We say a feature \( \varphi \) contains [completely] another feature \( \psi \) if an image \( I \) that has \( \psi \) (we write \( \text{has}(I, \psi) \)) has also \( \varphi \), i.e. \( \{ I | \text{has}(I, \psi) \} \subseteq \{ J | \text{has}(J, \varphi) \} \). We write

\[
\varphi \text{ contains } \psi
\]

holds. As a consequence, if \( \varphi \) contains \( \psi \), then the probability that an image \( I \) containing \( \psi \) is relevant (we write \( P(\text{rel}(I) | \text{has}(I, \psi)) \) ) is not altered by the fact
that we learn that $has(I, \varphi)$:

$$P(\text{rel}(I)|has(I, \psi)) = P(\text{rel}(I)|has(I, \psi), has(I, \varphi))$$ (4.27)

However, the weighting described in section 4.2 assumes that we can say

$$P(\text{rel}(I)|has(I, \psi), has(I, \varphi)) = P(\text{rel}(I)|has(I, \psi)) \cdot P(\text{rel}(I)|has(I, \varphi)))$$ (4.28)

This means, in log idf weighting the feature $\psi$ gets $\log(P(\text{rel}(I)|has(I, \varphi)))$ too much weight if we do not correct for correlation. In short, $\varphi$ should not be weighted at all, if $\psi$ is present, and Viper’s weighting scheme does not take that into account.

A chain of features containing each other

A complete perfect correction of the error is possible. Let us assume that there is a set of features $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ which contain each other ($\varphi_1$ contains ... contains $\varphi_n$). This means that

$$P(\text{rel}(I)|has(I, \varphi_1), \ldots, has(I, \varphi_i)) = P(\text{rel}(I)|has(I, \varphi_i))$$ (4.29)

while a tf.idf weighting scheme would weight things as if

$$P(\text{rel}(I)|has(I, \varphi_1), \ldots, has(I, \varphi_i)) = \prod_{j=1}^{k} P(\text{rel}(I)|has(I, \varphi_j))$$ (4.30)

We are assuming that the weighting function given $I$ is linear in the term frequency of feature $\varphi_i$

$$f(\varphi_i) \propto \theta_i$$ (4.31)

We are looking for a weighting function $f'(\varphi_i)$ such that

$$f(\varphi_i) = \sum_{j=1}^{i} f'(\varphi_j)$$ (4.32)

for any $\varphi_i \in \Phi$. The solution is fairly simple:

$$f'(\varphi_1) = f(\varphi_1)$$ (4.33)

$$f'(\varphi_{n+1}) = f(\varphi_n) - \sum_{i=1}^{n} f(\varphi_i)$$ (4.34)

$$f'(\varphi_{n+1}) = f(\varphi_n) - f(\varphi_{n-1})$$ (4.35)

As a consequence, an image which contains the features in $\Phi$ will get the weight:

$$\sum_{i=1}^{n} f'(\varphi_i) = f(\varphi_n) - f(\varphi_{n-1})$$

$$+ f(\varphi_{n-1}) - f(\varphi_{n-2})$$

$$\ldots$$

$$+ f(\varphi_2) - f(\varphi_1)$$

$$+ f(\varphi_1)$$

which is exactly what we wanted to achieve.

What do we gain? We have derived a probabilistically sound weighting function for features that are completely contained within other features. This, for example, is the case with hypernyms and hyponyms (“animal” is a hypernym of “cat”, “cat” is a hyponym of “animal”) within a thesaurus for words. Of course, it is possible to design feature sets, in which some features are the aggregation of others, creating a hypernym-hyponym alike structure between the features.
Extending to tree structures

Till now we have just treated the case of a chain \( \Phi = \{ \psi_1, \ldots, \psi_n \} \) of features such that \( \psi_1 \) contains \( \ldots \) contains \( \psi_n \). However, this is easily extensible to trees, if we assume the weighting function \( f(\varphi, t_f) \) to be linear in \( t_f \). For this task it helps to give an example of what we want to achieve:

Looking for documents which are related to the concept “limb”, a document which contains once the word “arm”, once the word “leg” and once the word “limb” contains three words describing the concept “limb”. So, querying for “limb” it should be weighted as high as a document containing three times the word “limb” as far as concepts are concerned.

The requirements elicited in this example are met by the following definition:

A feature \( \varphi \) contains a set of features \( \Psi = \{ \psi_1, \ldots, \psi_n \} \) iff for all \( 1 \leq i \leq n \) \( \varphi \) contains \( \psi_i \), and the term frequency of \( \varphi \), \( t_f^\varphi \), is equal to the sum of the term frequencies of the features in \( \Psi \):

\[
t_f^\varphi = \sum_{\psi \in \Psi} t_f^\psi \tag{4.36}
\]

4.5.2 Augmenting the index, not the query

We have been suggesting in the last sections to augment the index, instead of augmenting the query. This topic is important for the flexibility and the efficiency of the system, so it merits its own subsection.

Augmenting the index means: keep for each feature \( \varphi \) all the features \( \psi \) that contain \( \varphi \) in the index. Adjust the weighting scheme such that feature containment is taken into account.

Augmenting the query means:

- Remove from each document all \( \varphi(D) \) for which exists a \( \psi(D) \) and \( \varphi \) contains \( \psi \). That is: keep only the features that do not contain other features.
4.5. COPING WITH CORRELATIONS

- Add to the query for each ϕ(D) all ψ such that ϕ contains ψ.

This approach is attractive, as it leads to small inverted files (the number of features stored for a complete collection is minimized). The disadvantages are:

- that the query building process becomes more complex to implement
- and potentially queries can get very big (as a feature potentially can contain thousands of other features), thus inefficient.

The time for scoring a feature list depends mainly on its length (the number of (ϕ, tf) tuples read, however if the data resides on disk, each read takes some time that depends only on the storage medium, and not on the amount of data read: the seek time. Augmenting the query means augmenting the number of seeks. So for large databases, augmenting the query will perform best (as the number of tuples read determines the performance), however, for smaller size databases, the speed advantage will be negligible or it will even turn into a disadvantage with comparison to augmenting the document.

4.5.3 Partial containment

It is not always possible to design feature sets with solely correct probabilistic treatment of correlations in mind. As a result, such features are partially contained in each other. This case is more difficult. Van Rijsbergen [95] suggests the following approach:

As in general:

\[
P(\varphi_n, \ldots, \varphi_1) = P(\varphi_n | \varphi_{n-1}, \ldots, \varphi_1) \
\]

\[
\times P(\varphi_{n-1} | \varphi_{n-2}, \ldots, \varphi_1) \
\]

\[
\times \ldots \
\]

\[
\times P(\varphi_2 | \varphi_1)
\]

we can approach this by

\[
P(\varphi_n, \ldots, \varphi_1) \approx P(\varphi_{k_n} | \varphi_{k_{n-1}}) \
\]

\[
\times P(\varphi_{k_{n-1}} | \varphi_{k_{n-2}}) \
\]

\[
\times \ldots \
\]

\[
\times P(\varphi_k | \varphi_1)
\]

The \(k_i \in \{1, \ldots, i\}\), as well as the ordering of \(\Phi\) are chosen to minimize the error incurred by the approximation.

Explicit treatment of this formula necessitates looking at each document. This is infeasible. However, there is a memory-inefficient, time efficient partial solution. Assuming that

\[
P(\text{has}(I, \varphi) \land \text{has}(I, \psi)) \neq P(\text{has}(I, \varphi)) \cdot P(\text{has}(I, \psi))
\]  

(4.37)

we can add to each image containing both \(\varphi\) and \(\psi\) with the same term frequency \(tf\) the correction feature \(\omega\) that has the weight:

\[
f(\omega) = f(\varphi, tf) + f(\psi, tf) - f(\varphi \land \psi, tf)
\]  

(4.38)

For \(tf\) being different for both \(\varphi\) and \(\psi\) in a given image, this becomes

\[
f(\omega) = f(\varphi, \min(tf_{\varphi}, tf_{\psi})) + f(\psi, \min(tf_{\varphi}, tf_{\psi})) - f(\varphi \land \psi, \min(tf_{\varphi}, tf_{\psi}))(4.39)
\]
Now, finding an ordering of \( \{ \varphi_1, \ldots, \varphi_n \} \) such that \( P(\Phi) = \prod_{i=1}^{n-1} P(\varphi_{i+1}|\varphi_i) \) is infeasible to do during runtime. This would necessitate either calculating everything on the fly during retrieval (this would be much too slow, as one would have to look at document lists for each feature multiple times), or calculation of \( \mathcal{O}(\#\text{features}^2) \approx 10^6 \) potential helper features.

Rijsebergen suggests interpreting the feature set as a complete graph \( G = \{ \Phi, \Phi \times \Phi \} \), and to find a spanning tree \( T = \{ \Phi, T \subset \Phi \times \Phi \} \) such that the expected error incurred by using only one spanning tree for all occasions is minimal. The criterion is to find a spanning tree which leads to mutual information between features, i.e. to a maximum of the correction features, weighted by the probability of their occurrence. While classical algorithms for finding minimum weight (or maximum weight, for that matter) spanning trees exist, they assume the weights of all edges of a graph to be known beforehand. In our case, however, establishing the weights of all edges is computationally prohibitive, as it would require looking at \( \mathcal{O}(10^6) \) pairs of features.

Pruning this huge search space is relatively simple in the image processing case, as our knowledge about the feature set suggests possible correlations between image features. If this does not solve the problem at hand, the data mining literature suggests how to proceed. The publication [1] is only the starting point of a rich literature about finding association rules.

Pruning the number of features generated can be done by sorting the newly generated correction features by their weight and cutting of either at a minimum weight or a maximum number of features.

In this section, we have given derivations on how to weight features in the presence of correlation. For this derivation we defined the concept of containment of features. The general case, presented in section 4.5.3, is computationally complex during indexing, and has been known for quite some time [95]. The case of complete containment of features is specific to indexing in presence of taxonomies or visual features. While the case of of complete containment can be seen as a corollary of van Rijsebergen's work, the insight that this case is computationally simple to handle, seems rare or even new.

Examples on how feature containment affects feature weighting will be given in section 5.5 in the next chapter, when discussing the bothrops query engine.

### 4.6 Viper, a query engine using inverted files

During the last 2\(\frac{1}{2}\) years, the Viper query engine has been the center of the work of the Viper group at the centre universitaire d’informatique at the university of Geneva. Viper is the combination of a relatively simple feature set (described in section 4.4), and an inverted file-based query engine that entirely relies on the independence assumption i.e. that does not support containment-corrected weighting (as it was suggested in §4.5.1).

The initial version Viper was implemented end of October 1998 and ran first on a SUN SparcStation 5 under Solaris. It is now integrated into the GIFT and thus has the GIFT’s system requirements.

#### 4.6.1 Data access in Viper

Viper accesses data directly, i.e. by reading files using c standard input/output routines, commands, without passing by a general database management system. On small systems, this improves performance, as this makes the memory footprint of Viper small, and there is no additional communication overhead. For larger systems, however, we will expect advantages when basing such a query engine on a generic database management system (like e.g. PostgreSQL [72] or MySQL [56]),
as these systems are likely to provide disk access optimizations that would be hard to implement for a small research group. Another reason for using a database is that a database will provide access to data in a more defined and simpler to document way: much of the data structures becomes clear when querying the database using SQL's describe tables statement. Recently Cornelia Luoni, a license student implemented a prototypical MySQL-based accessor for Viper, which is not yet fully integrated into Viper. The lessons learned with this prototype are taken into account in bothrops, to be described in the next chapter.

Viper's usefulness as an experimental platform benefited greatly from the possibility to change the weighting function, and thus to follow some of the experiments described in [75].

The time efficiency of the Viper system was greatly improved by pruning [85]. The most successful pruning technique involved reducing the query to the terms with the highest weight. Details will be described in Henning Müller's thesis.

4.6.2 Separate normalization

As described in section, Viper's feature extraction generates a feature comprised of four subsets: color block (CB, see Fig. 4.7) and histogram (CH) features, as well as gabor (i.e. texture) block (GB) and histogram (GH) features. Initially, all of these features were weighted in an uniform fashion: each feature received $c \cdot \log idf$ weight. This was unsatisfactory. As it puts too much weight on the larger feature sets in relation to the smaller feature sets, Henning Müller suggested to give each feature group their own weight using a process we called separate normalization: instead of performing just one query, the query was separated into four parts, $Q = Q_{CH} \cup Q_{CB} \cup Q_{GH} \cup Q_{GB}$ according to the four feature groups, and instead of calculating for each document $D$ of the collection the score

$$S(D|Q) = \sum_{i \in (Q \cap D)} f(\varphi, D, Q)$$

We calculated:

$$S(D|Q) = \frac{1}{4} \sum_{Q_i \in \{Q_{CH}, Q_{CB}, Q_{GH}, Q_{GB}\}} S(D|Q_i)$$

As the scores for each Viper query are normalized, such that the query applied on itself earns the score $S(Q|Q) = 1$, the above procedure is tantamount to weighting the features of each of the four feature subsets differently (see Fig. 4.8). Applying this idea improved performance considerably. Why?

Intuitively, this can be explained by the mutual information between the feature groups. The fact that a given color $col$ is the most frequent color range in a given rectangular region, is highly correlated with a high-valued global color histogram entry for the same color $col$. Separate normalization makes it more probable that these relationships are weighted correctly — without explicitly taking them into account.

4.7 Summary

In this chapter we have shown how to index and weight any kind of multi-dimensional real-valued or discrete-valued data using inverted files and binning. We have given

---

Footnote: On the TSK500/2500 test collections, the performance of the color block feature alone is approximately equivalent to the performance obtained with each of the other feature groups alone. For this reason we omit the corresponding plots.
Figure 4.7: (TSR2500) Comparison of Viper (with only color block features activated) vs. Viper with all feature groups activated and separate normalisation.

Figure 4.8: (TSR2500) Comparison of Viper (with all feature groups activated, but no separate normalisation) vs. Viper with separate normalisation.
4.7. SUMMARY

a probabilistic derivation of some weighting functions (§4.2). We have raised some
issues like the tradeoff between too narrow and too selective bins (§4.3). We have
shown how to get around the choice of the right size of bins by defining hierarchies
of bins. Along with these suggestions, we have shown how to weight hierarchies of
bins in accordance with the derivations in section 4.2 (§4.5.1).

Finally, we have presented Viper, a query engine using inverted files, mainly
relying on an experimentally found good choice of bin sizes (without correcting
the partial containment of features), to reach a very good performance. Some
benchmarking results are given, using the snakemeter package (4.6).
Chapter 5

Towards unified retrieval of MPEG-7 and other structured multimedia document formats

In the last chapter we have discussed “flat” documents, which consist of a homogeneous set of features. Treating documents as flat documents is (up to a certain level) appropriate for texts and images, however, such an approach loses its justification when treating multimedia documents.

MPEG-7 is the example for multimedia documents that are heterogeneous. They usually contain multiple multimedia documents, related by a common content description. The MPEG-7 standard, the Multimedia Content Description Interface is a rich set of descriptors and description schemes that encompass both visual low level features (e.g. color histograms) and high level semantic descriptions (e.g. structured and linguistic description schemes).

As MPEG-7 is encoded in an XML document, the principal structure of an MPEG-7 document is a tree, the XML parse tree. However, the elements of the documents can be interconnected by edges, so conceptually, the structure of an MPEG-7 document is a graph.

During the design of MPEG-7, groups submitting suggestions for description schemes to the standardization process had to prove the usefulness of their descriptors and description schemes by describing a part of the standard MPEG-7 dataset using their own suggestions, and software implementing these suggestions. The result was the MPEG-7 experimental model (XM) software. While being very useful for its purpose, the XM software is very heterogeneous, and very large.

Along with many other scientists, we consider it a worthy goal of research to work towards indexing methods that are as multimedia as MPEG-7 (we want a query engine that can process any kind of multimedia data) and as extensible as MPEG-7. We want a query engine which for any new MPEG-7 descriptor requires only adjustment of parameters, instead of a new query algorithm. The following sections describe our steps into this direction. We will describe a query engine based on inverted file techniques that permits approximate retrieval of graphs with labeled edges and nodes. Furthermore, we will describe feautreenet, a method to describe and classify features.

MPEG-7 is destined to be immediately usable, and as such it comprises pre-defined tags for many application domains. In contrast, the techniques described
here are considered to be very abstract, we just took care that MPEG-7’s content

descriptions could be mapped onto our data structures and algorithms.

An MPEG-7 document can be seen as a complex network of interconnected

media items and their descriptions.

Conceptual graphs are a knowledge representation technique (along with asso-
cia
ciated graph manipulation techniques) that allow expression of real world concepts

in a way adapted to symbolic reasoning. As conceptual graphs are a very versatile

technique, several groups have worked on fast retrieval of conceptual graphs. In the

next section, we will describe such approach.

In section 5.3 we will describe an approach were we modify and reduce the con-

ceptual graph approach in order to reach a representation that guarantees efficient

querying while retaining the ability to query structured multimedia data.

5.1 Conceptual graphs: a primer

“A conceptual graph is a structure of concepts and conceptual relations where every

arc links a concept node and a conceptual relation node”, or more precisely (the
definitions are taken from the standard preparation document of the upcoming

central graph ISO standard [84]):

Definition of the conceptual graph

A conceptual graph g is a bipartite graph, which consists of two kinds of nodes called
concepts and conceptual relations.

- Every arc a of g is a pair a = (r, c), consisting of a conceptual relation r and

a concept c in g. The arc a is said to belong to r; it is said to link r to c; but

it does not belong to c.

- A conceptual graph g may have concepts that are not linked to any conceptual

relation; but every arc that belongs to any conceptual relation r in g must

link r to exactly one concept c in g.

- Three kinds of conceptual graphs have distinguished names:

  - The blank is an empty conceptual graph with no concepts, conceptual

    relations, or arcs.

  - The singleton is a conceptual graph that consists of a single concept, but

    no conceptual relations or arcs.

  - A star is a conceptual graph that consists of a single conceptual relation

    r, every arc that belongs to r, and every concept c that is linked by some

    arc (r, c) that belongs to r.

Concepts within the conceptual graph

Every concept has a concept type t and a referent r.

Concept types and type hierarchies

The conceptual graph formalism gives the possibility to define in addition to concept

types also concept hierarchies.
5.2. RETRIEVING CONCEPTUAL GRAPHS

Conceptual relations

Every conceptual relation $r$ has a relation type $t$ and a nonnegative integer $n$ called its valence.

- The number of arcs that belong to $r$ is equal to its valence $n$. A conceptual relation of valence $n$ is said to be $n$-adic, and its arcs are numbered from 1 to $n$.

- For every $n$-adic conceptual relation $r$, there is a sequence of $n$ concept types $t_1, \ldots, t_n$, called the signature of $r$. A 0-adic conceptual relation has no arcs, and its signature is empty.

- monadic, dyadic, triadic relations are synonymous to 1,2,3-adic relations, respectively.

In addition to these most basic concepts, the conceptual graph standard will also define $\lambda$-expressions, i.e. ways to define new relations from existing relations, and it allows the definition of contexts, i.e. to nest conceptual graphs into each other.

5.2 Related work about retrieval of conceptual graphs

In Iadh Ounis’ thesis work [62], a logical formalism is presented on how to use conceptual graphs for information retrieval. He presents a method operating on simple conceptual graphs (i.e. without contexts). Ounis uses inverted files to calculate projections of conceptual graphs on each other (i.e. it is verified if the query matches exactly to a subgraph of the matched document). To this end, he indexes relations: in the inverted file, he stores relations as tuples $(relation, c_1, \ldots, c_n)$, where the $c_1, \ldots, c_n$ are concepts. In addition to that he puts into the index a list of witnesses, i.e. identifiers for concept nodes that are part of a relation. In a way, one could describe his method as having another inverted file index within each document, for quickly obtaining all concept nodes that could match a part of a query. Querying thus becomes a series of set operations on lists of witnesses.

Using his method, Ounis obtains roughly a complexity which is $O(n^4)$, $n$ being the size of the graph used as a query. He does not specify how the number of documents $N$ influences complexity, but we can conjecture that this is roughly linear, yielding $O(n^4 \cdot N)$ complexity.

Ounis’ methods are similar to our methods presented below. The main differences are that Ounis treats the projection of graphs in a boolean way, i.e. there is no way to express an “almost-successful” projection. The other difference is that his strict approach allows him to treat the full problem in polynomial time: any graph on which the query graph can be projected will be found by his algorithm in polynomial time.

In our approach, to be presented in the next section, we want rather to relate nodes with attached features. In contrast to conceptual graphs, we cannot claim a concept to be defined for each node. As a consequence, we cannot guarantee a clear cutoff condition for match/non-match of nodes. All this is the case for un-annotated image segments, to give just one example where this approach might be useful. Please note that our approach is quite different from [30], which is based on histograms of line orientations of large planar graphs. The matching described below is more costly, and more precise.
5.3 Greedy matching for fast graph similarity retrieval

In the following, we will assume that the document can be described as a set of (document) nodes.

- nodes can contain each other
- nodes can contain flat features
- nodes can be related to each other using relation features

When matching scores will be assigned to each node in function of the features contained in query and node.

It is well known that graph matching is a difficult problem. Most graph problems are NP-hard or NP-complete. However, in our situation, we can make use of the structural knowledge of the graph, and our knowledge about what interests us in the graph to simplify the problem:

We are interested in finding documents where similar things have similar spatial or semantic relationships. We solve the problem by first looking for similar things (i.e. their representation as nodes) and we will check for their interrelationships afterwards. Evidently, the crucial step is the first one: without proper identification of similar nodes, the verification of the relation will be impossible. This means, we have to design our feature sets such that they guide the node identification process.

In the following algorithm description, we will call $Q$ the set of nodes that are part of the query, and $R$ the set of nodes that are part of the result.

- Do an inverted file query for each query node $v^q_i$, using only the flat features in $v^q_i$. (This process is described section 4.2 on page 51.)
- Make a list of triplets $(v^q_i, v^r_j, score_{ij})$ of the scores obtained in the query, listing the query nodes and the result nodes and the score of $v^r_j$ relative to $v^q_i$. Call this list $L$.
- Sort $L$ in descending order by its score.
- Let $T \subset Q \times R$. $T \leftarrow \emptyset$
- Traverse $L$.
  - Call the current element $c := (v^q_a, v^r_b, score)$.
  - If there is no pair $(v^q_a, v^r_b) \in T$ such that $v^q_a$ and $v^r_b$ are contained in the same common ancestor node $A$, set $T \leftarrow T \cup (v^q_a, v^r_b)$. We say these nodes have been matched.
- For each query relation feature $(id, v^q_i, v^q_j)$, i.e. a feature that is contained in at least one query node,
  - if there exists a $v^r_j \in T$ and a $(v^q_i, v^r_j) \in T$ such that there exists a result relation feature $(id, v^r_i, v^r_j)$, increase the score of $v^r_i$.
- The score of an ancestor node is given by the sum of the scores of its children. Count only the score of matched nodes.

Please note the small complexity of the process: the flat inverted file query for the nodes takes the same amount of time as would be taken without relation queries. This step takes $O(N \times |\Phi_{Q,1}|)$ time, where $\Phi_{Q,1}$ are the flat features of the query. Sorting the scores for the nodes takes $O(N \cdot \log(N))$ of time, $N$ being the number
of images, thus roughly proportional to the number of nodes. Sorting images we have to do in any case when performing ranked queries. We traverse the list of sorted score/node pairs, building a node-to-node hash $O(N)$, later we make use of the translation, using $O(|\Phi_{Q,2}|)$ time\textsuperscript{1}, $\Phi_{Q,2}$ being the set of relation features in the query. In multimedia applications, $|\Phi_{Q,2}| \ll |\Phi_{Q,1}|$, so the time complexity will be governed by the retrieval of nodes via low-level features.

Guiding the matching process: an example

Consider the case of the structured image annotation scheme used in chapter 7. Here, we have action and thing nodes: action nodes have a verb as well as a list of modifiers as flat features. Similarly, thing nodes are made of a noun which can be an actor (a living entity capable of action), or a thing (a dead entity incapable of action). As actions, things also can be modified using modifiers.

Nodes (things or actions) which participate in the same action or which are otherwise related are connected by relationship features.

To guide the node identification process, we also stored with each thing and each action modifiers that indicated participation in a relation. See the tables 7.1, and 7.2 for the list of original features and the features added to guide the matching process.

A query example

Consider a piece of structured annotation (as described in chapter 7). The following expresses "a man asks a man". The words preceded with the $\xi$ symbol are node names.

\begin{verbatim}
actor(man, $\xi$man1).
action(ask, $\xi$ask).
actor(man, $\xi$man2).
performs($\xi$man1, $\xi$ask).
isPerformedOn($\xi$ask, $\xi$man2).
\end{verbatim}

For simplicity, we will consider this annotation being matched to an annotation with identical structure:

\begin{verbatim}
actor(man, $\xi$man1').
action(ask, $\xi$ask').
actor(man, $\xi$man2').
performs($\xi$man1', $\xi$ask').
isPerformedOn($\xi$ask', $\xi$man2').
\end{verbatim}

Looking at this annotation, we would expect the system to match $\xi$man1 to $\xi$man1', $\xi$man2 to $\xi$man2', $\xi$ask to $\xi$ask'. We would expect so, because these nodes have the corresponding position in the structure.

However, our program sees the annotation as given in Fig. 5.1. Looking at the flat (i.e. non-relation) features of $\xi$man1 to $\xi$man2' (see Fig. 5.2), we can see that we have no means to decide how to match $\xi$man1 and $\xi$man2 to $\xi$man1' and $\xi$man2' without taking the relation features into account, i.e. without actually traversing the annotation graphs. This, however, we want to avoid for efficiency reasons. In order to make correct matching more probable, we guide the matching process by adding flat features to each node, as shown in Fig. 5.3. Now, just matching the flat features (Fig. 5.4) of the graph will suffice to get the correct match between the nodes: $\xi$man1 will be matched to $\xi$man1', $\xi$man2 to $\xi$man2' and $\xi$ask to $\xi$ask'.

\textsuperscript{1}This is the case if we are using hash maps for retrieval of the translation items, if we are using binary trees, this will become $O(|\Phi_{Q,2}| \cdot \log N)$.
5.3.1 Weighting features using node types

For weighting features we use the tf idf weighting. However, we adapt this weighting to the fact that in our context we retrieve nodes, and not entire documents. We also take into account that the interesting parameter for calculating the document frequency of a feature is not the number of documents containing the feature, but rather the number of nodes that could contain the same features. What we mean by this might be best explained using an example.

Imagine a collection of size \( N \), were each document \( D_i \) consists of two unrelated nodes, \( v^i_k \) and \( v^i_l \). To \( v^i_k \) a set of keywords is attached, to \( v^i_l \) a set of low level features. We will calculate the document frequency for a low level feature \( \phi^i_j \) as the number of nodes containing \( \phi^i_j \), divided by \( N \) (the number of nodes with the same type), and not divided by \( 2N \) (the total number of nodes).

A node can have several types, how types are assigned to nodes is currently is decided by the indexer.

5.3.2 Weighting relations

The problem of how to weight relations is non-trivial. While, as we showed above, a document-frequency related weight like tf idf is a reasonable and successful assumption for single words, choosing an equivalent weighting function for binary and ternary relations is less evident. In the tradition of text-retrieval weighting methods Ounis et al. suggest defining the term frequency of an arc of a conceptual graph as the number of times a relation \( r \) links two nodes which contain the same concept.

This appears reasonable, however, in our structure we have no equivalent to the concept node, as we want to be very flexible in the choice of the features we attach to a document node: think for example of a segment within an image (symbolized by a node) with visual features attached. We would like to be able to relate this concept either to other image segments (spatial relations) or to annotation (“this segment visually contains the items described in the following annotation”). In this case, attaching a concept to an image segment node would be useless, as all potentially matching nodes would be instances of the same concept (image segment).
5.4 featurenet: a unified way of describing and relating features

Till now, we have written only implicitly about the relation of features to features (as opposed of the relation of features to document nodes). Classic Viper does not even take feature relations into account at all, and has been very successful in spite of these shortcomings.

In the following, we will present a framework for describing features, and for putting them into relation with each other. We consider this as important for the following reasons:

- Describing features can enable coexistence of many independent feature extraction techniques in one system. In the ideal case, they can exist without a mediating central instance.

---

Figure 5.2: The same as Fig. 5.1, however, the relation features have been removed, and only the flat features remain

We are in need of a narrower concept than just “image segment”. However, treating each segment as unique is not adapted to the problem, either. What we need is a structure that is able to accommodate both the notion of weighting based on concepts and weighting related to some replacement of what concept does for us in weighting:

Within the weighting scheme, the concept has the function of enabling the estimation how many nodes can enter relations of similar interest.

As a solution, we suggest storing with each relation \( r_i \) of each graph a pair of keys: \( (k_i, (k_{i,1}, k_{i,2})) \). Then, the document frequency can be established as the number of nodes that contain the relation \( r_i \), associated with the same keys \( (k_{i,1}, k_{i,2}) \).

In the case of annotation, we can choose as keys\(^2\) the concepts used in the annotation. So, this reduces to the case described by Ounis et al. In the case of visual features of images, we suggest to do some un-supervised clustering of the image segment nodes of the database, then to label the clusters, and use these labels for the purpose of calculating the document frequency.

---

\(^2\)For simplicity, we assume the keys not to be numbers, but rather character strings, here.
• Describing features can enable *transcoding* of features: knowledge that feature $\varphi_A$ in the collection $A$ corresponds in collection $B$ to the disjunction of features $\varphi^1_B$ and $\varphi^2_B$, respectively, can enable re-indexing of images without re-extraction of features.

• Feature description can separate to some extent feature description from feature extraction. To a certain extent, we can describe for a feature how binning is to be performed, thus ensuring coherent behavior of tools on a by-database basis (as opposed to the less desirable by-program-version dependence).

• As we have seen, feature correlations are important for *correct weighting* of features and feature combinations. A suitable information structure can tell us about expected correlations between features, as well as the absence of correlations.

• Relating features can also help diminishing the negative effects of binning: Instead of querying just for features contained in the query image, we could
expand the query using features that are semantically or perceptually close to the features in the query image.

- Relating features can improve relevance feedback: from feedback, we might learn little about a single feature, however, we might learn more from features higher in the taxonomy. This corresponds, for example, to learning of generalized association rules, as described in [90].

The feature classification method described here is inspired by WordNet, hence the name featurenet. According to the Five Papers about WordNet [47]\(^3\), WordNet is an electronic thesaurus which was constructed for showing the robustness of a number of computer linguistic concepts. Within WordNet, words are associated to synonym sets, the so-called synsets. Synsets are related among each others via pointers. Each synset contains a set of pointers, each pointer having a specific semantic meaning, for example hypernym-relation, meronymic relation and similar relations (meronyms are words which have something to do with each other).

This architecture permits finding efficiently all meronyms, hypernyms etc. for a given word.

A key property of the word-net approach that the semantics of a word is mainly defined by

- the number of the synset, i.e. which words have the same meaning?
- as well as the relations to other words.

A word is uniquely described in WordNet by

- the word itself, e.g. "dog"
- the word kind: "noun"
- the sense number: "1"

These parameters define a unique synset to which the word belongs.

### 5.4.1 Describing features in featurenet

The main goal of describing features is again achieving independence. We want designers of feature sets to be independent from each other, the feature sets being able to coexist with as little intervention as possible from a central instance. Here we were influenced by MIME types [17] (content-types of the Multipurpose Internet Mail Extensions):

A content item in an e-mail which is not plain US-ASCII text, has to be encoded. For telling the e-mail client what to do with the content item, a type name is sent with the content item, as well as a description of the encoding. The e-mail client can now decide, if it knows this content type, and what to do with it. The types have the form of paths e.g. text/plain, and classify the item in a hierarchy. Knowing this hierarchy, finding a useful point to insert a new content type into the hierarchy is straightforward.

featurenet stores for each feature a structured description, where each item of the structured description corresponds to a MIME-type/value pair. The advantage of this is that firstly, name clashes are unlikely to occur. Secondly, if there is a name clash, a feature extractor can find out if the feature type that has been registered under the same MIME-like-type has the same or similar semantics. featurenet's representation is human readable and self describing if the type names are well-chosen.

\(^3\)You get the five papers under the same URL as one single PostScript(TM) file.
As we have said before, a feature (i.e. bin) is usually a labeled point subset in a multidimensional space. As we have shown in the examples in section 4.3, usually one will use hierarchies of bins for improving the results. In the same fashion, we will use several kinds of features on the same spatial regions of the image: color analysis and texture analysis, edge extraction, and so on.

On page 56 we described the feature space as a Cartesian product of sets, the\textit{ dimensions}. In many cases, these dimensions can be grouped into dimensions that have a similar semantics, the \textit{hyper-dimensions}. Feature space can be thus seen as a product of the spaces spanned by it’s \textit{hyper-dimensions}. For example $x, y$ both have spatial meaning, they are thus part of a \textit{hyper-dimension}.

We can write our feature space as a product of its \textit{hyper-dimensions}:

$$S = \prod_{i=1}^{n} d_i$$
$$= \prod_{k=1}^{o} H_k$$

\textit{Hyper-dimensions} and \textit{dimensions} are the building blocks of \texttt{featurenet}'s feature description. However, instead of “naming” the dimensions by indices, we name them using MIME-like paths.

In \texttt{featurenet}, a \textit{feature-type} (i.e. one feature space) is fully described by its name and a set of \textit{hyper-dimensions}.

$$\text{feature-type} := (\text{feature-type-name}, \text{arity}, \text{hyper-dimensions})$$

A hyper-dimension is fully described by a tuple:

$$\text{hyper-dimension} := (\text{hyper-dimension-path}, \text{dimensions})$$

- The \textit{hyper-dimension-path} is a human-readable MIME-like path describing and identifying the \textit{hyper-dimension}.
- \textit{dimensions} is a set of \textit{dimensions}.

A \textit{dimension} itself is described by dimension name and dimension type, as well as by the path to the hyper-dimension it belongs to. Please note that this means that there can be several dimensions with the same name, if they are part of differing hyper-dimensions:

$$\text{dimension} := (\text{hyper-dimension-path}, \text{dimension-name}, \text{dimension-type})$$

The \textit{dimension-type} can be one of

- \textit{unit interval}: the set of floating point interval comprised in $[0; 1]$
- \textit{real}: the set of floating point numbers
- \textit{integer}: the set of integer numbers
- \textit{string}: the set of character strings
5.4. FEATURENET

- **empty**: this dimension contains no value and is used as a “connector” for relations between features

A feature instance is a bin within a feature space. That is, a feature instance is a subset of the feature type, along, of course with the feature type name. Each feature instance is described by a set of hyper-dimension instances, each of the hyper-dimension instances are in turn given by a set of dimension instances. A dimension instance can be a coordinate, or an interval. The dimension instances themselves contain an element corresponding to their type, i.e. one of the following:

- **unit interval**: a floating point interval comprised within [0; 1]
- **real**: a floating point number
- **integer**: an integer number
- **string**: a character string

This structure is at the same time simple and flexible, and simple to implement. Please note that we might have suggested an unrestricted hierarchy of dimensions, hyper-dimensions, hyper-hyper-dimensions and so on. We would like to state that we have thought of this, but that we dropped this idea for sake of simplicity and efficiency of the implementation. More importantly, this three-level hierarchical structure makes comparison and matching of feature types simpler.

Expressing Viper's features in featurenet

Each Viper color feature is at the same time a color bin (expressed in HSV color space), and a spatial bin, (expressed in x and y Cartesian coordinates). This leads us to

- **Feature type**: viper-color
  - **hyper-dimension**: /color/hsv
    * **dimension**: h, **type**: unit interval
    * **dimension**: s, **type**: unit interval
    * **dimension**: v, **type**: unit interval
  - **hyper-dimension**: /spatial/2d/block
    * **dimension**: x, **type**: unit interval
    * **dimension**: y, **type**: unit interval

A possible instantiation of this feature type would be

- **Feature instance**: viper-color, **identifier**: 334
  - **hyper-dimension**: /color/hsv
    * **dimension**: h, **value**: [0.4; 0.5]
    * **dimension**: s, **value**: [0.2; 0.3]
    * **dimension**: v, **value**: [0.8; 0.9]
  - **hyper-dimension**: /spatial/2d/block
    * **dimension**: x, **value**: [0.5; 1.0]
    * **dimension**: y, **value**: [0.0; 0.5]

Feature net gives the possibility to do queries for feature points. This means, you give a continuous feature \( \varphi \), and featurenet retrieves all \( \varphi_r \), such that \( \varphi \in \text{set}(\varphi_r) \).
Mapping wordnet’s hypernyms to featurenet

When modeling WordNet using featurenet for the annotation described in chapter 7, we are just interested in a subset of WordNet’s capabilities. We are mainly interested in keeping the hypernym/hypernym dependencies, and making use of the synset mechanism in WordNet. The feature type which describes a word attached to a node (e.g. actor($\text{man}, \$\text{man}$)) would become:

- **Feature type**: `wm-node-word`
  - `hyper-dimension`: `/linguistic/word/wordnet`
    - `dimension`: word, type: string
    - `dimension`: type type: string
    - `dimension`: sense type: integer
  - `hyper-dimension`: `/wm/annotation/relation`
    - `dimension`: type, type: string

The feature instance for `actor($\text{man}, \$\text{man}$)` would become:

- **Feature instance**: `wm-node-word`, `identifier`: 22
  - `hyper-dimension`: `/linguistic/wordnet/word`
    - `dimension`: word, value: `man`
    - `dimension`: type value: `n`
    - `dimension`: sense value: `1`
  - `hyper-dimension`: `/wm/annotation/relation`
    - `dimension`: type, value: `actor`

Expressing: in WordNet, we are looking for the sense number 1 of the noun `man`. This `man` is an actor. Resolving the variable name `$\text{man}$` into a node identifier `etc` is not done by featurenet, and thus not treated here.

Similarly, we can express the synset belonging to the word `man` in `actor($\text{man}, \$\text{man}$)`, `actor(#11323, \$\text{man})`:

- **Feature instance**: `wm-node-synset`, `identifier`: 44
  - `hyper-dimension`: `/linguistic/wordnet/synset`
    - `dimension`: id, value: `11323`

5.4.2 Relating features

In the previous sections, we have described a human-readable fashion for describing bins for the purpose of indexing tuples of any conceivable kind in inverted file indexes. Now we will show how featurenet captures relations between features. Let us recapitulate:

- We want to encode containment relationships to adjust weighting
- We want to encode relationships to guide learning processes, as well as to represent their results, e.g. the said containment relationships.

featurenet’s relations connect a source (a feature instance, hyper dimension instance, or dimension instance) to a destination (a feature instance). Why three source types? This might become clear in three examples:
feature-instance-to-feature-instance relation

Consider two instances of the viper-color feature type defined above:

- **Feature instance**: viper-color, identifier: \( \varphi \)
  - **hyper-dimension**: /color/hsv
    * dimension: \( h \), value: \([0.0; 0.1]\)
    * dimension: \( s \), value: \([0.0; 0.1]\)
    * dimension: \( v \), value: \([0.0; 0.1]\)
  - **hyper-dimension**: /spatial/2d/block
    * dimension: \( x \), value: \([0.0; 0.1]\)
    * dimension: \( y \), value: \([0.0; 0.1]\)

- **Feature instance**: viper-color, identifier: \( \psi \)
  - **hyper-dimension**: /color/hsv
    * dimension: \( h \), value: \([0.0; 0.2]\)
    * dimension: \( s \), value: \([0.0; 0.2]\)
    * dimension: \( v \), value: \([0.0; 0.2]\)
  - **hyper-dimension**: /spatial/2d/block
    * dimension: \( x \), value: \([0.0; 0.2]\)
    * dimension: \( y \), value: \([0.0; 0.2]\)

The bin corresponding to \( \psi \) contains the bin defined by \( \varphi \) along every single dimension. As a consequence, \( \psi \) contains \( \varphi \), which we want to express in a feature relation:

\[(\varphi \rightarrow \psi, \text{is-contained-by})\]

hyper-dimension-to-feature-instance relation

Consider \( \varphi \) defined as above, and \( \xi \) defined as follows:

- **Feature instance**: viper-color, identifier: \( \xi \)
  - **hyper-dimension**: /color/hsv
    * dimension: \( h \), value: \([0.0; 0.1]\)
    * dimension: \( s \), value: \([0.0; 0.1]\)
    * dimension: \( v \), value: \([0.0; 0.1]\)
  - **hyper-dimension**: /spatial/2d/block
    * dimension: \( x \), value: \([0.0; 0.2]\)
    * dimension: \( y \), value: \([0.0; 0.2]\)

Along the /color/hsv hyper dimension, \( \varphi \) and \( \xi \) have identical bins. However along the spatial hyper dimension /spatial/2d/block, \( \xi \) strictly contains \( \varphi \):

\[(\varphi : \text{spatial}/2d/block \rightarrow \xi, \text{is-contained-by})\]

In an analog way, we can construct an example where a similarity-dimension might be useful as a source type of a featurenet feature relation.
Another example is relating the WordNet word with its synset (see the previous section for the featurenet description of 22 and 44):

\[(22:/\text{linguistic/wordnet/word} \rightarrow 44, \text{aggregation})\]

The relation type aggregation expresses that 22 contains 44, and that still, a query for a feature point will just yield feature 22 (and not the list \((22, 44)\)) as result. We then have to request all features that are derived from 22 by aggregation.

Feature relation types

In the current implementation, featurenet knows 4 relation types:

- **is-contained-by**: the destination feature contains the source feature. This usually means that for all dimensions the bins of the destination are at least the same size as of the source.

- **is-partially-contained-by**: the destination feature $\psi$ partially contains the source feature $\varphi$, typically $P(\psi, \varphi) > P(\psi) \cdot P(\varphi)$. Usually this relationship is due either to partial containment of $\text{set}(\psi)$ and $\text{set}(\varphi)$ or to neighborhood of these sets.

- **excludes**: source and destination cannot occur in the same document.

- **aggregates-to**: This case is special with respect to the three previous cases, in that the destination feature does not have a bin. It is derived by aggregation from other features. This feature relationship has mainly its use for creating features that correspond to non-contiguous regions in feature space.

Additional dimensions vs. named relations

It might be surprising that we did not choose named relations, and chose to define "empty" dimensions of a feature instead (see the description of feature dimensions at page 81). We considered that the possibility to relate a feature $\varphi$ to a feature $\psi$ using a containment relation with type $t$ was a property of both $\varphi$ and $\psi$. Defining most of the properties (except for the containment) rather by the dimension of the relation source than by a type attribute of the relation reflects the view that which relations are useful depends on the $\varphi$ and $\psi$.

To illustrate our view: there cannot be a synonym relationship between wavelet coefficients. Synonyms are words with different sound and spelling but similar meaning. (Take, for example the synonyms for one sense of the word man: *valet, valet de chambre, gentleman, gentleman's gentleman, man*). Wavelet coefficients are not words, and have no semantic meaning all by themselves, so there cannot be any synonym relation amongst them.

5.4.3 Using featurenet

The usefulness of featurenet can be best appreciated in a usage scenario. For indexing a collection of images using Viper-alike features we have to take the following steps:

First we have to prepare featurenet for usage. All these steps usually will be done by one program:

- Define the feature types used in the collection (if not done already)
- If this has not already happened: populate the database with feature instances (i.e. the bins we want to use during indexing), and feature relations.
• Instruct `featurenet` to cache binning data relative to the feature types you will use (e.g. `viper-color`).

• If possible: store binning code.

The feature extraction code then has to take the following steps:

• If possible: retrieve binning code from `featurenet`.

• For each image
  
  - extract the features
  
  - bin the features using binning code or a query on `featurenet`
  
  - create new document nodes where needed
  
  - add extracted features to the document node where needed

Please note: all the information about how to bin features resides in the repository of `featurenet`. This is a huge advantage, as it allows adapting the binning to the database (e.g. depending on the selectivity needed, or complexity constraints) without changing the code. More importantly consistency between the features generated by the feature extraction method and the feature description is assured.

The binning code takes into account that in the case of visual features, the feature instances usually are generated in a predefined order that permits calculating a feature instance that corresponds to a point in feature space, instead of looking the bin up in `featurenet`'s database. Evidently, calculating the bin from the point can generate huge efficiency gains. We experienced a speedup of more than a factor of 10 in our current implementation, in which the binning code is realized as a short Perl script that is interpreted within `featurenet`.

In the case of Viper's features or most other visual low level features, we know in advance how many features, and which features are used by the system. However, if we consider the case of unconstrained vocabulary annotation, it is simply inefficient to add a feature description for each word of the English language. Here we will add a feature each time the needed feature does not occur in our `featurenet` instantiation.

The `featurenet` feature server

`featurenet` is implemented in c++, and currently it is built as a GIFT plug-in, i.e. a dynamically linkable library with some standard functions to establish the link to the GIFT. As such a library, it can be linked also to other c or c++ programs. In order to provide a simple link also for scripting languages we implemented also a server that receives `featurenet` requests (in XML) via TCP/IP, parses them, executes them and sends the result back as XML. We successfully employed this system for indexing the structured annotation described in chapter 7.2.2 using `featurenet`, PERL and WordNet. Fig. 5.5 depicts the structure of the feature server.

5.5 bothrops, a query engine for structured queries

While Viper was rather intended as a proof of concept, bothrops was designed to make use of all of the benefits of inverted file indexing. bothrops uses `featurenet` for feature administration, and thus enables coexistence of many kinds of features. The following table summarizes bothrops' virtues with respect to Viper's.

As we can see in figure 5.7, bothrops uses as an accessor a near-complete `featurenet` feature server. Currently we make little use of the flexibility gained from

\footnote{It is planned to do groups and subgroups in IP-number like fashion: each group/subgroup is assigned a number of bits in the feature group number}
Figure 5.5: A featurenet based feature server that provides services needed by a feature extraction method. The feature extraction method can add new features and new feature relations. Furthermore, it can request feature id’s for given feature net feature descriptions. These feature id’s can be used for adding creating document nodes and attaching features to these document nodes.
### 5.5. **BOTHROPS, A QUERY ENGINE FOR STRUCTURED QUERIES**

<table>
<thead>
<tr>
<th></th>
<th><strong>bothrobs</strong></th>
<th><strong>Viper</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>feature relations</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>feature groups</td>
<td>named</td>
<td>integer$^4$</td>
</tr>
<tr>
<td>feature description</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>relating sub documents</td>
<td>partial</td>
<td>no</td>
</tr>
<tr>
<td>relation queries</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>correct weighting of relation queries</td>
<td>in preparation</td>
<td>not applicable</td>
</tr>
<tr>
<td>containment-corrected weighting</td>
<td>for complete containment only</td>
<td>no</td>
</tr>
<tr>
<td>multiplicity treatment</td>
<td>intersection</td>
<td>multiplication or intersection depending on feature group</td>
</tr>
<tr>
<td>relevance feedback</td>
<td>combining results</td>
<td>combining query</td>
</tr>
<tr>
<td>data access</td>
<td>MySQL</td>
<td>file system</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th><strong>bothrobs</strong></th>
<th><strong>Viper</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: *This table compares bothrobs and Viper’s main features*

This, except for doing containment corrected weighting (according to section 4.5). However, the *featurenet*-based architecture will enable future versions of bothrobs to use the feature relation information for improved feedback techniques.

The fact that bothrobs accesses data using a relational database management system not only provides flexibility for extension, but also eases writing programs that make statistics about features or change the a-priory weighting of features. The following SQL statement generates a non-normalized histogram over the document frequencies of all features in a given database:

```sql
SELECT
  floor(fi_document_frequency/100),
  count(floor(fi_document_frequency/100))
TO FILE '/tmp/histo.dat'
FROM feature_instance
GROUP BY floor(fi_document_frequency/100)
```

Plotting the file `/tmp/histo.dat` yields figure 5.6. Assigning a given pre-weight for all features of a certain type, or with a given dimension value takes only little more work.

### 5.5.1 Tests

**TSR500**

For first tests with bothrobs, we chose to index the TSR500 collection, a collection containing 500 images, provided by the TSR (*Televisien Suisse Romande*). This collection has already been used for the experiments described in [87], as well as in [86].

We prepared a feature set derived from Viper’s color features: we rescaled the image to 256 x 256 pixels, and we calculated the mode (i.e. most frequent) color for each 16 x 16 pixel block. In *featurenet* notation

- *Feature instance*: viper–color
- hyper-dimension: /color/hsv
  * dimension: h, value: \([0, \frac{1}{18}), [\frac{1}{18}, \frac{1}{9}), \ldots, [\frac{17}{18}, 1)\)
  * dimension: s, value: \([0, \frac{1}{2}), [\frac{1}{2}, \frac{1}{3}), [\frac{2}{3}, 1)\)
  * dimension: v, value: \([0, \frac{1}{2}), [\frac{1}{2}, \frac{1}{3}), [\frac{2}{3}, 1)\)

- hyper-dimension: /spatial/2d/block
  * dimension: x, value: \([0, \frac{1}{18}), [\frac{1}{18}, \frac{1}{9}), \ldots, [\frac{17}{18}, 1)\)
  * dimension: y, value: \([0, \frac{1}{18}), [\frac{1}{18}, \frac{1}{9}), \ldots, [\frac{17}{18}, 1)\)

To obtain a softer degradation of the matching function, we added features that contain 4 features each along the /spatial/2d/block hyper dimension.

- Feature instance: viper-color
  - hyper-dimension: /color/hsv [as before]
  - hyper-dimension: /spatial/2d/block
    * dimension: x, value: \([0, \frac{1}{2}), [\frac{1}{2}, \frac{1}{3}), [\frac{2}{3}, 1)\)
    * dimension: y, value: \([0, \frac{1}{2}), [\frac{1}{2}, \frac{1}{3}), [\frac{2}{3}, 1)\)

We repeated the process up to the features:

- Feature instance: viper-color
  - hyper-dimension: /color/hsv [as before]
  - hyper-dimension: /spatial/2d/block
Figure 5.7: The *bothrops* query engine within the GIFT.
As term frequency for each feature we chose the number of points in a given color, storing only the feature corresponding to the most frequent color range in a given $16 \times 16$ block. The resulting feature set corresponds to Viper's color block feature set, except for the fact that containing features contain completely their descendants, and term frequencies correspond to pixel frequencies.

Surprisingly, in our test collection, the latter degraded performance. We could increase performance by storing 1 as term frequency for every most frequent color in a $16 \times 16$ block. The larger block features would be assigned the number of $16 \times 16$ blocks containing the same color as mode color. The performance of these methods is compared in Fig. 5.8.

The explanation for these results might be found in [97]. Here Vertan and Boujemaa report that calculating a weighted histogram improves performance. Instead of plain counting, each pixel is weighted by its activity. In our experiment described above, the technique performed best that assigns more weight to pixels that would also be considered very active in Vertan's and Boujemaa's publication.

Our results are quite surprising in that the theoretically best initially does not lead to improved performance with respect to the original Viper color block feature set. However, when performing feedback, the new feature set with its associated intersection-based weighting method method gives better results over quite an extent of the precision-recall curve.

These results indicate that the tradeoffs when designing a feature set are very subtle. In our case, the new feature set captured less information (the information which color is the most frequent color in any of the larger blocks gets lost in the process), however, the remaining information was processed in a theoretically more correct manner.

**TSR2500**

We repeated some of the tests with a collection of 2500 images from the (Télévision Suisse Romande). We used for these experiments relevance data collected earlier (for [85]) from the members of the Viper group.

Again, we used *snakmeter* for the generation of several precision-recall graphs which then were summarized into several plots:

bothrops' color block features performed here better than Viper's color block features (Fig. 5.10) and Viper with all feature groups activated (but without separate normalization, Fig. 5.11). However, bothrops is clearly outperformed by Viper when all of Viper's feature sets are activated and normalized separately (see section 4.6.2, page 67 for an explanation of separate normalization and Fig. 5.12 for the comparison plot).

We also repeated the comparison between the performance of corrected and uncorrected weighting in bothrops (see section 4.5, page 62 for an explanation of the weighting correction). In contrast to the results obtained with the TSR500 collection, we obtained higher performance using the corrected weighting, as is shown in figure 5.13. While it appears reasonable that statistically better weighting performs better when the collection is larger, we still find it too early to conclude that this weighting method will necessarily perform better for large collections.

Having obtained slightly contradictory results using the same benchmark on two collections, we are looking forward to the Benchlothon contest for more conclusive results.
5.6 Conclusion

Based on the experiences made with Viper, we have proposed a new inverted file indexing and querying framework that unites standard “flat” inverted file queries, with similarity queries on graphs, and which provides self-documenting feature descriptions as well as containment-corrected weighting.

Bothrops is built on the use of a relational database system, which increases ease of use with respect to the file-based Viper.

We see building this framework as an achievement in itself. Besides the increase of use by the use of an relational database system, we hope to use Bothrops as a tool for the investigation in how much and in which situations the theoretical results about containment corrected weighting obtained in chapter 4.5 improve performance, as well as for designing feature sets that take advantage of the statistically corrected weighting methods.

Current benchmarking results are very encouraging. Our new weighting method improved performance on the larger of the two test collections, which is in good agreement with the hypothesis that our probabilistically motivated methods should perform best with large collections.

The relation query capacities will enable querying for objects, their relations and their relation to annotation items. We will extend our work on more elaborate feedback schemes for similarity queries on graphs. Our use of an relational database management system as basis for Bothrops will ease investigating schemes that implement across-session learning.

Finally, FeatureNet will be a tool for work on query expansion schemes that make
Figure 5.9: Experiment with the TSR500 collection. Comparison of three query processing methods (after one step of feedback): Viper, restricted to color features only, bothrops color features as described in 5.5, with and without containment-corrected weighting as described in 4.5.1

use of statistical, as well as spatial relationships between features. Containment-corrected weighting is its first application.
5.6. CONCLUSION

![Graph](image)

**Figure 5.10:** (TSR2500) Comparison of Viper (with only color block features activated) vs. bothrops’ color-only retrieval. The plot shows the retrieval performance both after one one-image query and one step of feedback where relevant images were fed back as positive examples.

![Graph](image)

**Figure 5.11:** (TSR2500) Comparison of Viper (with both color and texture features activated) vs. bothrops’ color-only retrieval.
Figure 5.12: (TSR2500) Comparison of Viper (with both color and texture features activated, as well as separate normalization) vs. bothrops' color-only retrieval.

Figure 5.13: (TSR2500) Comparison of bothrops with corrected weighting vs. bothrops without corrected weighting.
Chapter 6

A Bayesian image browser tracking the user’s goals

In the last two chapters, we have described ways of processing query by example. In the following, we will describe methods that have been developed for performing target searches. The user starts from scratch, i.e. without an example, and is then guided by the system towards the image he is looking for. This technique is an answer to the so-called page zero problem: Usually the problem of finding a useful seed for QBE search has been ignored. Most of the systems give the user the possibility to get random choices of images in the database. This is suitable for testing QBE methods, but clearly insufficient, if one actually wants to use the database for real queries. What one needs here is a process, which provides a suitable seed.

Little attention has been paid to systems which support the browsing process [12,96], i.e. a process in which the user moves freely through feature space, by expressing his or her preferences. To our knowledge these systems were specialized browsers without nearest-neighbor capabilities. This chapter focuses on the browsing process and puts it in context with query by example, showing that these capabilities are complementary.

The method for measuring performance of browsing systems established in [12] is the target test: the user is shown an image, which he or she then has to find in the database. The number of interaction steps is recorded. A good system minimizes the number of interaction steps necessary for finding an image.

One point of criticism against the target testing method is its failure to capture the often gradual, sometimes sudden, changing of mind users experience during longer query sessions. In this chapter we describe tracker, a system which takes the possibility of query drift into account. Along with it we propose an extension of the target test which permits the measurement of the performance of systems which try adaptively to help the user in the browsing process.

In the following section 6.1 we describe the Bayesian method used by PicHunter as well as our modifications to its model, which rely on a model of user feedback inconsistency: if the user gives feedback which is strongly inconsistent with earlier judgments, this can be caused by a change in what he or she is looking for.

Section 6.2 then establishes a new benchmark for the performance of image databases: the moving target test. It is put in context with other performance measures.
6.1 Bayesian methods for browsing queries

6.1.1 PicHunter: a static target searching system

Information retrieval systems usually try to find (more or less explicitly) documents \( \delta \) which have a high probability of being wanted by the user given the query. Here the query can be text as well as an example. In most systems the user is given the chance to improve the result by giving feedback, thus looking for \( \delta \) which optimizes \( P(\delta \text{ wanted by the user} | \text{query, feedback}) \). This paradigm has been largely adopted by the image database community. In image databases, the situation however is quite different, from the (text) information retrieval case, due to the inability of the user to formulate pictorial queries as precisely as a textual or a SQL query. In many cases the goal will be to find one or more interesting images in the collection, without being able to furnish a suitable seed for a query by example. This leads to the browsing query case, in which an explicit query does not exist, and in which we look for images which optimizes \( P(\delta \text{ wanted by the user} | \text{feedback over several steps}) \). The main difference from the normal IR case is that relevance feedback is not seen as an improvement to an already well-phrased query, but as a query language for a user who is unable to formulate ad-hoc queries.

Cox et al. try to solve this rephrased query problem by maintaining a probability distribution which contains for each document \( \delta \) of the database the probability that \( \delta \) is the target of the query, i.e. the goal the user has in mind when starting the query. To this end the system, PicHunter, gives at each interactive step a small set of images, the suggestion \( S \), to the user. The user responds by giving some feedback \( F \) to the system. The update is done using the classic Bayesian rule:

\[
P(T = \delta | F, S) = \frac{P(F | T = \delta, S)P(T = \delta, S)}{P(F | S)}
\]  

(6.1)

In this formula \( P(F | T = \delta, S) \) is the user model: it describes the expected feedback, if we know that the target is \( \delta \) and the suggestion \( S \) had been given. \( P(T = \delta, S) \) is the prior knowledge of the target’s whereabouts, and \( P(F | S) \) (the probability that \( F \) is given at the same time that \( S \) is suggested) is a normalizing factor. The suggestions \( S \) are chosen to minimize the expected number of comparisons needed to find the target using an entropy argument. The feedback given by the user is given by comparison, i.e. the user decides which of the suggested images are closer to the target than other suggested images. Thus the elementary feedback a user can give is one comparison between two images: we write

\[
\text{user}(T : i, j) : \iff \text{The user considers } i \text{ closer than } j \text{ to the target } T
\]  

(6.2)

User modeling in PicHunter is based on the assumption that one is in the possession of a distance metric \( d(i, j) \) for images \( i \) and \( j \), which captures the human “internal” similarity measure to such an extent that one can assume that it is perfect except for blurring by “mistakes” of the user:

\[
P(F | T) = \frac{1}{1 + e^{\frac{d(T, i) - d(T, j)}{\sigma}}}
\]  

(6.3)

where \( \sigma \) is a free parameter which has to chosen before the process by the implementor. In the case of \( \lim_{\sigma \to 0} \) (6.3) converges to

\[
P(F | T) = \begin{cases} 
1 & d(T, i) < d(T, j) \\
0.5 & d(T, i) = d(T, j) \\
0 & d(T, i) > d(T, j)
\end{cases}
\]  

(6.4)

This limit case of no blurring shows two potential weaknesses of this method:
6.1. BAYESIAN METHODS FOR BROWSING QUERIES

1. \( \sigma \) obviously depends on the image collection as well as the intended users. More importantly, it depends also on the metric employed for the search.

2. Once an image \( \delta \) is discarded from the set of potential targets (i.e. \( P(T = \delta) \ll 1 \)) by “wrong” feedback, it is difficult for \( \delta \) to be reconsidered.

These two points are equally important for the moving target problem which we address in this chapter.

6.1.2 tracker: modeling the changing mind

Motivation

The shortcomings of the PicHunter can be summarized by saying that it would be desirable to take our uncertainty regarding the user model into account, using stronger methods than blurring. This need is emphasized by another observation: In the beginning of a real-world browsing query the user usually has a target in mind. During the query, however, the user often changes his or her target as a function of what he or she thinks can be found in the database.

One method for coping with this is to provide the user with an explicit means to express changes of mind. However, this would place the burden on the user to decide clearly, if he or she changes his or her ideas about the target or not, and how much these changes affect user feedback given in previous steps.

We propose here a framework which consists of weighting the different comparisons according to the degree to which they are trusted. Comparisons which are in less contradiction with others are more trusted.

Consider the simple case of a modified Hi-Lo game (the Hi-Lo game is classic programming exercise for beginners: let the user find a real number in a given interval). Consider finding a number \( t \) in the interval between \([0, 1]\). One person, \( A \), asks \( B \) which one of the points \( x_1, x_2 \) is closer to \( t \). If the \( B \) is good at arithmetics, there is no hesitation: \( A \) has to choose \( x_1 \) and \( x_2 \) such that they are equidistant from the midpoint. By doing this, \( A \) can expect to half the set of points still to be considered. The problem will be reduced to either finding a point in \([0, 0.5)\) or \((0.5, 1]\). However, if you consider players \( A \) and \( C \), where \( C \) sometimes has little problems with addition subtraction and comparing numbers, \( A \) will either employ the same tactics as PicHunter i.e. “blurring” and/or \( A \) will deliberately choose \( x_1 \) and \( x_2 \) so that “surprising” results can be detected.

This observation leads us to a tradeoff: a sequence of comparisons chosen so as to strictly minimize the number of steps to be taken will each time halve the distribution, but it will not allow for the detection of inconsistencies except at the moment where there are no more points to consider. A “no-surprises” sequence of comparisons, however, will start at one end of the interval and consider every point.

A framework for the definition of inconsistent user behavior

For our definition of inconsistency of user feedback, we regard it as convenient to express (6.1) rather as an intersection of probabilistic sets. A probabilistic set over a set of items \( I \), \( \mathcal{I} \), is a set of pairs \((p, i)\) where \( p \in [0, 1] \) denoting the probability of an item that an \( i \in \mathcal{I} \) is in the set.

\[
\mathcal{I} := \{(p, i) | i \in I \land p \in [0, 1]\} \tag{6.5}
\]

We define now the intersection between two probabilistic sets \( \mathcal{I}, \mathcal{J} \) with weighting constant \( w \) (to be explained below), using the function \( f \) as

\[
\cap (\mathcal{I}, \mathcal{J}, w, f) := \{(f(p, p, w), i) | (p, i) \in \mathcal{I} \land (p, i) \in \mathcal{J}\} \tag{6.6}
\]
We write the multiplication of the probability of each element in a probabilistic set $I$

$$\times (\alpha, I) := \{(\alpha p, i) | (p, i) \in I\}$$

We can write the normalization (in the probability distribution sense of the word) of a probabilistic set, as

$$\text{Normalise}(I) := \times \left( \frac{1}{\sum_{(p, i) \in I} p} \right)$$

If the suggestion was $S$, the user feedback $F$ and the knowledge prior to a learning step is described by the probabilistic set $I$, the learning step (6.1) becomes

$$J = \cap \{ (P(T = X|S,F), X)|X \in \text{database}, 1, \text{multiply} \}$$

with $\text{multiply}(a, b, c) = a \cdot b \cdot c$.

We will call $\{(P(T = X|S,F), X)|X \in \text{database}\}$ the feedback set of $F$ (and $T$):

$$\text{Feedback}(F, S, T) := \{(P(T = X|S,F), X)|X \in \text{database}\}$$

Thus we can regard the learning process as a sequence of intersections of probabilistic sets. The probabilities of the members $(p, i)$ of the resulting sets represent a plausibility of $i$ being the target. Accordingly we define: a weighted set of user feedback steps $F_k$ with weighting constants $w_k$, $\{(w_1, F_1), \ldots, (w_N, F_N)\}$ is consistent under a plausibility predicate $\text{Pred}$ (given the combination function $f$) iff

$$\exists (p, i) : \text{Pred}(p, i) \land \left( (p, i) \in \bigcap_{k \in \{1, \ldots, N\}} \text{Feedback}(F_k, S_k, T), w_k, f \right)$$

i.e. there is at least one image in the intersection of the feedback sets which is considered to be plausible by the predicate $\text{Pred}$. For short, we write

$$\text{Consistent}(\{(w_1, F_1), \ldots, (w_N, F_N)\})$$

iff $(w_1, F_1), \ldots, (w_N, F_N)$ is consistent.

The $w_k$ in equation (6.11) correspond to our "belief" into Feedback $(F_k, S_k, T)$ as result of a useful comparison. Our learning problem is thus to be augmented by the search for $w_k$ which make $(w_k, F_k)$ consistent. If one wants to view this in a more Bayesian style, $w_k$ modifies the user model, depending on the belief in the comparisons. This leaves us the choice of a function which finds for the $F_k$ proper $w_k$.

Furthermore, we have to choose a combination function $f$ and the user model.

First of all we want to emphasize the temporal component: we assume that more recent comparisons are more credible than older ones. Without this assumption, our relations become symmetric, and we have no way of deciding which comparison is to be weighted lower if two comparisons are inconsistent.

Knowing that computing time is limited, one possibility is to make each $w_k$ a function of the feedback given in the $m$ previous and $m$ following steps $(F_{k-m}, F_{k-m+1}, \ldots, F_{k+m-1}, F_{k+m})$ for some small $m$. In this chapter we take a more rule based approach given by the following algorithm:

$N$ is the number of feedback steps given so far.

1. $i \rightarrow N$

2. $w_i \rightarrow \left\{ \begin{array}{ll} 1 & \text{Consistent}(\{(w_1, F_1), \ldots, (w_{i-1}, F_{i-1})\}) \\ 0 & \text{otherwise} \end{array} \right.$
3. until \( i = 1 \lor \# \{ w_k \mid k \in \{1, \ldots, N \} \land w_k = 0 \} > \vartheta \) do \( i \to i - 1 \) and goto 2

This leaves us with the choice of a suitable \( \text{Pred} \), the choice of \( \vartheta \), as well as the choice of a combination function \( f \). The \( \text{Pred} \) is a simple threshold comparison. Here, we chose \( f(a, b, c) = \text{multiply}(a, b, c) \).

The user model was chosen as

\[
P(F|T) = \begin{cases} 1 & d(T, i) \leq d(T, j) \\ 0.1 & d(T, i) > d(T, j) \end{cases}
\] (6.12)

Similar to the original papers about PicHunter, we estimated the expected entropy gain by sampling several suggestions \( S_i \) from the current distribution, and taking the \( S_i \) which maximized the decrease of entropy in the current distribution. In order to generate suggestions which enable inconsistent user feedback, we slightly favor \( S \) which minimize \( \sum_{k=1}^{N} w_k \), i.e. which force forgetting of user feedback.

The system

The tests in the present work were performed using Viper the system which uses techniques inspired by text retrieval (inverted files), on a very large quantized feature space, as described in chapter 4. Viper obtains in Query-By-Example good Precision-Recall in interactive time for databases of moderate size (\( O(1000) \) images).

Viper's features and scoring algorithms yield a distance measure which is not symmetric. This is in accordance to psychophysical evidence, that human similarity perception is not symmetric [94]. However, for use in the present framework, asymmetry of the distance measure is undesirable. The distance measure is also required to fulfill the triangle inequality. A symmetric distance matrix fulfilling these constraints to a suitable extent was built for the experiments.

Viper is designed for flexibility. Our present system allows the use of browsing queries and nearest neighbor queries in parallel. This capability was used for some of our tests.

6.2 Evaluation

The normal QBE process corresponds to deep exploration of the feature space in the immediate neighborhood of the example. The relevance feedback usefully leads to a deformation of the feature space in order to better capture the user's view of similarity, and to capture the differences between the user's example and the user's information need. How much the feature space chosen for a program corresponds to the average information need of the unexperienced user can be measured in precision-recall plots and derived measures, as well as using other methods using user relevance data.

Target testing, however, captures the mobility of the user within the feature space. He is supposed to move in an autonomous fashion in feature space, thus finding an image whose characteristics are not known to the database. In this target testing and query by example are testing two complementary properties which should be present in every CBIRS. Browsing query systems help in query formulation, while good QBE systems have a good query performance.

In most real cases the user will be rather interested in approaching the target using a browsing query mechanism. After a certain point however, he or she will profit from a good and quick overview over the images which are inside a certain region of the feature space which can be obtained using a nearest neighbor query.

The weakness of the target testing method, however, is its focus on one target image. This generates mainly two problems:
arguably, the user will react in a different way, if knowing the whole image to
be found, than if he or she knows only a semantic category the wanted image
falls in.

equally arguably, a real user will change his or her mind during the browsing
process. Giving a target image to the test person is like telling the buyer
which jeans to buy before sending him or her into the shopping mall and
guaranteeing that he or she will find it in one of the businesses in the mall.

We consider that giving a target image to the user is the one method to verify
that different results for different systems are not only due to the fact that the test
persons are more or less demanding. However, we suggest to model the inevitable
change of mind by giving the user a sequence of targets he or she has to visit. This
simulates the moves of the target and the ability of the system to follow moves and
to detect abrupt changes of the target. This method of testing we call moving target
test.

In the experiments described in the following subsections we give some mov-
ting target tests, for “simple” images taken from a complex real-world database
containing 2500 images (TSR2500) provided by Télévision Suisse Romande, the
broadcasting corporation of the French speaking part of Switzerland. Before the
presentation of these results we give a short summary of simulations which capture
the “best case”.

6.2.1 Simulation

We simulate a user who uses exactly the same distance metric as the program. This
user tries to find a sequence of four images in the database. The number of tries
for each retrieval is counted.

Of course, these simulations do not prove anything about real users. However,
they show that the requirements for the use of this method are met by the distance
measurement, combination function and algorithm. If the user does not make any
“mistakes”, he or she will be able to find a sequence of images. Because of the
perfection of the simulated user the simulations should provide an upper limit for
the performance of this method.

We did two kinds of simulations:

1. a simulation, in which the simulated user gave at each step negative feedback
   for all images but the best match to the target, giving positive feedback for
   the best match. Here the simulated user had to see 38 images on average for
   reaching each query target.

2. a simulation in which it gave negative feedback to the worst match and positive
   feedback to the best match. Here the simulated user had to see 75 images on
   average for reaching each query target.

Simulations of a quickhunter, a tracker version without forgetting (this is close
to PicHunter) suggest that without forgetting feedback, performance seriously de-
grades after finding the first target. See the appendix B to find a visualization
of two runs of such a simulation. These runs illustrate very well the rationale of
tracker.

6.2.2 User Experiments

For moving target tests we chose four scenarios:
6.2. EVALUATION

Viper with and without feedback memory  for previous feedback steps: As a reference, Viper was used with a random seed to find the targets of the target sequence. 20 images were visible at each feedback step.

tracker: tracker memorized the last 10 feedback steps and cumulated the knowledge gained from older feedback in an additional eleventh feedback set. Here, $\theta$ was ignored. At each step 5 images were shown.

Split screen: In addition to the suggestions (5 images) provided by tracker 15 images from a Viper nearest neighbor query were shown. Using this, the user had the opportunity to explore the feature space given by the feedback images chosen by the user. The fact that 15 images were shown from a nearest neighbor query, and only 5 from a suggestion, is not a contradiction. The choice of a suggestion takes time linear to its size, while a nearest neighbor query scales much more favorably.

As a general modification to the original PicHunter papers, in our experiments, the user had explicitly to give both positive and negative feedback, thus leaving space for indifference in case of uncertainty of the user.

Choice of test images: Preliminary experiments had shown that the TSR2500 database contains many very small clusters of images, which are so semantically different from each other that it is difficult for the user to judge image similarity (Geman et al. [19] write in this context of “virtually random” comparison outcomes). For the test described here we chose four images from large clusters in the database: banknotes, trademarks and flags, sunsets (very dark background), airplanes (mostly sky).

Test persons  two test persons with image processing background performed multiple experiments with tracker and modified versions. As it will be described and discussed below we had learning effects during some of the tests.

Viper without feedback memory  The user was started from a random set of 20 images, giving feedback in order to move in the direction of the next target. Here the user was able to give feedback at each step. This feedback was taken into account for the calculation of a new set of 20 images resembling the given feedback and then forgotten. 420 ± 100 images had to be seen by the user for finding all images in each target sequence. The details can be found in table 6.1.

<table>
<thead>
<tr>
<th>Image</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$ bill</td>
<td>80</td>
<td>60</td>
<td>120</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Honda</td>
<td>40</td>
<td>80</td>
<td>80</td>
<td>140</td>
<td>140</td>
</tr>
<tr>
<td>Sun</td>
<td>60</td>
<td>120</td>
<td>120</td>
<td>260</td>
<td>120</td>
</tr>
<tr>
<td>747</td>
<td>140</td>
<td>60</td>
<td>80</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Sum</td>
<td>320</td>
<td>320</td>
<td>400</td>
<td>600</td>
<td>460</td>
</tr>
</tbody>
</table>

Table 6.1: This table summarizes the outcomes of 5 moving target tests (Test 1 through 5) with Viper without feedback memory. The test consisted in finding a one dollar bill, a Honda logo, an image of the sun, and an image of a flying Boeing 747 in this sequence. Each time the user started at a different random state. In each cell of the table the number of images seen to find the next target is noted (e.g. in the first test, the user needed to see 140 images, to find the 747 after having found the sun image.) The complete number of images seen in each complete moving target test is summed up in the last column.
**Viper with feedback memory** Here the user also started with a random set of 20 images, giving feedback in order to move in direction of the next target. In contrast to the method described in the paragraph above, the user was able to cumulate feedback over several steps.

We did two runs of five tests with the same expert user. We observed that the user learned quickly, how to optimize his target testing performance when using Viper. He was able to reduce the number of images seen before finding the last target by approximately a third.

On first use by the user, $420 \pm 140$ images had to be seen by the user for finding all images in each target sequence. The details can be found in table 6.2.

<table>
<thead>
<tr>
<th>Image</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1\text{ bill}$</td>
<td>120</td>
<td>100</td>
<td>140</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Honda</td>
<td>80</td>
<td>80</td>
<td>60</td>
<td>220</td>
<td>80</td>
</tr>
<tr>
<td>Sun</td>
<td>80</td>
<td>40</td>
<td>440</td>
<td>80</td>
<td>40</td>
</tr>
<tr>
<td>747</td>
<td>20</td>
<td>120</td>
<td>40</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>Sum:</td>
<td>300</td>
<td>340</td>
<td>680</td>
<td>480</td>
<td>320</td>
</tr>
</tbody>
</table>

Table 6.2: *This table summarizes the outcomes of 5 moving target tests (Test 1 through 5) with Viper with feedback memory. The test task was exactly the same as in table 6.1.*

After having gained some experience, only $260 \pm 40$ images had to be seen by the user for finding all images in each target sequence. The details can be found in table 6.3.

<table>
<thead>
<tr>
<th>Image</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1\text{ bill}$</td>
<td>100</td>
<td>120</td>
<td>100</td>
<td>100</td>
<td>60</td>
</tr>
<tr>
<td>Honda</td>
<td>100</td>
<td>40</td>
<td>60</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>Sun</td>
<td>80</td>
<td>80</td>
<td>20</td>
<td>60</td>
<td>40</td>
</tr>
<tr>
<td>747</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Sum:</td>
<td>320</td>
<td>280</td>
<td>220</td>
<td>280</td>
<td>220</td>
</tr>
</tbody>
</table>

Table 6.3: *This table summarizes the outcomes of 5 moving target tests (Test 1 through 5) with Viper with feedback memory. The test task was the same as in table 6.1.*

**tracker** Five experiments with an identical target sequence were performed, starting at different starting points. The user needed $65 \pm 11$ iterations for performing the task; the user therefore scanned on average 82 images in 16.5 iterations before finding a target of the target sequence. This is approximately 15 times better than chance. While these results are quite satisfactory (better than to Viper before learning) and surprisingly close to the simulation results, the user found his situation quite difficult: images chosen by the system are not necessarily close to the images marked positive by the user. Often this is desired. Sometimes, however, the user has the impression that his or her feedback had been “misunderstood”.

$320 \pm 55$ images had to be seen by the user for finding all images in each target sequence. The details can be found in table 6.4. Within the experiment the performance, attained by the test user using tracker did not increase.

Surprisingly, for the chosen queries, the simulation results are only slightly better than those of a skilled human user. This might be explained by the fact that a human user can consciously induce small contradictions, if he or she does not like
6.3 Conclusions

In this chapter we presented an approach to tracking query drifts in target searches. In target searches the user tries to find an item (target) in the database starting from a small random sample.

The approach was implemented in the system tracker. In order to measure the performance of the system we made user experiments with expert users. They used the system to find a number of targets in a fixed sequence. Targets were simple images taken from a dataset of 2500 images. Future tests will involve random targets, as well as non-expert users.

Our user experiments gave two main results:

1. tracker enables the user to move in feature space by giving feedback. It is able to follow changes of the user’s wishes by forgetting parts of the user feedback deemed to be inconsistent with newer user feedback. Moving to a new target from and old, found target seems not to be more costly than explicitly starting a new query.

<table>
<thead>
<tr>
<th>Image</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 bill</td>
<td>110</td>
<td>85</td>
<td>75</td>
<td>130</td>
<td>85</td>
</tr>
<tr>
<td>Honda</td>
<td>105</td>
<td>75</td>
<td>40</td>
<td>135</td>
<td>105</td>
</tr>
<tr>
<td>Sun</td>
<td>100</td>
<td>25</td>
<td>85</td>
<td>50</td>
<td>105</td>
</tr>
<tr>
<td>747</td>
<td>45</td>
<td>45</td>
<td>140</td>
<td>80</td>
<td>50</td>
</tr>
<tr>
<td>Sum:</td>
<td>360</td>
<td>230</td>
<td>340</td>
<td>395</td>
<td>310</td>
</tr>
</tbody>
</table>

Table 6.4: This table summarizes the outcomes of 5 moving target tests (Test 1 through 5) with tracker. The test task was exactly the same as in table 6.1.

the current suggestion. The system then will open up the distribution by forgetting parts of the old feedback.

Split screen tracker/Viper This method was a reaction to the subjective impressions of our test user when performing tracker queries. At each iteration 20 images were shown to the user. 5 of them were a suggestion by tracker, 15 the result of a nearest neighbor query, which used the feedback given in the last step. As one can see, these results are clearly the best of our tests.

Also the subjective impression when using this version was satisfactory: the tracker part provide the user with good seeds which in the present simple setting were quickly usable for successful nearest neighbor queries.

On average, only 225 ± 30 images had to be seen by the user for finding all images in each target sequence. The details can be found in table 6.5.

<table>
<thead>
<tr>
<th>Image</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 bill</td>
<td>80</td>
<td>20</td>
<td>60</td>
<td>140</td>
<td>100</td>
</tr>
<tr>
<td>Honda</td>
<td>40</td>
<td>20</td>
<td>60</td>
<td>60</td>
<td>40</td>
</tr>
<tr>
<td>Sun</td>
<td>60</td>
<td>40</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>747</td>
<td>60</td>
<td>100</td>
<td>60</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>Sum:</td>
<td>240</td>
<td>180</td>
<td>200</td>
<td>260</td>
<td>240</td>
</tr>
</tbody>
</table>

Table 6.5: This table summa rises the outcomes of 5 moving target tests (Test 1 through 5) with tracker. The test task was exactly the same as in table 6.1. These results are the best over all tested methods.
Figure 6.1: The last step, when trying to find the second of our four test images for the split screen (Viper/tracker). The first line shows images suggested by tracker, the next lines show the results of a nearest neighbor query on the feedback given in the last step. In the third row on second position, you can see a part of the target image.
2. A combined system of a target searching system and a nearest neighbor QBE system was preferred by the user and performed best in our test. It enables the user to choose quickly between browsing movement in the feature space and intensifying his or her search in some point of the feature space.

Our results suggest that one should not see target searching systems as a stand-alone method, but rather as a convenient, efficient way of solving the problem of finding a suitable seed for nearest neighbor (QBE) queries.

In the future we want to explore more of the parameters given in section 6.1.2. At present we have the subjective impression that our system forgets too late, and sometimes too completely. We are especially interested in a method which does not have any parameters which have to be chosen before the query process. Furthermore, we would like to adapt our target searching methods so that they scale better with the database size than does the present explicitly maintained probability distribution. The results given in section 6.2.2 for “Viper with feedback memory” encourage this approach.
Chapter 7

Automatic benchmarking of image browsers

Content-Based Image Retrieval Systems (CBIRS) are designed to help their user in finding images, making use of the content of each image in the collection, as opposed to labels attached to the images. The existence of large, yet un-annotated image collections, as well as the inherent limitations of image annotation motivate the research in this area.

Most current CBIRSs provide Query By Example (QBE). Here the user gives one or more positive and negative example images in order to describe the images he or she would like to retrieve using the CBIRS. The system then will present a list of images to the user who usually has the possibility to refine his or her query, by giving additional examples from the response set. Techniques for the evaluation of such systems are close to those used in text retrieval. An overview of such techniques, adapted to the case of image retrieval is given in [49].

While QBE addresses the question how to find images similar to a given, small set of images, interactive browsing addresses the problem of finding a given image in a collection. That is, QBE addresses the problem of closely exploring a given point (or a small region) in the collection, whereas browsing systems address the problem of mobility within the collection.

There are two main directions of research in image browsing: deterministic and stochastic browsing systems. Both of them present the user with successively refined overviews of the collection. The user then can express his or her preferences by marking one (optionally more, depending on the system) images as relevant or irrelevant with respect to the goal of his or her search. This information is then processed, leading to a new, refined overview of the collection.

The differences between deterministic and stochastic systems lie in the way the overviews are provided, and in the way one can navigate through the image collection. Deterministic systems provide a hierarchy which guides the image search performed by the user. The hierarchy usually is pre-calculated. This drawback is at the same time an advantage: each image search will start with the same initial selection. The browsing process could be compared to moving through a city without a map. The user has the possibility to move through the collection using fixed paths. He or she has the possibility to memorize which images will lead to which other images during the search.

In contrast to this, stochastic systems provide overviews in function of user-feedback. In contrast to hierarchical systems one has the possibility to mark multiple images as more or less relevant to the query. As a consequence, at each stage of the retrieval process, the user has so many possibilities for feedback, that a pre-
calculation of the possibilities is infeasible. Thus the task varies with each image search, and it is too complex to attempt a brute-force calculation, leading to the use of Monte–Carlo methods. Monte–Carlo methods imply reproducibility on average, as opposed to exact reproducibility.

Both kinds of browsing systems have in common that a true test of their performance requires interaction with a user: the test user is presented with a target image, which he tries to find using the system. The performance is measured in terms of numbers of images the user had to look at. Images encountered twice are counted twice.

To our knowledge only one deep test of this kind has been done, the test of PicHunter [12,64]. In fact, for research groups of small size and low financial resources, tests like the one of PicHunter are difficult to conduct. Test users are hard to get and it is difficult to evaluate the influence of the test user’s background to the experiment (e.g. computer vision researchers make systems look better than other office workers, but by how much?). Moreover, in deterministic systems, users cannot be used twice for the same test because they are likely to remember useful details of the last test run.

This leads to the thought of automatic benchmarking using low–level features (i.e. color and texture). The user is replaced by a piece of software which tries to find the target image. Cox et al. used this in PicHunter as a proof of concept backed up by real–user experiments. Vendrig et al. [96] used this as the only benchmarking method for their deterministic browsing system. However, in both cases the benchmark uses the same user–model as the system to be tested, thus using the testing hypothesis (“we have a useful feature set coupled with a useful learning method”) for its own verification. Examples which illustrate the shortcomings of this approach are given below in § 7.2.2. It is argued that low–level feature based systems are not apt to function as user simulators for benchmarking other low–level feature based systems.

We advocate a browser benchmark which is based on structured annotation. The annotation is used to simulate the learning problem a browser is facing: closing the semantic gap between visual low–level features and the semantic concepts the user is looking for. The problem of finding a good annotation method for our purpose is non–trivial. We describe the development of a structured annotation method coupled with an appropriate retrieval method for graphs with weighted edges and nodes.

As mentioned earlier, the interaction concepts of hierarchical browsers and stochastic browsers differ. In hierarchical browsers, the user needs to backtrack actively, if he or she reaches one leaf of the hierarchy. A stochastic browser will present suggestion after suggestion until the target image is found. We present a pluggable software architecture using the communication protocol MRML [53] which copes with this problem.

This chapter is organized as follows: in section 7.1 we give a definition of the goals of an image browser. We also give examples which illustrate the problem faced by the designer both of an image browser and of the benchmark. Section 7.2 describes the annotation method, as well as the associated retrieval method we derived for the benchmark. This retrieval method permits QBE (as opposed to hand–formulated querying) on annotation. We describe how this method can be extended to the case of combination of annotation (i.e. semantic features) and low–level features. The usefulness of this method for QBE is evaluated using precision–recall graphs on 8 example queries.

Finally, this benchmark is used on a simple PicHunter clone (section 7.2.5).
7.1 Defining the goal of image browsers

For defining a useful performance measure, one needs to first define what is optimal performance. Otherwise all measurements will be useless. Defining a performance measure which is based on the simulation of user behavior, we also have to define which aspects of user behavior we want to simulate. To put it differently: in which aspects is the system supposed to help the user?

7.1.1 CBIRS using low level features

Content based image retrieval was invented as an enhancement to image annotation, and as an answer to the lack of annotation in common image collections. As the classical computer vision problem ("tell me, what's on this Image?") remains unsolved, CBIRS make do with low level features, sometimes accompanied by optional annotation and sophisticated interaction techniques. Using annotation in images has been shown to improve retrieval performance of low-level feature based systems[34]. This is unsurprising, because annotation contains the semantics we are not fully able to capture in low level features. However, the main issue for measuring the success of CBIRS research is evaluating the contribution of low-level feature based systems to retrieval success. As a consequence, we constrain the formulation of a benchmark for browsing systems on systems which do not use annotation for the search (at least while benchmarking).

7.1.2 Formal description of the browsing problem

In the following we assume that the user browses a given collection of images (of size N) to find one target image T. Derived problems (find one out of n images in a collection of size N) are usually easier, but not in any fundamental way. The user applies some kind of distance measure $d_{\text{semantic}}(I_1, I_2)$ between images $I_1, I_2$ which is mainly semantic-based, and generally a function $C \times C \rightarrow [0,1]$ which does not satisfy the triangle inequality (i.e. it is not necessarily a metric). The browser, however, will apply a different distance measure, based on low level features $d_0(I_1, I_2)$. The discrepancy between these measures is the consequence of the semantic gap. There are now two alternatives: either the browser's measure is close enough to the user's measure to permit browsing without having to traverse large parts of the collection, or the system tries to learn from the user's feedback a measurement $d_{\text{browser}}(I_1, I_2)$ which approaches $d_{\text{semantic}}$ in a sufficient manner.

7.1.3 Requirements for a good browser benchmark

As stated, the main use of an image browser is helping the user to close the semantic gap between low-level visual features and high-level semantics in order to browse through a collection in a way he or she understands. This is we identify as the principal requirement which should be evaluated by an image browser benchmark.

As a consequence, any automatic benchmark which uses low-level features only is useless for true evaluation. The process of learning a mapping between different color spaces is much easier than learning uniquely from user feedback that e.g. the user wants images with at least one dog on it. We feel that noise is insufficient to model the insufficiencies of the current feature-based user models (Fig. 7.1). Furthermore, such an evaluation masks the principal problem of image browsers: in many situations, meaningful answers are not possible without knowledge about the the low-level feature set, as illustrated in figure 7.2.

As a second consequence, if we want to get an evaluation of the discussed properties of the browser, the browser is not allowed to use annotation. Using annotation
Figure 7.1: An example where the semantically more distant image is considered closer to the query. Viper [86] (in high-speed-low-quality mode) considered the middle image closer to the left image, than the right image. This is due to matching the black trousers of the man in the middle picture to the dog jumping in the left picture. However, clearly the semantics of the right image is closer than the left.

Figure 7.2: An example that illustrates that in many cases, a sensible answer does not exist. During the browsing process, often the user is confronted with questions like the following: “What is more similar to the image of the man and the sitting dog: the mountain or the pound note?” Stochastic browsers provide the user a possibility to decline an answer if the selection does not offer the possibility for sensible feedback.

would help bridging the semantic gap in a straightforward way. It would rather work around the insufficiency of low-level features for retrieval, than perform learning to improve their usefulness.

Text annotation and low-level features also are separated by a semantic gap. This semantic gap might not be as large as the semantic gap between the user’s wishes and low-level features. However, it is considerable and similar in nature to the semantic gap a user experiences. As a consequence we suggest to benchmark image browsers by testing their target testing [12] performance when simulating users using a textual distance measure \( d_{\text{ext}} \). The distance \( d_{\text{ext}} (I_1, I_2) \) between two images is determined using text retrieval techniques on the annotation of \( I_1 \) and \( I_2 \). Details are described below.

7.2 Ranked QBE on structured annotation

We now focus on a semantic-based distance measure between images. At first glance, this problem seems to be easily tractable using classic text-retrieval techniques. However, this is not the case, as described in the following subsection.

We then describe the structured annotation approach we adopted, along with
the retrieval method we used on the annotation. We give a performance-evaluation
of this annotation, and compare it to the performance of a similar, unstructured
approach.

7.2.1 Differences with classical text retrieval

Textual information retrieval is an old research area of strong economic interest. Much
research has been done in the last 40 years. The successful establishment of
a common benchmark by the Text REtrieval Conference (TREC [93]) has rendered
results comparable and has created a general competition for the best text retrieval
solution.

Presently, the systems performing best in TREC use very little linguistic or
semantic knowledge. As E. Voorhees [98] states text “is regarded as little more than
a bag of words”. To summarize the basic principle of many systems: for each
term (word) of the query, each document which contains this term receives a score,
depending on the frequency of the term in the document \( tf \), as well as the frequency
of the term in the collection \( idf \), leading to \( tf \cdot idf \) measures. The rationale is: if a
document contains a term rare in the collection, this distinguishes the document
well from others. If a document contains a term frequently, it is supposed to be
important for the document.

Natural language processing (NLP) techniques have had little success up to
now. It has been identified as one reason [98] that the disambiguation techniques
are too error-prone: the precision gained by more accurate modeling of the word
relationships is lost by trusting too much in wrongly established word relationships.

However, in our case the situation is different:

- TREC deals with documents of kilobytes in size whereas annotation usually is
  much shorter. As a consequence, statistical measures like \( tf \cdot idf \) will produce
  less accurate results.

- While chances are high that in well written long texts multiple synonyms
  of one meaning are used, usually only one synonym of a given word will
  appear in a very short text. Thus the analysis of short texts requires better
determination of the true word sense.

- In the scenario of a query that has been formulated by hand, one can assume
each query term to be relevant to the user. However, in our case we are
interested in the distance between documents, \( i.e. \) we are interested in QBE
instead of hand-formulated queries. As a consequence, not every query term
is also relevant to the user who gave the example.

- Annotation is made for retrieval purposes. Adding structured annotation to
an image takes only little more time than adding unstructured annotation
to an image. We can use the structured annotation for replacing the faulty
disambiguation step by hand-made disambiguation.

Consider the following example: A database contains an image, annotated by the
caption A dog jumping over a bar. bushes in the background. people in
the background. Using this as a positive example in a QBE query on captions, a
normal text query might retrieve a statue. bushes in the background. people
in the background. The result A dog jumping over a bar. would get a lower
rank, as less items match with the query.

Intuitively, each item we want to describe has to be described using at least one
word, even if it is of little importance. Because of the shortness of the annotation
text, it is a matter of chance (\( i.e. \) the statistics of the database), if the term we
employed for the background item is rare or frequent in the database, and thus,
<table>
<thead>
<tr>
<th>Syntax example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setting(athletics)</code></td>
<td>the word athletics describes the image as a whole.</td>
</tr>
<tr>
<td><code>actor(dog,$dog)</code></td>
<td>the word dog designates an entity which is capable of action. A graph node is created for this entity. It can be referred to using the word $dog. The node ID $dog is visible for all subsequent facts that refer to the same image. i.e. the node name $dog will be reusable in another image.</td>
</tr>
<tr>
<td><code>actor(dog)</code></td>
<td>Abbreviation for <code>actor(dog,$dog)</code>. Similar abbreviations exist for other tags</td>
</tr>
<tr>
<td><code>actorS(dog)</code></td>
<td>Many dogs ($dog), an abbreviation for a combination of several tags.</td>
</tr>
<tr>
<td><code>actorS(dog,10)</code></td>
<td>10 dogs ($dog).</td>
</tr>
<tr>
<td><code>thing(house,$house)</code></td>
<td>The house designates an entity incapable of action. $house designates the node created for this entity.</td>
</tr>
<tr>
<td><code>modifies(black,$dog)</code></td>
<td><code>Modifies</code>, further describes an entity node which has been created.</td>
</tr>
<tr>
<td><code>enumerates(10,$dog)</code></td>
<td>Specialization of <em>modifies</em> for numbers.</td>
</tr>
<tr>
<td><code>action(run,$run)</code></td>
<td>Creates a node describing the action run.</td>
</tr>
<tr>
<td><code>staticAction(stand,$stand)</code></td>
<td>For discerning verbs like sit, lie, stand.</td>
</tr>
<tr>
<td><code>performs($man,$hit)</code></td>
<td>The entity $man performs the action $hit: a man is hitting (in the following is assumed that $man, $hit etc. have been instantiated in a sensible way).</td>
</tr>
<tr>
<td><code>isPerformedOn($hit,$dog)</code></td>
<td>a dog is hit.</td>
</tr>
<tr>
<td><code>using($hit,$stick)</code></td>
<td>a stick is used for hitting.</td>
</tr>
</tbody>
</table>

| Table 7.1: This table describes the main syntactic elements which were used for structured annotation. We omitted some instructions for spatial relationships on an image. |

which weight it will receive. In other words, there is a need for pre-weighting of terms.

Onnis [61] suggests segmenting the image and choosing the term frequency of each annotation item proportional to the number of pixels that are representing the image. We did not choose this approach for two reasons: firstly, one can imagine quite a number of cases where the amount of pixels covered is not at all related to the importance of the item. Take for example a formula one car on the race track. The car will occupy less space on the image than the track, however most of the time we will be rather interested in the car than in the track. Secondly, we did not have a tool for interactive segmentation necessary for performing such a combined annotation task.

Instead of Onnis' suggestions, we chose to derive a pre-weighting from structured annotation, as described next.

### 7.2.2 The structure of the annotation

As it was expressed in the last subsection, structuring the annotation causes little overhead. Furthermore there is the need for structuring the annotation, in order
### 7.2. RANKED QBE ON STRUCTURED ANNOTATION

<table>
<thead>
<tr>
<th>Syntax example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>isSubjectOfActionWord(§dog, bite).</td>
<td>The node §dog participates in a performs relation with an action node §n that contains the action action(bite, §n)</td>
</tr>
<tr>
<td>isObjectOfActionWord(§dog, bite).</td>
<td>The node §dog participates in a isPerformedOn relation with an action node §n that contains the action action(bite, §n)</td>
</tr>
<tr>
<td>isPerformedByThingWord(§bite, dog).</td>
<td>The node §bite participates in a performs relation with an actor or thing node §n that contains the actor actor(dog, §n)</td>
</tr>
<tr>
<td>isPerformedOnThingWord(§bite, dog).</td>
<td>The node §bite participates in a isPerformedOn relation with an actor or thing node §n that contains the actor actor(dog, §n)</td>
</tr>
<tr>
<td>isRelatedToActionWord(§dog, bite).</td>
<td>The node §dog participates in a isPerformedOn or in a performs relation with an action node §n that contains the action action(bite, §n)</td>
</tr>
<tr>
<td>isRelatedToThingWord(§bite, dog).</td>
<td>The node §bite participates in a isPerformedOn or in a performs relation with an actor or thing node §n that contains the actor node actor(dog, §n)</td>
</tr>
</tbody>
</table>

Table 7.2: Guiding features for relationship matching. As described in section 5.3, it is useful to add flat features to node in order to achieve a better node-to-node matching, and thus a better matching of relations. For this reason the above features were automatically added to the annotation when indexing.

to add information about the importance of the different items of the annotation. Structuring the annotation is meta-annotation. Its main advantage for us lies in enabling the use of a-priori information about the importance of items in the annotation text.

In our annotation effort, we focused on emphasizing the importance of participation in an action as opposed to passiveness. The rationale of this is that most of the time, if there is action on an image, the parts of the image that are not implicated in the action are less important, they are to be considered as part of the background. This is especially the case for our image collection, where unknown individuals are mainly interesting because of their function in the image, which is mainly defined by the action they perform, like e.g. participating in a political demonstration or performing a high jump.

Furthermore, we wanted to be able to express subject-object relations: classic text retrieval methods will not distinguish between 'Man bites dog.' and 'Dog bites man.' However, while the first example surely is a news item¹, the second happens every day.

We designed a small set of relationships, not with the intention of being linguistically complete, but with a view on our image collection and the interesting

---

¹It actually was a news item in USA Today November 6th, 2000
relationships between items in this collection. For structuring our annotations, we
implemented a small language which compiles facts with Prolog-like syntax into
Prolog. The main syntactic elements of the annotation are described in Table 7.1.

This permits the expression of simple semantic networks. The annotation
presented here is not a full-fledged linguistic annotation. However, it captures the
basic relationships between items of the annotation. Please note that the annotator
has to name by himself the nodes of the semantic net for one image, but that the
burden to find non-conflicting node IDs for the whole collection is taken care of
by the compiler that translates this structure into Prolog. The language presented
here has the advantage that while being close enough to programming languages
for testing purposes, the syntax is short and simple to memorize. The left half of
Fig. 7.3 shows the semantic network derived from the annotation

\[
\begin{align*}
\text{actor}(\text{man}). \\
\text{actor}(\text{man}, \text{~man2}). \\
\text{action} (\text{ask}). \\
\text{performs}(\text{~man}, \text{~ask}). \quad \% \text{ read: "man performs ask"} \\
is\text{performed} (\text{~ask}, \text{~man2}). \quad \% \text{ read: "ask is performed on man ~man2"}
\end{align*}
\]

The structure of the annotation maps exactly on the structures, bothrops is able
to process. See 5.3 for a description of bothrops’ algorithms

7.2.3 Performance of the annotation

In this section we will describe the performance of the annotation when doing QBE
on an annotated collection. The goal is to show that this annotation gives a good
“one-shot” retrieval performance, making the annotation a suitable basis for the
simulation of a real user in a browsing scenario. Our results are compared to the
results of applying classical text retrieval methods on unstructured annotation with
the same content. This unstructured annotation was derived from the structured
annotation by removing the structuring.

For this experiment, the described annotation and retrieval scheme was used on
the 500 images provided by the Télévision Suisse Romande, the French-speaking
Swiss television station. This image collection was chosen for its diversity. It con-
tains scenes of varying complexity and varying degree of action.

The images were presented in portions of four images to the annotator on a
1024 × 768 pixel 13.3” LCD panel. The resolution per image was 256 × 256 pixels.
7.2. RANKED QBE ON STRUCTURED ANNOTATION

The annotator (the author) had the opportunity to scroll back and forth both in the annotation and in the image collection. The average time spent per image was about 5 minutes. After the complete annotation was done, a debugging pass was performed. Here, drifts in annotation strategy (detected using test queries) as well as typographical and syntax errors were corrected.

We then performed 8 QBE queries, using the annotation scheme and the retrieval method described in section 5.3. The performance of these QBE queries were evaluated using relevance data which were collected for the experiments with Viper[85]. User data was collected for five users. Each one performed the queries by hand, thus providing for each image a list of images relevant to the query. We kept all results for each user, thus storing the whole range of user behavior. This enabled testing the performance on relevance feedback, as shown in [50]. In our present experiment, these relevance judgments were used to obtain precision–recall graphs of one-shot-queries on structured annotation.

For all queries, the structured annotation performed at least as well as the equivalent unstructured annotation (derived from the structured annotation by suppressing the structure). However, once again, it becomes clear that the problem of QBE for images is ill-posed: what is considered as relevant differs widely between the test subjects. With most query images both structured and unstructured annotation reached perfect performance for at least one test user. With some other images there is an advantage for the structured annotation, as shown in 7.4.

Both annotation methods performed very badly on test images that showed buildings as the only noticeable image item. We see as an explanation that both the annotator and the test users have no architectural background. So the relevance data were rather a product of the visual impression than of the semantics. However, in architecture and art, established classification methods exist [24]. These might be included in future versions of our annotation.

We also experimented with the use of a thesaurus for improving the retrieval performance. We found that in our scenario, synonym sets and synonym disambiguation can be used in a beneficial way (e.g. volume—book instead of volume of liquid, for example). When using WordNet [14], we experienced performance improvements when using WordNet synonym sets instead of the words of the annotation. We would like to underline that also in this case disambiguation has been done by hand in order to improve the query result. Trying to add WordNet hypernyms (i.e. generalizations) to each annotation item in a straightforward way, degraded the query results. However, we suggest adding hand-selected hypernyms where appropriate (e.g. policeman—man).

Most important in this context is that the structured annotation produced long strings of images that were semantically consistent with the query. As a consequence, we found that our annotation can be used as user simulation for enabling a benchmark for image browsers.

In the past sections we have first described our needs for a distance measure for images which is closer to the human way of thinking than low-level features. We then described an annotation method as well as an associated retrieval method that permits QBE on images using the annotations. Now we will proceed with the description, on how to use QBE on annotation for benchmarking browsing systems.

7.2.4 snakemeter A benchmarking harness with pluggable components

We present a benchmark that uses MRML (see chapter 3), an extensible query protocol for content-based image retrieval systems. Here it serves to separate the benchmarking harness and the system under test.
Figure 7.4: Annotation example, describing 5 people in a library standing next to bookshelves, reading and choosing books. The precision-recall graph compares the performance of structured and the corresponding unstructured annotation for this example.

The benchmarking harness serves for generating a sequence of queries to the system under test. It will receive each result and process these results for generating new queries (optional) and evaluating the results. We chose to separate the benchmarking process into querying and graph generation, the idea being that it might be interesting to evaluate the results obtained from a sequence of query results multiple times. Furthermore it became clear during previous experience with a non-MRML benchmarking harness that it was important to provide several ways of evaluating the query results. We wanted to take a step towards a general solution of this problem. *snakemeter* is a prototype we hope to draw from its use conclusions for the design of the Benchathon benchmarking harness.

**Querying**

Even with the standardized communication layer, the following modules change when changing the retrieval method:

**Initial query formulation**: allowing for QBE, hand-formulated queries, or just requesting an initial selection of images.

**Relevance feedback method**: allowing the distinction between QBE and browsing queries. Determining if we are allowed to mark irrelevant images as negative examples, and if we are allowed to mark multiple images?

**Determining relevance**: in a QBE scenario, the only interesting parameter is the membership of a retrieved image in the set of images deemed relevant by test users. In browsing queries, relative relevance is important: at each step, we want to choose images which are more relevant than others.

For the query process, *snakemeter* allows the simple exchange of two modules, called the initial query generator and the feedback query generator. The kernel of *snakemeter* furnishes the query generators services, like e.g. finding the distance between given images by querying a given MRML compliant image retrieval system. This can be used for doing the browsing query feedback described earlier in this section.
7.2. RANKED QBE ON STRUCTURED ANNOTATION

Using the query generators, *snakemeter* automatically performs a user simulation, storing the full results of each query step as MRML string in a relational database. These data can then be evaluated using programs that query the database, the evaluators.

**Graph generation**

As it was said before, *snakemeter* stores all query results in a relational database. This takes lots of memory, but it permits a complete separation of the evaluating and report generating program parts from the querying program parts (short, the *evaluator*) without losing the flexibility gained by use of MRML. If we do not store the complete results, we constrain the evaluation methods that can be used afterwards.

When called, *snakemeter* will query the result database for results. For each result, an *evaluator object* will be constructed. The user now has the choice which evaluator objects to accumulate into *subgraphs* and how to accumulate subgraphs into *graphs*. What happens on merge is determined by the evaluator and by the merging level (graph, subgraph or accumulation).

Take for example a browsing query. For the target test the only interesting parameter is the number of images seen in the query process. For evaluating a series of target test, we will thus first accumulate by summing the result sizes of all queries that were issued by one simulated user for reaching a given target. A subgraph will thus be a line in the report saying 100 images seen for target 1 and user 2. Merging subgraphs will be (again) averaging over subgraphs yielding something like 100 images seen for target 1 averaged over users or 100 images seen for user 2 averaged over targets. The graph will contain lines for each test user (configuration written in Perl):

```
{
    graph=>  "ImageNumber",
    subGraph=>  "UserNumber",
    accumulate=>"QueryCount"
},
```

or each target, as we prefer:

```
{
    graph=>  "ImageNumber",
    subGraph=>  "UserNumber",
    accumulate=>"QueryCount"
}
```

Another example are the precision-recall graphs given in this thesis: they were done with an evaluator that creates a precision-recall graph for each query result. Merging means here again averaging. We used the settings:

```
{
    graph=>  [],
    subGraph=>  "QueryCount",
    accumulate=>["UserNumber","ImageNumber"]
}
```

\(^2\) The reason why we are writing about *graphs* and not about reports, is purely historical. When designing the *snakemeter* framework we mainly had experience with the automatic generation of precision-recall *graphs*. 

\[^{2}\]
To obtain one graph, where each subgraph represented the QueryCount, i.e. the level of feedback given (the initial query has QueryCount=1, the first feedback query has QueryCount=2). The accumulation was done by averaging over all users and all images queried:

```plaintext
{  graph=> ["UserNumber"],
    subGraph=> ["queryCount"],
    accumulate=>["ImageNumber"]
}
```

Would provide us with graphs (one graph for each user), containing two subgraphs (i.e. lines) that depict the average performance for the given user and the given query step, averaged over all test images.

![Sequence diagram for snakemeter’s MRML based benchmarking harness architecture. Here, MRML is used for the communication between snakemeter and the system under test. All components, except for the component marked Benchmarked system are part of the harness.](image)

**Figure 7.5: Sequence diagram for snakemeter’s MRML based benchmarking harness architecture. Here, MRML is used for the communication between snakemeter and the system under test. All components, except for the component marked Benchmarked system are part of the harness.**

**Future work on snakemeter**

As we have stressed, throughout this section, snakemeter is just a prototypical implementation. In addition to some deficits in the cleanliness of the programming code, there is still room for improvement:
Firstly, there is the disk storage problem: If querying for a large number of images, the MRML result returned is simply too large to store everything. Thousands of result elements containing two URLs each, each potentially having a length of 100 characters bring the raw MRML text of a large query result close to one megabyte. We are thus in need of XML compression methods [7], if we want to keep up our flexible approach. A simpler, but less time efficient and less memory efficient approach might be to use generic compression methods before storing the MRML result.

Secondly, there is the problem of the implementation language. If we assume storage of full MRML results in the result database, evaluating the results and generating the graphs (reports) is very slow when using interpreted languages like *e.g.* Perl.

Thirdly, the current implementation relies too much on main memory for doing the evaluation and report generation step. Here we need to think about reducing the memory footprint and doing more work on disk.

However, even with all the above weak points, *snakemeter* has fulfilled its goals: we have shown that it is possible to implement a framework that supports benchmarking of query processors with diverse query and interaction paradigms. More importantly, we have moved towards a benchmarking framework that leverages all the flexibility of MRML. We consider the efficiency problems encountered as solvable and worth the additional effort.

The goal here would be to have a system for the Benchathlon which not only provides the possibility to do benchmarking over the internet (like [48], [26]), but also to query a result database by distance, creating a resource for comparing old and new results, as well as the evolution of systems over time.
7.2.5 Benchmarking an imitation of PicHunter

*snakemeter* was applied on a system that uses PicHunter's Bayesian retrieval method. Our PicHunter-like system used color histogram distance and an image shape spectrum [58] based distance measure. In the following, we will call this system *quickhunter*.

For performing the benchmark we need three entities: The benchmarking system (benchmarking harness, driver, or DRVR, here: *snakemeter*) that dispatches queries to the system under test (SUT) (*quickhunter* in this case) and an annotation-based query engine (AQE, as described in section 5.3), which provides the distance measure.

For the benchmark, the *snakemeter* performed a target test (not a moving target test) for a list of images. For each target image, the benchmarking system performed a query by example on the AQE. The query result was pruned by hand so that it contained images with a relation to the query only once. Normalized, this ranked list served as $d_{ext}$. Now the driver requested a (random) selection of 9 images from *quickhunter*. The image of the selection with the smallest $d_{ext}$ was marked positive, the one with the biggest $d_{ext}$ was marked negative, and this query was submitted to *quickhunter*. *quickhunter* used this to calculate a new selection of 9 images, always based on its $d_{browser}$ distance measure. On finding the query image, the search was re-initialized, and the process was repeated for the next target image.

For each target image, we counted the number of images that were shown to the user before the target was found by *quickhunter*. As query images, we used the same images as for the evaluation of the annotation. The results are shown in Table 7.3. On average, the quickhunter needed to scan 170 images before the target was found, being more efficient than random search (250 images) by about 30%.

We also remark that, in these experiments, the performances of *quickhunter* varied by a factor of ten, depending on the “difficulty” of the target. As a consequence, we do not think that at the current state of research, the target testing performance of a system can be summarized by a single number.

Obviously, the combination of our image collection and our benchmarking method is very hard for systems that do not try to adapt their user model during retrieval. A further difficulty is that, in realistic collections, the number of images that are in any relationship with the target image is very small. This limits the possibilities of the benchmarking harness to give useful feedback, like it does with real users.

It is not surprising that at the current state of CBIRS research this benchmark is very hard for the benchmarked system. Therefore we propose a migration path towards semantic benchmarking: use a superposition of a semantics-derived and a visual (i.e., low-level feature) distance measure for user simulation. The weight with which both distance measures are superimposed could then be used to describe the degree of difficulty of the benchmark.

7.3 Conclusion

We proposed an automatic semantic-based benchmark for image browsing systems. The advantage of such a benchmark is that it is memoryless, and that it is not influenced by imponderables like the previous experience of test users.

We based our automatic benchmark on structured annotation that is augmented using a thesaurus. For this benchmark, we developed a fast query method for semantic networks that performs ranked similarity queries using inverted files, followed by a stage where more elaborate matching is performed.

We see the use of this work as two-fold: First, in the near future, we will study the interweaving between annotation and still image features, especially still image
### 7.3 CONCLUSION

<table>
<thead>
<tr>
<th>Query</th>
<th>#images</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>One dollar</td>
<td>23</td>
<td>perfect user agreement to the annotation.</td>
</tr>
<tr>
<td>500 German marks</td>
<td>50</td>
<td>many almost-relevant images</td>
</tr>
<tr>
<td>Corner view of building</td>
<td>140</td>
<td>medium user agreement, many almost-relevant images</td>
</tr>
<tr>
<td>Library</td>
<td>150</td>
<td>high user agreement visually inhomogeneous relevant set</td>
</tr>
<tr>
<td>Parliament</td>
<td>165</td>
<td>dto.</td>
</tr>
<tr>
<td>Lemons</td>
<td>241</td>
<td>perfect user agreement to the annotation, small set of related images (3)</td>
</tr>
<tr>
<td>Harbour</td>
<td>255</td>
<td>perf. agreement to annotation, but very small set of vis. similar rel. images</td>
</tr>
<tr>
<td>Russian palace</td>
<td>315</td>
<td>very bad user agreement with this annotation</td>
</tr>
</tbody>
</table>

Table 7.3: Benchmarking results for a PicHunter like system, quickhunter, for 8 queries on TSR500: #images designates the number of images that had to be seen by the simulated user before finding the target.

Trivially, the retrieval method proposed allows for inclusion of derived visual features with and without annotation. The algorithms used for this study, are now the algorithms used in the bothrops query engine that can provide such integration. Secondly, the benchmark devised in this thesis constitutes a tool, both for the development and for the evaluation of image browsing systems. We hope it will trigger more research for intelligent systems that learn the semantics of a query during the querying process.
Chapter 8

Conclusion

8.1 Summary

Within this thesis, we have covered a wide range of issues related to content-based image retrieval (CBIR).

Chapters 2 and 3 treat software engineering issues related to CBIR: How to build a CBIR system framework which is flexible enough for accommodating different ideas and different styles, and which provides useful services to the users of the framework. The GNU Image-Finding Tool's (GIFT's) architecture and the Multimedia Retrieval Markup Language (MRML) are solutions to this problem.

We call a query engine an independent query-processing unit within a retrieval system. The query engine is defined by its query processing unit (the query processor which can memorize a state) and its data accessing unit (the data accessor, which is stateless). Throughout the thesis we have described query engines that use the GIFT framework and MRML, as described in the following paragraphs.

Chapter 4 describes how to carry over inverted file indexing, a successful text retrieval technique, to indexing of points in discrete-valued and continuous-valued spaces. In this chapter we also investigate correlations between features and how to take them into account when doing query processing. The Viper query engine is the example of a simple but successful inverted file indexing based query engine. It performs image queries by example.

Chapter 5 extends the inverted file approach to similarity retrieval for graphs using greedy matching, and presents featurenet, a method for classifying and relating features in a self-documenting fashion. An implementation of most of these concepts is presented: the bothrops query engine. As Viper, bothrops also queries images by example. Viper and bothrops cover very similar grounds. As bothrops is more flexible than Viper it is likely to replace Viper within the GIFT framework.

In chapter 6 we investigate the problem of image browsing. An image browser tries to help the user in finding one image (the target) within the collection. The user starts "from scratch" i.e. he is not required to give any prior information to the browsing system. We describe a Bayesian image browser inspired by the PicHunter image browser that (in contrast to PicHunter) is designed to recognize when the user changes his target during the query process.

Due to the flexible design of the GIFT each of the above query engines can be used either separately or combined with one or several other query engines. Combined use means here that every query will be dispatched to several query engines at the same time and that results will be collected and combined to one common result.

Lastly, in chapter 7, we treat the problem of automatically evaluating image
8.2 Putting the results into context

This thesis is very broad, yet the work in every chapter of this thesis was needed for getting on with the work presented in another chapter. In this section we try to emphasize how the pieces fit together.

8.2.1 GIFT and MRML: software engineering

When this thesis was started, the decision to design a protocol for client/server interchange of data in our multimedia retrieval context was already taken. The goal was to enable learning and to exchange user interfaces. However, the details were unclear. July 1998, in a meeting between Zoran Pecenovic of the EPFL, Jilali Raki (a Diploma student), David McG. Squire and the author of this thesis, the latter suggested adopting a human-readable protocol instead of a binary-only protocol. Jilali Raki then implemented MREP (Multimedia Retrieval Exchange Protocol), a grammar expressed in flex and bison, two tools for compiler generation. As it quickly turned out, J. Raki's code was easily changeable, yet the grammar was not "forgiving": every change in the syntax sent by client or server required a change in the flex/bison grammar.

After some work on tools for simplifying the generation of forgiving grammars and automatic creation of more extensive helper code for client and server, the author decided to use XML as base for a more forgiving grammar. This decision led to MRML. Currently, MRML appears to be rare, if not unique, in that it does not try to be typed. Instead, it relies on documentation and graceful degradation for interoperability. This has the advantage of being simple to implement yet to be powerful in use.

The strong interest in building the flexible framework that is the GIFT can be explained by the fact that the author was pursuing his interest in browsing at the same time as he was maintaining the MREP-enabled parts of his Viper code, while others were taking care of making this code ready for a CGI-based demo for the WWW. It was clear that work on browsing would be less useful for the group if it could not be integrated with the other work done within the group.

It has been now a more than one year and a half that we use varying versions of MRML in day-to-day use. During that time GIFT evolved further towards the flexible yet powerful package it is now. Once the basic framework is understood, adding new work is very simple. The author has demonstrated this by programming tracker, bothrops and viPerl in addition to the Viper query engine and integrating them with the GIFT framework as plug-ins.

This thesis shows that we succeeded in building a flexible image retrieval framework that can be used as a tool for rapid development and demonstration of research prototypes in content-based image retrieval.

8.2.2 Diverse results for a common goal

tracker

The browsing experience gained with tracker was very useful and interesting, and it influenced the whole thesis work. In tracker we managed to implement a prototype
that added to a PicHunter-alike browsing system the capacity to detect inconsistencies, i.e. moments when the user changes his mind, or alternatively when the user model of the browser is inconsistent with the user’s feedback in the given context. Furthermore, in our experiments browsing and query-by-example proved to be complementary techniques.

**Browser benchmarking**

During the experiments we, the author and the test user, observed, how much we were able to improve performance by learning (us, the users were learning) when using the Viper system for browsing, i.e. when using a system with reproducible outcomes. This made clear in how much systems like the one described in [96] are incomparable to tracker, because we would have to have a huge “stock” of test persons with very similar background to compare such systems with ours. This lead us to the benchmarking scheme described in chapter 7.

Here, we make the case for a browser benchmark that uses structured annotation in order to simulate a user interacting with the system. We obtain two results:

Firstly, the benchmark obtained models well the subjective user impressions we had when using the browsing system tested. Secondly, our results suggest that structured annotation can be used as a tool to separate in an implicit way between more or less interesting parts of an image, giving a simple possibility to change the weights, and thus the priorities of the simulated user.

**8.2.3 Beyond Viper**

Viper, a query engine that uses inverted file indexing was designed by the author in the very beginning of this thesis according to requirements set by David Squire. It was designed to read the features extracted by a program by David Squire, to create an inverted file, and then to query this inverted file. Later, it was extended to be an experimental platform for diverse weighting methods. Initially being a tool to be started from the command line, Viper was the starting point for the GIFT framework. Diverse capabilities were added by other members in the group. With these additions, Viper performs well both in speed and in retrieval performance.

The main motivation to improve on Viper’s query methods and data representation, was given by our wish to build a system that learns from interaction. We considered the results obtained using tracker very encouraging, however, it was clear that the next useful step was to be a system that modifies the user model during browsing, while (if possible) maintaining the ease of use. At the same time, we needed to reduce time $O(N)$ and space complexity $O(N^2)$ for obtaining a system that would be useful also for larger collections. Trying to integrate the PicHunter approach (model $P(T|F)$) based on maximum-entropy inspired approaches) to Viper, failed, as in each feedback step there was too little useful information gained about the features that were to be in the target $T$. One reason for this is that there is no information kept in Viper that describes which and how features are related to each other. A step towards a solution is featurenet, described in chapter 5.

In short, in this thesis we present good browsing results, however, improving on these results needs preparation. Much of this preparation for a new, more powerful browser has been achieved in this thesis, and all of the results obtained in this process are useful, even when they are used without browsing. Currently they are of benefit for query processing within the GIFT framework, permitting us to go beyond Viper.

When designing the benchmark described in chapter 7, annotation experiments with simple keywords on a small collection of images had shown us that we would
need structured annotation in order to provide a-priori weights that are adjusted to
the function of a word in the annotation. The development of bothrops was started
to provide a query engine for structured annotation. During the development of
bothrops we focused on data structures and retrieval methods that would also be
useful for fast ranked retrieval of heterogeneous multimedia documents.

bothrops unites a speed comparable to that of Viper, with the possibility to query
for documents that are comprised of multiple nodes. In principle, these nodes can be
anything from an annotation item (concept), to an image segment, or even a video
sequence. Currently we use this framework for indexing structured annotation, as
well as for content-based indexing of images by color layout. In contrast to Viper,
bothrops uses relations between features (provided by featurenet) to improve the
weighting of these features. The theoretical basis for the weight corrections is an
adaptation of van Rijssbergen’s results for text retrieval to the image retrieval case,
where we can change the design of the feature set to simplify the correction of
weights.

We report experiments on two test collections (500 images and 2500 images,
obtaining better performance on the 2500 image collection) using bothrops’ color
layout features, obtaining results that strongly encourage our approach. Here, we
are looking forward to confirmation by the Benchatlon effort, an international
initiative for a common content-based image retrieval benchmark.

featurenet is a tool to describe features in a human-readable, as well as machine-
readable fashion. Describing features enables us to administer automatically multiple
groups of features without the need for a name-assigning or number-assigning
organization. It also gives the possibility to relate features to each other, thus
enabling simple calculation of containment relations between features. These con-
tainment relations are the base for probabilistically corrected weighting of features,
as used in bothrops. Using many different feature groups is a precondition for in-
dexing heterogeneous multimedia documents such as HTML pages, or MPEG-7
descriptions.

In our opinion, the bothrops/featurenet framework unites efficiency with exten-
sibility. In this framework, we are extending the methods first used in the Viper
query engine, putting them on a stronger theoretical base, as well as extending
Viper’s methods to the difficult case of graph similarity retrieval.

8.3 Achievements

8.3.1 Query by example

In this thesis we have implemented Viper, a content based image retrieval query
engine based on inverted file techniques. This query engine permits query by exam-
ple with relevance feedback. Viper is the most direct adaptation of text-retrieval
techniques to image retrieval to date. In this thesis, we have shed some more light
on the probabilistic implications of the text retrieval techniques that are used within
Viper, adapting and applying results from probabilistic text retrieval. We have em-
phasized that this is especially important in image retrieval, as correlations between
features are expected to be stronger than correlations between words.

Based on the inverted file paradigm, we have designed a novel retrieval method
that unites efficiency with the capability to process similarity queries on graph struc-
tures. It is designed for graphs that are simplified conceptual graphs with added
capability for large numbers of unstructured features, a case that is very interesting
for structured multimedia documents. Our algorithm trades an acceptable loss in
precision for obtaining time efficiency of the same order (i.e. \( n \times \log(n) \)) as in in-
verted file retrieval for unstructured documents. As a corollary, we can perform also
efficient Viper-like queries (i.e. for unstructured documents). The new algorithm was implemented in the bothrops query engine.

Algorithms as the ones used in Viper and bothrops provide much flexibility: the algorithm is able to process a large number of features, simplifying the combination of diverse models leading to diverse feature representations. However, we can only leverage this flexibility if we provide a tool for proper administration of features. Only with such a feature administration tool are we able to combine multiple features and multiple models with automatic maintenance of consistency. featurenet, designed and implemented within this thesis provides such a feature administration utility. It has been used as an accessor by bothrops.

In short, we made considerable advances towards an efficient, theoretically well-founded image retrieval system that permits not only inserting/deleting/retrieving of images (all during runtime), but also adding new classes of features to the system during runtime. This is especially important in the context of new upcoming extensible data formats (read: formats that imply features unknown before runtime) like MPEG-7.

8.3.2 Browsing

In this thesis, we also presented work on interactive browsing of image collections. Here we give the user the possibility to advance towards a target image by successively refining collection overviews. tracker, presented in this thesis is innovative in that it questions old feedback, in order to detect the changes of mind of the user, providing the user with mobility in feature space.

These results are encouraging, and pinpoint the fact that we need to move further towards systems that adapt the user model during runtime.

8.3.3 The GIFT framework: combine query engines

As it has been stated in the literature, combining diverse models as well as diverse search methods can increase the performance of the resulting system. The GIFT permits combination of diverse query engines at runtime. Thus, the GIFT puts the query engines described in the previous sections into a common framework: all query engines described in this thesis can be used stand-alone, and together with other query engines.

While the framework is in itself a contribution (it is freely available, and external contributions are welcome), it is also a study on how to design a system that is flexible enough for being a useful tool for research that is open for future enhancements, and is lightweight enough to be maintained by one person devoting only part of his or her time to software issues. This achievement became possible through consequent use of XML-based tools and techniques.

8.3.4 Benchmarking of browsers

Furthermore, we presented an original elaborated approach towards automated image browser benchmarking, in which we use ranked retrieval results obtained using structured annotation to simulate a user looking for an image only by semantic concepts. Our results that were obtained using a test collection support the need for browsers that adapt their user model.

We describe a benchmarking harness prototype, snakemeter. In addition to its immediate usefulness for our research, snakemeter constitutes a step towards benchmarking harnesses that provide both a framework for flexible automatic query formulation, but also a framework for flexible automatic report generation. The goal is here to change behaviour using plugins for both query formulation and report
8.3.5 MRML: putting the pieces together

All the software work has benefited from MRML, the protocol language for multimedia retrieval developed in this thesis. Ironically, MRML draws its usefulness from its limitation on the most important issues. It does not try to provide an universal query language, but rather to provide as quickly as possible a simple but viable solution to the communication problem in image databases, focusing on extension mechanisms to ensure long term usability of the technique. Its focus on simplicity in use and implementation, is in contrast to related efforts, such as MPEG-7.

8.4 Future work

As we have described in the previous sections, we presented in this thesis a general framework for content-based image retrieval based on inverted-file indexing, implemented in the GIFT which uses bothrops and featurenet.

From this new starting point, many directions of work are possible. All the projects described in the following section depend heavily on the real-life performance of our framework. Possibly we will need to include something similar to contexts (in the conceptual graph's sense) into the bothrops query processor. We will also need to investigate, in how much a purely greedy approach to similarity retrieval of graphs is appropriate for large heterogeneous documents. This cannot be tested without a large real-world user base, and a base of heterogeneous documents along with a feature extraction method that translates these documents into graphs. We hope to work on these issues in the next few years. Furthermore we want to explore the areas described in the following sections.

8.4.1 Software issues

bothrops and featurenet are implemented to a level that permits performing all experiments described in this thesis, however, we need more experience to know which operations would need to be supported by the feature server to make feature server use the full range of featurenet's flexibility. Moreover, we would like to work (or to support work) on a generic feature extraction framework to integrate feature extraction more closely with the GIFT.

We also need to add to the GIFT persistent session management in order to have a clean and extensible way of learning from user interaction across multiple sessions.

8.4.2 Image retrieval issues

In this thesis we have concentrated on issues related to indexing as well as learning from user interaction. We mainly assumed here the feature set i.e. the vectors to be quantized and indexed to be given. In particular, it was perceived as a plus that dimensionality reduction was not necessary for acceptable speed, due to the virtues of inverted file indexing.

We think that now our techniques have sufficient maturity to study the combination of dimensionality reduction techniques with inverted-file based retrieval.

At the same time we would be interested in studying the use of more sophisticated feature extraction techniques with bothrops than the ones presented here.
8.4.3 Towards the intelligent desktop

In our opinion, time has come for making content-based retrieval an integrated tool on the computer user desktop. For making this vision a reality, we need to provide content-based retrieval for any type of file (in particular heterogeneous documents like web pages referencing multimedia data) along with user interfaces that permit using such content-based services in any work situation. bothrops and featurenet are our contribution to the retrieval side of such a system. Working towards the intelligent desktop doubtlessly will add to and improve both bothrops and featurenet, and it will drive our research for intelligent browsers.
Appendix A

Further documentation of MRML

A.1 State machine of MRML client–server communication

Client–server-communication in MRML is a sequence of connections. In each connection a single request or a small group of requests is answered by the server using a single message or a small group of messages. The state machine in fig. A.1 describes the communication starting with the point where the first makes contact with the server.

The client establishes first contact with the server by sending a get-server-properties message. As a response the client receives a server-properties message which is empty for standard MRML. However, this message is an important stub for extensions which concern the connection itself (e.g. finding out, if the server is able to do a session using a permanent connection.)

After receiving the configuration description, the client will ask for a list of sessions for a user, using the get-sessions tag. The reply is a session-list. The client will now open one session using open-session, getting an acknowledge-session-op as return. The opened session is required to have a sensible default state, i.e. a state which allows queries.

Opening the session, the server has received the user’s name, password and the session-id. This means, it has all the necessary information for knowing which collections and algorithms the user should see. Please note that no one is forced to do user-dependent configuration of the system, but MRML gives the possibility of doing so. So, after opening the session, the client has the possibility to request both lists of collections and algorithms.

Both collections and algorithms are described by the query-paradigms they allow, as well as some other parameters. In particular, an algorithm can contain as an attribute the ID of a collection on which it will be used.

Getting both a list of collections and a list of algorithms, the client has enough information to configure the session which has been opened: when configuring the session, the client sends a configure-session signal which contains an algorithm with the attributes algorithm-id and algorithm-type set. The attribute collection-id a suitable algorithm.

After this, the session is fully configured and can be queried (using query-step). Queries can be grouped into transactions for group queries for logging and learning purposes.
Figure A.1: The state grammar of MRML client-server communication.
A.2 query-paradigms

Algorithms are described by their algorithm-id and their algorithm-type, as well as by their query-paradigm-list. A query-paradigm-list contains query-paradigm elements which contain an unspecified number of attributes. One of which can be the attribute query-mode which at present has the possible values "qbe" or "browsing". All other attributes presently are extensions.

The main use of the query paradigm list is to enable clients to determine which collection can be used with which algorithm. In short, an algorithm can used with a collection, if their query-paradigm-lists match.

Two query-paradigm-lists $l_1$ and $l_2$ match, if there is at least one pair of query-paradigms $e_1 \in l_1, e_2 \in l_2$ such that $e_1$ and $e_2$ match. Two query-paradigms $e_{1,2}$ match, if for the sets of their attribute-value pairs $S_{1,2}$ holds:

$$((a, v_1) \in S_1 \land (a, v_2) \in S_2) \implies (v_1 = v_2)$$

In particular, a query-paradigm tag without attributes matches any other query-paradigm tag.

A.3 Algorithms

As it was said, Algorithms are described by their algorithm-id and their algorithm-type, as well as by their query-paradigm-list, and (optionally) an allows-children element (which in turn contains another query-paradigm-list).

As described in the last section, the first query-paradigm-list specifies which collection can be queried with this algorithm, and it informs the client about its properties. The client or its user can then decide if to proceed or not.

It is possible to specify algorithms recursively. Algorithms can contain other algorithms, possibly several of one type algorithm-type, the algorithm-id however, has to be unique in one configure-session statement. It is thus possible to let the client specify meta-queries. Which kind of meta queries can be built, decided by the allows-children tag. An algorithm $a_1$ is allowed to contain another algorithm $a_2$, if the query-paradigm-list contained in the allows-children tag of $a_1$ matches the query-paradigm-list of $a_2$.

A.4 MRML property sheets

MRML property sheets are a method to work around the fact that the a common set of configuration parameters for image databases is difficult to find and probably awkward to use. We suggest to achieve this by sending code which allows to build GUIs (i.e. the subset you would need for configuration of an algorithm), along with a specification of how to generate pieces of XML code from the GUI's state. This code is XML and it will not be executed, so, to our knowledge, there is no inherent security hole.
A.4.1 A simple example

Viper is a system which uses inverted files for the indexation of images. Each image is translated in a variable-length sequence of features which describe the image. Each feature is assigned a weight determined dependent on the frequency of the feature within the image and within the collection. How exactly this is done, depends on the weighting functions. Both retrieval performance and processing speed of the system depend on the weighting function.

Viper gives the possibility to choose the weighting function at runtime, using an attribute cui-weighting-function of the algorithm element. The following property sheet gives the possibility to choose between two weighting functions.

The “basic need” of a system would be to specify the collection, i.e. the database on which the retrieval is to be performed. For testing and comparison it would be interesting to have the choice between several algorithms (e.g. wavelet coefficient/color histogram based).

A choice out of a list of two elements:

```xml
<property id="p1"
    type="subset"
    caption="Weighting function"
    visibility="visible"
    sendtype="attribute"
    sendname="cui-weighting-function"
    minsubsetsize="1"
    maxsubsetsize="1">
    <property id="p2"
        type="setelement"
        caption="Best fully weighted"
        visibility="visible"
        sendtype="value"
        sendvalue="best-fully";
        defaultstate="selected"
    >
    </property>
    <property id="p3"
        type="setelement"
        caption="Classical IDF"
        visibility="visible"
        sendtype="value"
        sendvalue="classical-idf";
        defaultstate="unselected"
    >
    </property>
</property>
```
What does this do exactly?

- It defines a list from which the user is allowed to choose a subset of size between 1 and 1, i.e. an exclusive choice.
- When asked for its state this list will generate an attribute, i.e. the text given by send-name, plus =: cui-weighting-function=.
- The value of the attribute will be determined as follows: follows property Elements p1 and p2 are identical in structure. They denote the elements of our set which can either be selected or unselected. If selected they send a text which will be placed like an attribute value (value). This text will be "best-fully" or "classical-idf", depending on which of the two list items is chosen by the user.

As a result: the piece of MRML above will enable the interface to set up a property sheet which comprises a list of two items, of which one can be selected. Depending on the selection, the interface will send either

cui-weighting-function="best-fully"

or

cui-weighting-function="classical-idf"

to the server. The MRML client will use this property sheet when generating a configure-session message.

More complex: generate XML subtrees

The following example describes the generation of whole document subtrees. This feature is not yet immediately useful for Viper or CIRCUS. However it provides an explanation on how the text generated in the previous is included into configure-session message. More important is the fact that it provides a general framework for describing (GUI) entities which can send XML.

Consider the following example: Imagine an algorithm which runs the query image through a series of filters before running them through a simple query processor. Being a research system, we would like these filters to be run-time-configurable. Each filter needing some parameters, the and number of filters being variable, we simply need to define some new MRML tags which permit us to describe the sequence of filters. We would an output like the one given below:

```xml
<filter-list>
  <filter filtro-type="horizontal-gabor"
    filtro-gabor-sdev="50"
    filtro-gabor-wavelength="10"/>
  <filter filtro-type="gauss" filtro-gauss-sdev="5"/>
</filter-list>
```

The corresponding property sheet would look like:

```xml
<property id="p1"
  type="panel"
  caption="Filter Sequence"
  visibility="invisible"
  send-type="element"
```
send-name="cui-filter-sequence"

<property id="p1"
type="multi-set"
caption="Filter"
visibility="visible"

send-type="element"
send-name="cui-filter"

minsubsetsize="0"
maxsubsetsize="5">

<property id="p11"
type="set-element"
caption="Gaussian blur"
visibility="pop-up"

sendtype="attribute"
sendname="cui-filter-type"
sendvalue="gauss"

defaultstate="selected"

<property id="p111"
type="numeric"
caption="Standard deviation"
visibility="pop-up"

sendtype="attribute"
sendname="cui-filter-gauss-sdev"

/defaultstate="selected"

</property>

<property id="p12"
type="set-element"
caption="Horizontal Gabor"
visibility="pop-up"

sendtype="attribute"
sendname="cui-filter-type"
sendvalue="horizontal-gabor"

defaultstate="selected"

<property id="p121"
type="numeric"
caption="Tile size"
visibility="pop-up"

sendtype="attribute"
sendname="cui-filter-gabor-sdev"
The example above shows exactly the described scenario: The user has the choice to use sequences of 0 up to 5 filters. The filters can be either Gaussian blur or horizontal gabor filters (yes, this is a toy example).

The Gaussian blur can be configured by giving a number between 1 and 100, which will be sent as an attribute (cui-filter-gauss-sdev). The gabor filter can be configured using the two parameters cui-filter-gabor-sdev and cui-filter-gabor-wavelength.

Both the configuration panels will pop-up when the corresponding filter has been selected in the sequence.

In the following section we describe how the text is actually generated, and how dialog dynamics is specified.

A more formal description of MRML property sheets

As it has become clear from the examples, GUIs created using MRML property sheets are in fact a tree of property sheets. Both the XML generated by the property sheet and the dialog dynamics are defined using simple rules.

Dialog dynamics A property element is visible on the screen, if

1. all its ancestors are visible
2. AND
   • its parent is non-selectable OR selected
   • OR its parent has the visibility="visible" attribute set.

Selectability of property elements will be defined below.

A property element is active, if

1. all its ancestors are active
2. AND its parent is non-selectable OR selected

An active property element is defined as an element that can be used for its purpose, i.e. it will be enabled on the GUI screen.
Generating XML  XML is generated during a depth-first-traversal of the property
tree as follows:

- The XML string generated by a sequence of active elements is equal to the
  concatenation of the XML strings generated by each element. The sequence
  of concatenation is equal to the physical sequence of property elements in
  the MRML text.

- The XML string generated by an inactive element is empty.

- The XML string generated by an active element is given by the send-type of
  the property element

  send-type="element": If there is any beginning of an opening tag in the
  XML generated by the ancestors of this property element, it will be
  ended by adding an > to the text generated so far.

  Afterwards this property element will generate the beginning of the
  opening tag of an XML element with a name that is specified by the
  attribute sendname, followed by a space and the content of the attribute
  send-value. As an example: if for a given element the sendname attribute
  has the value xxx, and the content of the send-value attribute is
  'myattribute="5"', the generated output will be

  <xxx myattribute="5">

  After that the children are evaluated in sequence and a closing tag of the
  element will be generated. Before that, the opening tag will be ended, if
  necessary.

  </xxx>, in our example)

  send-type="attribute": If there is no beginning of an opening tag in the
  XML generated by the ancestors of this property sheet no text will be
  generated in this property sheet.

  If there is any beginning of an opening tag in the XML generated by the
  ancestors of this property sheet there are the following possibilities:

  value is nonempty Generate the text given by the values of the attributes
  sendname and send-value in the definition of this property.

  For example sendname="myattribute" send-value="33" will lead to the text
  myattribute="33" as output.

  value is empty begin an attribute definition with a name given by the
  value of the attribute sendname. For example

  sendname="myattribute" send-value="" will lead to the text
  myattribute= as output. The actual definition of the value can
  be provided in two ways:

  - If the current property has an inherent value (i.e. is numeric,
    boolean or textual), this value is taken, and thus the attribute
    definition will be ended.

  - The value definition will be provided by a child.

  send-type="value": If there is no attribute definition which has been begun
  by any ancestor or sibling of this property element, no text is generated.

  Otherwise either the inherent value or the value given by the attribute
  value send-value will be used, as described above.

  send-type="none": This property element will not generate any code.
A.5 The DTD of MRML

Here is the documented DTD of MRML. It has been derived from the DTD used by Viper via a Perl script. We removed all attributes and tags which are Viper-specific extensions to MRML and added some highlighting of the comments:

<!--

**Basic structure:** Messages are sent as MRML texts. In order to make it easy for the server to know who connects, each message is assigned the id of its session as an attribute.

*Author of this file: Wolfgang Mueller with lots of suggestions and corrections from David Squire, Arjen P. de Vries and Christoph Giess*

--> 

```xml
<!ELEMENT mrml (begin-transaction?, (get-configuration|configuration-description|get-sessions)*)>

<!ATTLIST mrml
    session-id ID #IMPLIED
    transaction-id ID #IMPLIED
>```

<!--

**Request: get-configuration**

This is the message an MRML client sends to the server on connection. The message `get-configuration` gives information about the basic server configuration.

--> 

```xml
<!ELEMENT get-configuration EMPTY>
```

<!--

**Response: configuration-description**

The `get-configuration` message is answered by a message which is supposed to hold a description about anything which is nonstandard MRML.

--> 

```xml
<!ELEMENT configuration-description (error?)>
```

<!--

**Request: get-sessions**

The `get-sessions` message permits the user to request existing sessions for a given user. It is sent by the client directly after after the `configuration-description` has been delivered, and prior to any other activity.

Authentication happens before any other activity to give the server the possibility to customise any other information sent to the user. For example, it might be sensible to send different property sheets to different classes of users, or to render some parts of the database only visible to the own work group.
-->  
<!ELEMENT get-sessions EMPTY>
>
<!ATTLIST get-sessions
   user-name CDATA #REQUIRED
   password CDATA 'guest'
>
<!--
Response: session-list

A session is described by its session-id. We also send over a more human-readable name
-->  
<!ELEMENT session-list (session*|error)>
>
<!ELEMENT session (error?)>
>
<!ATTLIST session
   session-id CDATA #REQUIRED
   session-name CDATA 'Default session'
>
<!--
Request: get-collections

gets a list of collections on the server.
-->  
<!ELEMENT get-collections EMPTY>
>
<!--
Request: collection-list

a list of collections on the server.
a collection is described by a list of the of the query paradigms which can be used
for querying it, as well as an ID and its human-readable name.
This means, you do not have to index all collections using all the methods you
want to propose to the server.
-->  
<!ELEMENT collection-list (collection*|error)>
>
<!ELEMENT collection (query-paradigm-list?|error)>
>
<!ATTLIST collection
collection-id CDATA #REQUIRED
collection-name CDATA #REQUIRED>

<!--

Tag: query-paradigm
arises both in algorithms and collections: this describes the kind of query which can
be performed with this algorithm/collection
-->

<!ELEMENT query-paradigm-list (query-paradigm*|error)>

<!ELEMENT query-paradigm (error?)>

<!ATTLIST query-paradigm
query-paradigm-id CDATA #REQUIRED>

<!--

Request: get-algorithms

gets a list of algorithms usable for one collection
-->

<!ELEMENT get-algorithms EMPTY>

<!ATTLIST get-algorithms
collection-id CDATA #IMPLIED
query-paradigm-id CDATA #IMPLIED>

<!--

Response: algorithm-list

returns a list of algorithms for a given collection on the server
-->

<!ELEMENT algorithm-list (algorithm*|error)>

<!--

Tag: algorithm

An algorithm can contain other algorithms, optionally a property sheet, optionally
a query paradigm list optionally an "allows-children" specification
-->

The DTD of MRML

<!DOCTYPE MRML SYSTEM "mrml.dtd" [(<!ENTITY s "">) ]>

<!ENTITY s "">
Tag: allows-children
This tag specifies for an algorithm what kind of algorithms can be children of this algorithm. No specification ⇒ allows no children.

-->

Request: get-property-sheet
get the property sheet for an algorithm

-->
--

**Request: configure-session**
configures the session by giving an algorithm

-->

<!ELEMENT configure-session (algorithm)>

--

**Tag: property-sheet**

**Basic idea:** send a property sheet together with a specification what should be the XML output coming back. Useful for configuring your database.

-->

<!ELEMENT property-sheet ((property-sheet)*|error)>

<!ATTLIST property-sheet property-sheet-id ID #REQUIRED
property-sheet-type (multi-set|subset|set-element|boolean|numeric|caption) CDATA #IMPLIED
visibility (popup|visible|invisible) 'visible'
send-type (element|attribute|attribute-name|attribute-value|child) CDATA #IMPLIED
send-name CDATA #IMPLIED
send-value CDATA #IMPLIED
min-subset-size CDATA #IMPLIED
max-subset-size CDATA #IMPLIED
from CDATA #IMPLIED
step CDATA #IMPLIED
to CDATA #IMPLIED
auto-id (yes|no) #IMPLIED
auto-id-name CDATA 'id'
defaultstate CDATA #IMPLIED>

--

**Tag: algorithm**

An algorithm will be either an empty element with attributes (add some at your will, it will talk with your server anyway, and this is the server which sent the property sheet), or a tree of algorithms.

What is the use of this? Think of configuring meta queries. Together with properties you get a powerful tool.

-->

<!ELEMENT algorithm ((algorithm*)|error)>
>
<!--

**Beginning and renaming sessions**

We want to give the user the possibility to save the current state into "sessions". This might be useful in the case that a user has several classes of goals which s/he knows in advance.

The user can request a new session. S/he can also request a name change for a session.

Ending sessions is implicit: we cannot afford being dependent on the user ending his/her session in a "nice" way, so we do not tempt programmers to do so

-->

<!--

**Interface side**

-->

<!--

send the desired feedback method together with a name for the session

-->

<!ELEMENT open-session EMPTY>

<!ATTLIST open-session user-name CDATA #REQUIRED
password CDATA #IMPLIED
session-id CDATA #IMPLIED
session-name CDATA #IMPLIED>

<!ELEMENT rename-session EMPTY>

<!ATTLIST rename-session session-id CDATA #IMPLIED
session-name CDATA #IMPLIED>

<!ELEMENT delete-session EMPTY>

<!ATTLIST delete-session session-id CDATA #REQUIRED>

<!ELEMENT close-session EMPTY>

<!ATTLIST close-session session-id CDATA #REQUIRED>
Simple user commands (for logging purposes)
(like e.g. back or forward in the command history) (at present the only commands)

-->

Request: query-step

At present we provide only query by example, and search for random images
(done, if one sends an empty query-step tag)

-->

Tag: user-relevance-element-list

List of URLs with user given relevances Our way of specifying a QBE for images.
relevances vary from 0 to 1
<!ATTLIST user-relevance-element
    user-relevance CDATA #REQUIRED
    image-location CDATA #REQUIRED
>
<!--
Response: query-result

each result image can be accompanied by the user given relevance, as well as the
similarity calculated by the program, based on the feature space.
calculated similarities vary from 0 to 1
-->
<!ELEMENT query-result ((query-resultelement-list*,query-result*)|error)
>
<!ELEMENT query-result-element-list ((query-result-element|error)*)
>
<!ELEMENT query-result-element (error?)
>
<!ATTLIST query-result-element
    calculated-similarity CDATA #REQUIRED
    thumbnail-location CDATA #REQUIRED
    image-location CDATA #REQUIRED
>
<!--
Error messages.
-->
<!ELEMENT error EMPTY
>
<!ATTLIST error
    message CDATA #REQUIRED
>
<!ATTLIST cui-time-stamp
    calendar-time CDATA #REQUIRED
>
<!--
Tag: cui-text-query
query using a number of keywords in a string
-->
<!ATTLIST cui-text-query
    query-string CDATA #IMPLIED
>
Appendix B

Two simulated moving target tests

In this section we give the visualization of two moving target test runs. Within the visualization, each state is represented by a greyscale image. Within the greyscale image, each greyscale dot represents one item $\delta$ of the collection, and its probability to be the target: $P(T = \delta | F, S)$. A brighter dot means a higher probability of the dot being the target. Both tracker and quickhunter discard once-seen items from the search. tracker reconsidered items that were part of forgotten feedback steps.

In our synthetic image collection used for the simulation, each image $I$ contained just one red dot at the point $(x_1, y_1)$ with otherwise white background. The distance between two images $I_1$, $I_2$ was given by the Euclidean distance between their respective red dots: $d(I_1, I_2) := \sqrt{(x_{11} - x_{21})^2 + (y_{11} - y_{21})^2}$.

For performing relevance feedback, snakemeter queried the matrix of distances between all images of the collection, using the target image as query. The retrieved list then was ranked, and distance values between 0 (lowest rank) and 1 (highest rank) were assigned to the images.

Then, snakemeter started a query process, using the normalized rank list as base for feedback: at each query step, the highest ranked image of the suggestion received positive feedback, and the lowest ranked image of the suggestion received negative feedback.

To test the robustness against wrong feedback, we subjected quickhunter and tracker to test runs, in which we applied noise to the feedback. Here, before calculating the feedback, we added a random number between 0 and 0.3 to the normalized rank of each suggestion image. We then took these modified ranks as the basis for the calculation of the automatic feedback given to the tracker and quickhunter respectively.

Even when applying incorrect feedback by using this noisy re-ranking method, tracker’s superior moving target test performance stays apparent: 173 (quickhunter) vs. 86 (tracker) feedback steps.

In order to avoid excessive length of the following two sections, we only show the states for simulation runs with correct feedback in the sense of chapter 6.

B.1 quickhunter

145 feedback steps were needed to find all the targets in the sequence $((0,0), (1,0), (0,1), (0,1))$. Please note how, once the distribution is narrowed down to one target, the distribution stays narrow. In fact, the distribution becomes in these images a
small slightly fuzzy grey dot. However, the distribution changes its center. Visually speaking, one gets the impression of a narrow distribution “running after” the target.

Please note how the trace of images seen marks the letter Z within the distribution.


B.2 tracker

Only 47 steps were needed by tracker for the same sequence of targets. Please note, how tracker opens up the distribution, once feedback has been recognized as inconsistent with the current probability distribution. At the same time, tracker forgets which images it has already seen.
Appendix C

Sample images and sample queries

C.1 Example images

C.1.1 20 randomly drawn images from the TSR500 test collection
APPENDIX C. SAMPLE IMAGES AND SAMPLE QUERIES

![Sample Images](image1.jpg)

263  287  296  304

![Sample Images](image2.jpg)

320  368  410  426
C.1.2 20 randomly drawn images from the TSR2500 test collection

133 137 202 395
480 518 610 657
670 775 877 954
1103 1246 1325 1422
1956 2201 2225 2466
C.2 Test queries

C.2.1 The 10 TSR500 test queries
C.2. TEST QUERIES

C.2.2 The 14 TSR2500 test queries

348  377  443  504

933  988  1143  1211

1361  1515  1720  2068

2199  2396
C.2.3 The moving target test images for chapter 6

349 1691 2073 2396
Bibliography


   e.g. URL: http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2046.html, November 1996.


[22] The GNU Image Manipulation Tool. 
   URL: http://www.gimp.org/.

   URL: http://www.kde.org/.


BIBLIOGRAPHY


[56] MySQL database. URL: http://www.mysql.com


[72] Postgresql database.


BIBLIOGRAPHY

[85] David McG. Squire, Henning Müller, and Wolfgang Müller. Improving response time by search pruning in a content-based image retrieval system, using inverted file techniques. In CBAVL99 [7], pages 45–49.


Doctors make many phone calls, talk to many people.
Doctors look out of their front windows, doctors frown, doctors show the strain.
Doctors are just people, born to sorrow, fighting the long grim fight like the rest of us.

[8]