SEARCH ALGORITHMS

on structured and unstructured data

in a large database

Mathys Cornelius du Plessis

SEARCH ALGORITHMS

on structured and unstructured data

in a large database

Mathys Cornelius du Plessis

Submitted in partial fulfilment of the requirements for the degree of
Magister Scientiae in the Faculty of Science at the University of Port Elizabeth.

December 2004

Supervisor : Prof. G. de V. de Kock

Acknowledgements

I wish to thank my supervisor, Professor G. de V. de Kock for his patience, guidance, enthusiasm and willingness to assist me at any time in spite of his busy schedule.

To Kevin Naudé and Tim Gibbon, I would like to express my gratitude for their valuable assistance in proofreading this dissertation.

A special word of thanks to my colleagues, family and friends for their constant encouragement and support during the duration of this project.

Abstract

This project is concerned with the development of a search algorithm for a large archival database.

The Port Elizabeth Genealogical Information System (PEGIS) contains a database consisting of almost 600000 individuals. The standard search algorithms are no longer sufficient to locate individuals in the database.

A new algorithm was required that allows searches on any of the words or dates in the database, as well as a means to specify where in the desired record a word should occur. A ranking function of retrieved records was also required.

A literature study on the field of Information Retrieval and on algorithms designed specifically for the PEGIS was done. These algorithms were adapted and hybridized to yield a search algorithm that allows for the boolean formulation of queries and the specification of the structure of search words in the desired records. The algorithm ranks retrieved records in assumed relevance to the user.

The new algorithms were evaluated with regards to retrieval speed and accuracy and were found to be very effective.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

1.1 Problem Domain

This project is concerned with the analysis, hybridization, implementation and evaluation of search algorithms using structured and unstructured data in a large database. The Port Elizabeth Genealogical Information System (PEGIS) is used as a case study.

1.2 The Port Elizabeth Genealogical Information System

The aim of genealogical research is to create a family tree or ancestor list of individuals. The purpose of a Genealogical Information System (GIS) is to allow for the interactive retrieval, editing, adding and deleting of individuals and their relationships. PEGIS was developed over the last 20 years in the Department of Computer Science and Information Systems at the University of Port Elizabeth. Through the efforts of the Port Elizabeth Genealogical Research Group, the Genealogical Database (GDB) now contains the details of about 600000 individuals. The GDB currently grows by several thousand individuals each month.

Searching for individuals in the GDB is important not only when information is drawn from the database, but also when information is added. It is essential that information of an individual is not duplicated, and researchers should thus be provided with an effective means of locating existing individuals. Due to the rapid expansion of the GDB (in 1991 it

contained only 90 000 individuals), searching for individuals has become an increasingly challenging task. Before the algorithms implemented by the author, searches could only be performed using queries of the following form: *Surname, First names, Birthdate*. The result of such a query is the individual with closest alphabetical and chronological match with the query. The user can then scroll through the alphabetical successors of the first hit. Note that, although the birthdate is an optional refinement to the query, it is highly recommended to specify first names, since only specifying a surname will require the user to scroll through many people with the required surname before the correct record is found. Queries where the surname is unknown are impossible. Furthermore, permuting the order of the first names or discarding one of the first name would result in undesired search results.

Apart from the problems mentioned above, simple searches on name and birth date are no longer effective because of the large number of individuals with the same names (the GDB currently contains more than 390 individuals named SCHALK WILLEM VAN DER MERWE). The problem is compounded by variations in spelling of names (for example Coertz, Coorts, Koorts, Koort, Coertse, Koortsen, Coertsen etc. were all derived from the German surname Kürz) and inaccuracies in the details of an individual. Work has been done on solving spelling variations of names by dividing the names up into equivalence groups [Ple91]. This will be discussed in more detail in a later chapter.

Genealogical researchers often posses information other than a person's surname, first names and birthdate, for example, the names of an individual's parents, spouses or children, or perhaps the person's birth place or occupation. Searches on this information should also be supported.

1.3 Data Structures

The data stored in the GDB can be divided into three main categories: **Person, Event** and **Relationship**. Not all information is relevant to searching (for example, references to

books, archives etc. where information on an individual was found) and such information will thus be excluded in the following discussion. The above mentioned categories contain the following information [Ple01]:

Person:

Surname, given names, father, mother, gender, nickname, general information.

Event:

Event type, place, start date, end date, details. Provision is made for the following events: *Birth, Baptism, Death, Burial, Residence, Will and Estate, Adoption, Military, Medical, Education, Occupation* and *Other.*

Relationship:

Husband, Wife, Marriage type, place, date, divorce date, details.

The actual data structures used to store information can be seen in appendix A. The fields defined above contain general textual information (except *given names* and *surname*), for example, the field *Birth place* could contain the string **Martjie Venter Hospital, Tarkastad** or **On the farm Doornkloof near Cradock**. Most of the fields are left empty for many of the individuals.

The data in the GDB was parsed into the above fields from a previous version of the GIS that, for example, did not provide separate fields for different events (except birth, baptism and death). Although the parsing was reasonably successful, much of the event information in the GDB was simply placed in the *general information* field. As it requires a time consuming manual process to consider each record individually and make the necessary changes by hand, the data is currently still not perfectly distributed in the correct fields.

1.4 Goals of Research

The goal of this research is to create a search function for the GIS. This function should allow the user to search the GDB by means of words and dates that occur in the records, and should rank the resultant records in order of how well each record satisfies the query.

The ranking of the records is very important, since much more than just surname and first names are taken into account. Queries often results in several thousand hits, and should thus be presented in order of most relevant.

The user is likely to know in which field the word or date is located. This is very useful information and it is essential that the search algorithm takes the structure of data in the record and the query into account. For example, if the user knows that a certain word or term is a person's surname, a record that contains the term as a first name should be ranked lower than a record that contains the term as a surname.

Functionality to search on nearby relations of a person (parents, spouses and children) is essential, since this is often the only information known about an individual.

Finally, the algorithms that provide the above functionality should be designed in such a way as to make searches extremely fast and efficient.

1.5 Scoping

Information retrieval is a wide and intensively researched field. Several books have been written that give a complete overview of information retrieval and its applications, notably Ribeiro-Neto et al. [RBY99] and Witten et al [TCB99]. It is not the intention of this project to rewrite these sources. The different information retrieval approaches that are not suitable to being applied to the GIS will only be discussed briefly. Several techniques used by the information retrieval community to minimize response times and disk space used, for example index and record compression, will not be investigated.

1.6 Structure of Dissertation

In chapter 2, the field of information retrieval will be discussed broadly. Retrieval models suited to implementation in the GIS will be focused on.

Chapter 3 describes search algorithms created especially for the GIS. Although most of these algorithms were not implemented, they provide many novel approaches and provide a benchmark for the functionality that should be provided by the new search algorithm for the GIS.

Chapter 4 will be devoted to the description of an information retrieval model for the GIS. This model will be a hybrid of the algorithms and models discussed in chapters 2 and 3.

In view of implementing the algorithms in chapter 4, chapter 5 will focus on the various indexing techniques that could be used in an efficient implementation.

Chapter 6 will consider the various lexicon refinement techniques, whereafter an in depth study of the search terms will be done. The chapter will conclude with a discussion of the lexicon to be used in the GIS search algorithm.

In chapter 7 the implementation of the search algorithm will be described. The discussion will focus on how the retrieval model developed in chapter 4 can be efficiently implemented using the indexing techniques of chapter 5. Various methods used to make searches faster will be discussed.

Chapter 8 will describe the evaluation of the new search algorithms.

The conclusions drawn from this research, as well as future work will be discussed in chapter 9.

Chapter 2

RETRIEVAL SYSTEMS

2.1 Introduction

A data retrieval system can be described simply as a system that returns a record specified by the user. A data retrieval system is simple to create assuming that every record is uniquely identified by some key attribute (for example, in a database storing the study records of all the students at a university, this key attribute would be the student number of each of the respective students) and that the value of that key attribute is known for every record in advance.

An information retrieval (IR) system can be described as a system that returns several records *related* to a query of the user. An example of an information retrieval system is the search engines used to search the World Wide Web, for example, *Google* [Goo04] and *Yahoo* [Yah04]. The main difference between a data retrieval system and an information retrieval system is that with the former the user knows exactly which record should be retrieved, while in the latter the user is interested in records that are similar to a certain query (i.e. not an exact match).

Searching the GDB is always concerned with the location of a specific person (or ascertaining if the person is absent from the database), and thus falls in the data retrieval category. There are, however, several problems which make locating an individual difficult:

1. Apart from a record number automatically assigned to each individual in the GDB, there are no attributes (not even compound attributes) that are guaranteed to uniquely identify an individual.

2. Not all relevant information is stored for all individuals, for example, assume that a specific individual's first name would uniquely identify that person in the GDB. There is no guarantee that the first name was known to the person who originally added the individual to the database.

3. There is a large amount of incorrect data in the database, due to the many discrepancies on birth certificates, marriage certificates, death notices and other genealogical sources.

4. Often only information on an individual's parents, children or spouses are known.

For these reasons, retrieval from the GDB becomes an information retrieval problem, in the sense that it can not be said with full certainty whether a specific record should be retrieved in response to a query.

In the following sections the theories and implementation techniques developed for information retrieval systems will be investigated with specific emphasis on relevance to the GIS.

2.2 Information Retrieval Models

This section describes models used to determine which records to retrieve from a record set, given a query. Implementation techniques for these models will not be discussed. The discussion will focus on models that have been shown to yield good retrieval results while minimizing retrieval time.

2.2.1 Definition

Each record can be described by a set of terms, called *index* terms (Indexes will be discussed in detail in chapter 5). An information retrieval system aims to take as input a query (in the form of a set of terms) and to return a set of records relevant to the query.

A retrieval model M can be defined formally [RBY99] as a quadruple

$$[\boldsymbol{D}, \boldsymbol{Q}, F, sim(d_j, q)]$$

where:

- $\boldsymbol{D}$ is the set of all records in a collection.
- $\boldsymbol{Q}$ is the set of all possible queries.
- F is a framework for modelling record representations, queries and their relationships.
- $sim(d_j, q)$ is a similarity function that computes the similarity of $d_j \in \boldsymbol{D}$ to $q \in \boldsymbol{Q}$.

In the following sections of this chapter, several frameworks will be discussed.

2.3 Boolean Model

This model has been employed by most of the older information retrieval systems. The Boolean model is based on Set Theory. Assume a universal set of all index terms (call this set $\boldsymbol{I}$), then the record set is defined as:

$$\boldsymbol{D} \subseteq 2^{\boldsymbol{I}}$$

(i.e. $d_j \in \boldsymbol{D}$ implies $d_j \subseteq \boldsymbol{I}$). A record can thus be defined as a subset of the universal set. It is clear that some records may not be defined by a unique set.

A query consists of index terms joined by the logical expressions *and*, *or* and *not*. A query can always be written in **disjunctive normal form** (i.e. the disjunction of conjunctions of literals), for example, the query:

$$[(a \vee b) \wedge (c \vee \neg d)]$$

can be written as:

$$[(a \wedge c) \vee (a \wedge \neg d) \vee (b \wedge c) \vee (b \wedge \neg d)]$$

where $a,\ b,\ c,\ d \in \boldsymbol{I}$ and $(a \wedge c), (a \wedge \neg d), (b \wedge c), (b \wedge \neg d)$ are called the conjunctive components of the query.

The retrieved records will be those which contain the conjunctive components of the query as subsets.

Formally: We can define a query q as:

$$q = \bigvee_{r=1}^{n} q_r$$

where

$$q_r = \bigwedge_{s=1}^{m_r} q_{rs}$$

with $q_{rs} \in \boldsymbol{I}$. Define q' as the set of conjunctive components of q:

$$q' = \{q'_1, q'_2, ..., q'_n\}$$

i.e.

$$q'_r = \{q'_{r1}, q'_{r2}, ..., q'_{rm_r}\}$$

The similarity of a record d_j to the query q is defined as:

$$sim(d_j, q) = \begin{cases} 1 \text{ if } \exists\ q'_i \mid (q'_i \in q') \wedge (q'_i \subseteq d_j) \\ 0 \text{ otherwise} \end{cases}$$

From the above it can be seen that a specific record is categorized as either relevant or not relevant. Salton et al. [SFW83] identified the following disadvantages of the Boolean model:

1. The number of retrieved records obtained in response to a query is difficult to control; depending on the occurrence frequency of the query terms and the actual term combinations used in the query, many records might be retrieved, or alternatively, none at all.

2. The retrieved records are not ranked in order of presumed importance to the user.

3. No provisions are made for assigning importance factors or weights to the terms attached either to the records or to the queries; thus, all terms included in the records and queries are assumed to have equal importance.

4. Boolean query formulations may produce counterintuitive results: for example, in response to the query:

$$[a \vee b \vee c]$$

a record containing only one of the query terms is deemed as important as a record containing all of the query terms. Similarly, in response to the query:

$$[a \wedge b \wedge c]$$

a record containing two of the terms is deemed just as irrelevant as a record containing none of the search terms.

In an attempt to address these problems, the Vector Space Model was developed.

2.4 Vector Space Model

The Vector Space Model [SL68] is based on the idea of assigning weights to index terms. These weights are ultimately used to rank retrieved records in order of assumed relevance to a query. As the name implies, every record can be considered to be a vector. These vectors are constructed as follows [RBY99]:

Let t be the number of index terms in the record set and k_i be an index term. Let $\boldsymbol{I} = \{k_1, ..., k_t\}$ be the set of all index terms. A weight $w_{i,j} > 0$ is associated with with every index term k_i of a record d_j. If term k_i does not appear in d_j then $w_{i,j} = 0$. Associated with a record d_j is an index term vector $\overrightarrow{d_j}$ represented by $\overrightarrow{d_j} = (w_{1,j}, w_{2,j}, ..., w_{t,j})$. Let g_i be a function that returns the weight of an index term k_i in any record vector, i.e. $g_i(\overrightarrow{d_j}) = w_{i,j}$.

A vector for a query can be defined as follows:

For a query q, let $w_{i,q}$ be the weight associated with the pair $[k_i, q]$, where $w_{i,q} \geq 0$. The query vector is given by $\overrightarrow{q} = (w_{1,q}, w_{2,q}, ..., w_{t,q})$.

The terms in a query are thus also weighted. The similarity between a query and a record can be defined as the cosine of the angle between the query vector and the record vector and can thus be calculated using the inner product rule:

$$\begin{aligned} sim(d_j, q) &= \frac{\overrightarrow{d_j} \cdot \overrightarrow{q}}{\| \overrightarrow{d_j} \| \| \overrightarrow{q} \|} \\ &= \frac{\sum_{i=1}^{t} w_{i,j}.w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \cdot \sqrt{\sum_{i=1}^{t} w_{i,q}^2}} \end{aligned} \tag{2.1}$$

Thus, $sim(d_j, q)$ will be a number between 0 and 1. The records are sorted according to their similarity to the query and the first n, where n is an appropriate retrieval threshold, will be returned as the result of the search.

2.4.1 Term weighting

Several approaches to assign weights to the record terms have been suggested. The earliest and most well known of these schemes is the *tf.idf* approach, proposed by McGill et al. [MJM83]. Let $F_{i,j}$ be the frequency that index term k_i occurs in record d_j, then define the weight, $w_{i,j}$, of term k_i in record d_j, as:

$$w_{i,j} = tf_{i,j}.idf_i$$

where

$$tf_{i,j} = \frac{F_{i,j}}{\max_l F_{l,j}}$$

i.e. the frequency of a term k_i in record d_j divided by the frequency of the most common term in record d_j, and

$$idf_i = \log \frac{N}{f_i}$$

where N is the total number of records and f_i is the number of records that contain term k_i.

The weights for the query vector are normally calculated as:

$$w_{i,q} = idf_i.(0.5 + 0.5 \times tf_{i,q})$$

where $tf_{i,q}$ is the frequency of term k_i in query, q. Note that, for the query vector weights, the term frequencies are computed over the query and not over $\boldsymbol{D}$.

Several variations of the *tf.idf* approach have been suggested. The most notable attempt at defining a heuristic by which the weighting scheme for a specific scenario could be decided, was the result of experimental work of Salton et al. [SB88]. The equation giving a term's weight can be broken up into three components, each with several possible values:

1. **Term Frequency Component**

b	1.0	Binary weight equal to 1 for terms present in vector (term frequency is ignored)
t	tf_i	Raw term frequency (Note that this differs from the previous definition)
n	$0.5 + 0.5\frac{tf_i}{\max_l tf_l}$	Augmented normalized frequency (tf factor normalized by maximum tf in the vector, and further normalized to lie between 0.5 and 1.0)

2. **Collection Frequency Component**

x	1.0	No change in weight
f	$\log\frac{N}{f_i}$	Inverse collection frequency factor (N is the total number of records and f is the number of records that contain term)
p	$\log\frac{N - f_i}{f_i}$	Probabilistic inverse collection frequency factor

3. **Normalization Component**

x	1.0	No normalization
c	$\frac{1}{\sqrt{\sum_{l=1}^{t} w_l^2}}$	Cosine normalization where each term weight is divided by a factor representing the Euclidian vector length.

Combining the above mentioned three components yields a term weighting strategy for record and query term weights. For example, a similar weighting scheme for record vectors

to the classical *tf.idf* approach would be found by selecting the raw term frequency, **t**, for the term frequency component, the inverse collection frequency factor, **f**, for the collection frequency component and cosine normalization, **c**, for the normalization component. The weighting scheme of the classical approach is found by only using the augmented normalized frequency, **n**. This selection is written as **tfc.nxx** by combining first the letters representing the various selected components for the record vector and then for the query vector. The resultant weighting is

$$w_{i,j} = \frac{tf_{i,j}.\log\frac{N}{f_i}}{\sqrt{\sum_{l=1}^{t}\left(tf_{l,j}.\log\frac{N}{f_l}\right)^2}}$$

for record term weights, and

$$w_{i,q} = 0.5 + 0.5\frac{tf_{i,q}}{\max_l tf_{l,q}}$$

for query term weights.

The following recommendations for selecting a term weighting scheme were formulated by Salton et al. [SB88], after studying the results obtained using different combinations of the term weighting components on several databases:

Query vectors

1. Term-frequency component
   - **b** Use when all terms occur only once
   - **t** Use for long query vectors
   - **n** Use for short query vectors
2. Collection-frequency component
   - Very similar results are obtained for both **p** and **f**, with slightly better results for **f**
3. Normalization component
   - Always use **x** since query normalization does not effect ranking

Document vectors

1. Term-frequency component
    - **b** Use for short record vectors, especially with a controlled vocabulary
    - **t** Use for varied vocabulary
    - **n** Use for technical vocabulary
2. Collection-frequency component
    - For semi-static collections use either **f** or **p**. Slightly better results are obtained for **f**
    - For dynamic collections, where many changes are often made to the database, **f** and **p** requires constant updating. Use **x**.
3. Normalization component
    - **c** Use when the deviation in vector lengths is large
    - **x** Use for short record vectors of homogeneous length

Attempts have been made to create a more effective weighting function using genetic algorithms [Ore02]. Despite the fact that only small improvements were made over the standard *tf.idf* approaches, genetic algorithms still provide a novel way to customize the weighting scheme for a specific IR system.

2.4.2 Disadvantages

A major disadvantage of the Vector Space Model is that a Boolean formulation of queries can not be used (i.e. the **and**, **or** and **not** can not be used in queries). Several hybrid Boolean/Vector Space systems were developed that first used the Boolean Model to retrieve the initial set of records and then used the Vector Space Model to rank these records. An alternative to these systems was suggested in the form of the Extended Boolean Model (discussed in section 2.6 on page 16).

2.5 Fuzzy Set Model

Before the Extended Boolean Model is discussed, a very brief description of the Fuzzy Set Model will be given.

2.5.1 Fuzzy Set Theory

Unlike Boolean logic, Fuzzy Set Theory considers the membership of elements to a set to be gradual, i.e. an element is only associated with a particular set to a certain extent. Formally [Dur94]:

Definition 2.5.1 *Let $\boldsymbol{X}$ be a universe of discourse, with $\boldsymbol{x} \in \boldsymbol{X}$. A fuzzy subset $\boldsymbol{A}$ of $\boldsymbol{X}$ is characterized by a membership function $\mu_A : \boldsymbol{X} \rightarrow [0,1]$ which associates with each element $\boldsymbol{x}$ of $\boldsymbol{X}$ a number $\mu_A(\boldsymbol{x})$ in the interval $[0,1]$ (i.e. $\mu_A(\boldsymbol{x}) \rightarrow [0,1]$ $\forall \boldsymbol{A} \subseteq \boldsymbol{X}$).*

As in Boolean logic, the *complement*, *union* and *intersection* can be defined for fuzzy sets. Let $\boldsymbol{X}$ be the universe of discourse, $\boldsymbol{A}$ and $\boldsymbol{B}$ be fuzzy subsets of $\boldsymbol{X}$, and $\overline{\mathbf{A}}$ be the compliment of $\boldsymbol{A}$ relative to $\boldsymbol{X}$. Let $\boldsymbol{x}$ be an element of $\boldsymbol{X}$.

We define the intersection between $\boldsymbol{A}$ and $\boldsymbol{B}$ as:

$$\mu_{A \cap B}(\mathbf{X}) \quad = \quad \min(\mu_A(\mathbf{x}), \mu_B(\mathbf{x})) \ \forall \ \mathbf{x} \in \mathbf{X}$$

since no element is more likely to be in the intersection than in one of the original sets.

We define the union between $\boldsymbol{A}$ and $\boldsymbol{B}$ as:

$$\mu_{A \cup B}(\mathbf{X}) \quad = \quad \max(\mu_A(\mathbf{x}), \mu_B(\mathbf{x})) \ \forall \ \mathbf{x} \in \mathbf{X}$$

since no element in the union can have a membership value that is less than the membership value of either of the original sets.

We define the complement of $\boldsymbol{A}$ as:

$$\mu_{\overline{A}}(\mathbf{x}) \quad = \quad 1 - \mu_A(\mathbf{x})$$

Fuzzy sets thus provides a way to represent vagueness and imprecision in sets.

2.5.2 Fuzzy Set Model for Retrieval

In the standard Fuzzy Set retrieval model, record terms can be weighted as in the vector space model (see Section 2.4). Then, given queries $(k_i \vee k_m)$, $(k_i \wedge k_m)$ and $(\neg k_i)$, and a record d_j with weights $w_{i,j}$ and $w_{m,j}$ associated with record terms k_i and k_m respectively, we can calculate the similarity between the queries and d_j as [SFW83]:

$$sim(d_j, q) = \max[w_{i,j}, w_{m,j}] \text{ for } q = (k_i \vee k_m) \tag{2.2}$$

$$sim(d_j, q) = \min[w_{i,j}, w_{m,j}] \text{ for } q = (k_i \wedge k_m) \tag{2.3}$$

$$sim(d_j, q) = 1 - w_{i,j} \text{ for } q = (\neg k_i)$$

It is easy to see that if terms are weighted only 0 or 1, the fuzzy set model reduces to the Boolean model. In fact, the fuzzy set model provides only a marginal improvement over the Boolean model, since the rank of a retrieved item only depends on lowest or highest term for **and** and **or** queries respectively. For this reason the fuzzy set model has never enjoyed much popularity with the information retrieval community.

2.6 Extended Boolean Model

2.6.1 Motivation

The Extended Boolean Model [SFW83] can be seen as a unification of the Boolean, Vector Space and Fuzzy Set Models. Consider table 2.1.

Three record classes can be identified for two-term queries: those containing both terms, those containing one of the terms and those containing neither of the terms. The similarity between query and record would thus be 1 for the **and** and the **or** queries if the record

	Terms		Similarity to Query	
Record	k_i	k_m	$(k_i \vee k_m)$	$(k_i \wedge k_m)$
d_1	1	1	1	1
d_2	1	0	1	0
d_3	0	1	1	0
d_4	0	0	0	0

Table 2.1: Conventional Boolean Retrieval

contains both terms, 0 for the **and** and 1 for the **or** query if the record contains one of the terms, and 0 for both queries if the record contains none of the terms. When only two terms are considered, the term assignment can be represented in a two-dimensional graph, with each axis assigned to a different term (see figure 2.1).

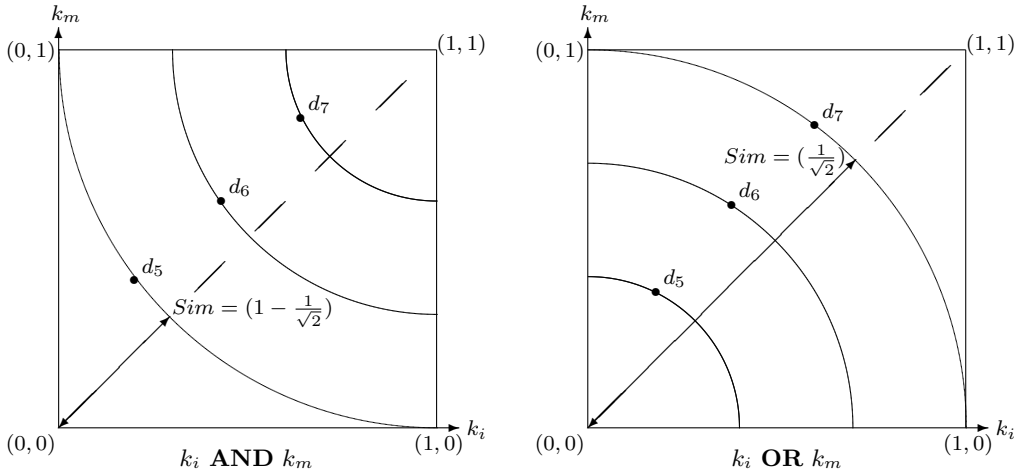

Figure 2.1: Equidistance lines from $(1,1)$ for **and** and from $(0,0)$ for **or**.

For an **and** query, the point $(1,1)$ is the desirable location, and for an **or** query, the point $(0,0)$ is the position to be avoided. Thus, a retrieval system could rank the results of a search in increasing order of distance from $(1,1)$ for an **and** query, and in decreasing order of distance from $(0,0)$ for **or** queries. Furthermore, because such a system no longer makes use of set operations that require a term to either be an element of a set or not; record terms could be weighted. For a record d_j with term weights $w_{i,j}$ and $w_{m,j}$ for terms k_i and k_m respectively, the distance from the point(0,0) is:

$$\sqrt{(w_{i,j} - 0)^2 + (w_{m,j} - 0)^2}$$

and the distance from point $(1, 1)$ is given by:

$$\sqrt{(1 - w_{i,j})^2 + (1 - w_{m,j})^2}$$

If we assume that $0 \leq w_{i,j} \leq 1$ for any i and k then these distances can be normalized by dividing by the maximum distance between $(0, 0)$ and $(1, 1)$, i.e. $\sqrt{2}$. The distances can be used to compute the similarity between a query q and a record d_j:

$$sim(d_j, q) = \sqrt{\frac{(w_{i,j})^2 + (w_{m,j})^2}{2}} \text{ for } q = (k_i \vee k_m) \quad (2.4)$$

$$sim(d_j, q) = 1 - \sqrt{\frac{(1 - w_{i,j})^2 + (1 - w_{m,j})^2}{2}} \text{ for } q = (k_i \wedge k_m) \quad (2.5)$$

Table 2.2 shows the similarity values calculated, using equations 2.4 and 2.5, for the three record classes described earlier.

	Terms		**Similarity to Query**	
Record	k_i	k_m	$(k_i \vee k_m)$	$(k_i \wedge k_m)$
d_1	1	1	1	1
d_2	1	0	$\frac{1}{\sqrt{2}}$	$1 - \frac{1}{\sqrt{2}}$
d_3	0	1	$\frac{1}{\sqrt{2}}$	$1 - \frac{1}{\sqrt{2}}$
d_4	0	0	0	0

Table 2.2: Extended Boolean Retrieval

As opposed to the Conventional Boolean Model, records that contain only one of the query terms receive a similarity value of $\frac{1}{\sqrt{2}}$ for **or** queries and $1 - \frac{1}{\sqrt{2}}$ for **and** queries. Note that the presence of one term in a record in an **or** query is worth less than the presence of both terms, but is still worth more than the presence of a single term in the case of an **and** query. The model also assigns a non-zero value to records that contain only one term in the case of an **and** query, which is a significant improvement over the Conventional Boolean Model.

Examples of three records, d_5, d_6 and d_7, containing weighted terms can be seen in figure 2.1. Lines representing equidistant points are used to illustrate the similarity values for each of the records. Note that:

$$sim(d_7, q) > sim(d_6, q) > sim(d_5, q) \text{ for } q = (k_i \vee k_m)$$

$$sim(d_7, q) > sim(d_6, q) > sim(d_5, q) \text{ for } q = (k_i \wedge k_m)$$

and that (for $q_1 = (k_i \vee k_m)$ and $q_2 = (k_i \wedge k_m)$):

$$sim(d_7, q_1) > sim(d_7, q_2)$$
$$sim(d_6, q_1) > sim(d_6, q_2)$$
$$sim(d_5, q_1) > sim(d_5, q_2)$$

Equations 2.4 and 2.5 can be extended to allow for query term weights, $0 \leq w_{i,q} \leq 1$ for term k_i in query q:

$$sim(d_j, q) = \sqrt{\frac{w_{i,q}^2.(w_{i,j})^2 + w_{m,q}^2.(w_{m,j})^2}{w_{i,q}^2 + w_{m,q}^2}} \text{ for } q = [(w_{i,q}, k_i) \vee (w_{m,q}, k_m)] \tag{2.6}$$

$$sim(d_j, q) = 1 - \sqrt{\frac{w_{i,q}^2.(1 - w_{i,j})^2 + w_{m,q}^2.(1 - w_{m,j})^2}{w_{i,q}^2 + w_{m,q}^2}} \text{ for } q = [(w_{i,q}, k_i) \wedge (w_{m,q}, k_m)] \tag{2.7}$$

Before continuing, the concept of the L_p vector norm will be briefly discussed [Nic90].

2.6.2 L_p Vector Norm

Definition 2.6.1 *A norm of a vector is defined as a function $f(\overrightarrow{x})$ (normally written $\| \overrightarrow{x} \|$) that satisfies the following conditions:*

1. $f(\overrightarrow{x}) \geq 0$
2. $f(\overrightarrow{x}) = 0 \Leftrightarrow \overrightarrow{x} = 0$
3. $f(\overrightarrow{x} + \overrightarrow{y}) \leq f(\overrightarrow{x}) + f(\overrightarrow{y})$
4. $f(|\boldsymbol{c}|\overrightarrow{x}) = |\boldsymbol{c}| f(\overrightarrow{x})$ *where* $\boldsymbol{c}$ *is a scalar*

In other words: Only the zero vector has norm zero, the norm of a vector must satisfy the triangle inequality *(3)*, and the norm must be homogeneous with respect to scalar

multiplication.

The most commonly used norm is the *Euclidean* norm which gives the length of vectors in Euclidian space and is defined as:

$$\| \overrightarrow{x} \| = \sqrt{\sum_{i=1}^{t} |x_i|^2}$$

where t is the number of vector elements. The *Euclidean* norm is a special case of the L_p norm that can be defined as:

$$\| \overrightarrow{x} \|_p = \sqrt[p]{\sum_{i=1}^{t} |x_i|^p}$$

The most interesting L_p norms are:

- $p = 1$: The *absolute value* norm.
- $p = 2$: The *Euclidean* norm.
- $p = \infty$: The *Chebyshev* norm.

Note that the case $p = \infty$ is a limiting case which becomes:

$$\| \overrightarrow{x} \|_\infty = \max(|x_1|, ..., |x_t|)$$

It is often useful to work with the normalized L_p norm:

$$\| \overrightarrow{x} \|_p \text{ (normalized)} = \sqrt[p]{\frac{\sum_{i=1}^{t} |x_i|^p}{t}} \tag{2.8}$$

where t is the number of vector elements.

2.6.3 The *p* Norm Model

Equations 2.4 and 2.5 were derived using the *Euclidean* norm. Since only two query terms were used, the equations can be seen as a special case ($t = 2$ and $p = 2$) of the more general equations in normalized L_p norm form:

$$sim(d_j, q) = \sqrt[p]{\frac{(w_{1,j})^p + ... + (w_{t,j})^p}{t}}$$

$$\begin{aligned} \text{for } q &= (k_1 \vee^p ... \vee^p k_t) \\ sim(d_j, q) &= 1 - \sqrt[p]{\frac{(1 - w_{1,j})^p + ... + (1 - w_{t,j})^p}{t}} \\ \text{for } q &= (k_1 \wedge^p ... \wedge^p k_t) \end{aligned} \tag{2.9}$$

(2.10)

where $\vee^p$ and $\wedge^p$ refer to **OR** and **AND** queries where p was selected as the vector norm.

Equations 2.9 and 2.10 can be generalized to account for weighted query terms as in equations 2.6 and 2.7:

$$sim(d_j, q) = \sqrt[p]{\frac{w_{1,q}^p.(w_{1,j})^p + ... + w_{t,q}^p.(w_{t,j})^p}{w_{1,q}^p + ... + w_{t,q}^p}} \quad \text{for } q = [(w_{1,q}, k_1) \vee^p ... \vee^p (w_{t,q}, k_t)] \tag{2.11}$$

$$sim(d_j, q) = 1 - \sqrt[p]{\frac{w_{1,q}^p.(1 - w_{1,j})^p + ... + w_{t,q}^p.(1 - w_{t,j})^p}{w_{1,q}^p + ... + w_{t,q}^p}} \quad \text{for } q = [(w_{1,q}, k_1) \wedge^p ... \wedge^p (w_{t,q}, k_t)] \tag{2.12}$$

The value of p can be varied to simulate different retrieval models.

$p = 1$

When p is set equal to 1, it can be shown that both equations 2.11 and 2.12 reduce to:

$$\begin{aligned} sim(d_j, q) &= \frac{w_{1,q}.(w_{1,j}) + ... + w_{t,q}.(w_{t,j})}{w_{1,q} + ... + w_{t,q}} \\ &= \frac{\sum_{i=1}^{t} w_{i,j}.w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,q}^2}} \end{aligned} \tag{2.13}$$

where t is the total number of record terms.

Apart from not normalizing each record vector, the Extended Boolean Model thus becomes equivalent to the Vector Space Model (compare equations 2.1 and 2.13).

$p = \infty$

When p tends to ∞ and the query terms are equally weighted, it can be shown that equations 2.11 and 2.12 reduce to equations 2.2 and 2.3 respectively. Thus for $p = \infty$ the Extended Boolean Model becomes equivalent to the Fuzzy Set Model, and by extension, the conventional Boolean Model.

2.6.4 Implications

By varying p between 1 and infinity, the ranking behavior of the Extended Boolean Model can be changed from that of the Vector Space Model to that of the Boolean and Fuzzy Set Models. Table 2.3 shows how changing the value of p will affect a retrieval system.

p Value	Operator	Written	Result
∞	**AND**	$\wedge^{\infty}$	Item not retrieved unless all query terms present
∞	**OR**	$\vee^{\infty}$	At least one of each group of query terms is required
3	**AND**	$\wedge^{3}$	Presence of all terms worth more than presence of only some
3	**OR**	$\vee^{3}$	Presence of several terms worth more than presence of only one
1	**AND**, **OR**	$\vee^{1}$, $\wedge^{1}$	Queries are ranked only on term weights

Table 2.3: p-Value Interpretation

Another advantage to the the Extended Boolean Model is that it does not exclude queries like:

$$(k_1 \vee^{2} k_2) \wedge^{\infty} k_3$$

i.e. it is possible to define which retrieval model should be used for specific sections of the query.

2.7 Probabilistic Model

The Probabilistic Model was developed Robertson and Jones ([RJ88] in [RBY99]) for systems where the user does not have a definite idea of which index terms would appear in the relevant records. The user would thus start a retrieval process by making a guess of which index terms would appear in relevant records. A list of relevant records is displayed, and

the user then selects which records are most relevant. The information in these records is then used to refine the query and thus return a more accurate result set. This process can be repeated as many times as necessary. The main advantage of the Probabilistic Model is that records are ranked in decreasing order of their probability of being relevant.

The goal of the probabilistic model is to estimate the probability that the user will find record d_j relevant, given query q. It is assumed that relevance only depends on the information in the record and the query, and that there is a subset of all records, $\boldsymbol{R}$, that the user prefers in response to query q. $\boldsymbol{R}$ is thus the set of all relevant records.

It now remains to show how the probability of relevance should be calculated for a record d_j, given query q. Consider the following definition [RBY99]:

Definition 2.7.1 *For the Probabilistic Model all index term weights are binary i.e.,* $w_{i,j} \in \{0,1\}$ *and* $w_{i,q} \in \{0,1\}$. *A query* q *is a subset of index terms. Index term vectors are set up for records as in the Vector Space Model. Let* $\boldsymbol{R}$ *be the set of records known (initially guessed) to be relevant. Let* $\overline{\boldsymbol{R}}$ *be the compliment of* $\boldsymbol{R}$ *(i.e. the set of non-relevant records). Let* $P(\boldsymbol{R}|\overrightarrow{d_j})$ *be the probability that record* d_j *is relevant to the query* q *and* $P(\overline{\boldsymbol{R}}|\overrightarrow{d_j})$ *be the probability that* d_j *is not relevant to* q.

We will originally define:

$$sim(d_j, q) = \frac{P(\boldsymbol{R}|d_j)}{P(\overline{\boldsymbol{R}}|d_j)}$$

Using Bayes' rule it can be rewritten as:

$$sim(d_j, q) = \frac{P(\overrightarrow{d_j}|\boldsymbol{R}) \times P(\boldsymbol{R})}{P(\overrightarrow{d_j}|\overline{\boldsymbol{R}}) \times P(\overline{\boldsymbol{R}})}$$

where

- $P(\overrightarrow{d_j}|\boldsymbol{R})$ is the probability of randomly selecting the record d_j from the set $\boldsymbol{R}$ of relevant records.
- $P(\boldsymbol{R})$ is the probability of randomly selecting a relevant record from the entire record set.

- $P(\overrightarrow{d_j}|\overline{\boldsymbol{R}})$ is the probability of randomly selecting the record d_j from the set $\overline{\boldsymbol{R}}$ of non-relevant records.
- $P(\overline{\boldsymbol{R}})$ is the probability of randomly selecting a non-relevant record from the entire record set.

Since $P(\boldsymbol{R})$ and $P(\overline{\boldsymbol{R}})$ are constants, we redefine the similarity function as:

$$sim(d_j, q) = \frac{P(\overrightarrow{d_j}|\boldsymbol{R})}{P(\overrightarrow{d_j}|\overline{\boldsymbol{R}})} \tag{2.14}$$

Assuming independence of index terms, equation 2.14 can be rewritten as:

$$sim(d_j, q) = \frac{\left(\prod_{w_{i,j}=1} P(k_i|\boldsymbol{R})\right) \times \left(\prod_{w_{i,j}=0} P(\overline{k_i}|\boldsymbol{R})\right)}{\left(\prod_{w_{i,j}=1} P(k_i|\overline{\boldsymbol{R}})\right) \times \left(\prod_{w_{i,j}=0} P(\overline{k_i}|\overline{\boldsymbol{R}})\right)} \tag{2.15}$$

where

- $P(k_i|\boldsymbol{R})$ is the probability of term k_i being present in a randomly selected record from $\boldsymbol{R}$.
- $P(\overline{k_i}|\boldsymbol{R})$ is the probability that term k_i is not present in a record randomly selected from $\boldsymbol{R}$.
- $P(k_i|\overline{\boldsymbol{R}})$ is the probability of term k_i being present in a randomly selected record from $\overline{\boldsymbol{R}}$.
- $P(\overline{k_i}|\overline{\boldsymbol{R}})$ is the probability that term k_i is not present in a record randomly selected from $\overline{\boldsymbol{R}}$.

Since all term weights are binary, equation 2.15 can be rewritten as:

$$sim(d_j, q) = \frac{\left(\prod_{i=1}^{t} P(k_i|\boldsymbol{R})^{w_{i,j}}\right) \times \left(\prod_{i=1}^{t} P(\overline{k_i}|\boldsymbol{R})^{(1-w_{i,j})}\right)}{\left(\prod_{i=1}^{t} P(k_i|\overline{\boldsymbol{R}})^{w_{i,j}}\right) \times \left(\prod_{i=1}^{t} P(\overline{k_i}|\overline{\boldsymbol{R}})^{(1-w_{i,j})}\right)} \tag{2.16}$$

Taking logarithms on both sides of equation 2.16 we get:

$$\begin{aligned}\log(sim(d_j,q)) &= \sum_{i=1}^{t} w_{i,j}\log P(k_i|\boldsymbol{R}) \\ &+ \sum_{i=1}^{t}(1-w_{i,j})\log P(\overline{k_i}|\boldsymbol{R}) \\ &- \sum_{i=1}^{t} w_{i,j}\log P(k_i|\overline{\boldsymbol{R}}) \\ &- \sum_{i=1}^{t}(1-w_{i,j})\log P(\overline{k_i}|\overline{\boldsymbol{R}}) \end{aligned} \tag{2.17}$$

which can be rewritten as:

$$\begin{aligned}\log(sim(d_j,q)) &= \sum_{i=1}^{t} w_{i,j}\log P(k_i|\boldsymbol{R}) \\ &+ \sum_{i=1}^{t}\log P(\overline{k_i}|\boldsymbol{R}) \\ &- \sum_{i=1}^{t} w_{i,j}\log P(\overline{k_i}|\boldsymbol{R}) \\ &- \sum_{i=1}^{t} w_{i,j}\log P(k_i|\overline{\boldsymbol{R}}) \\ &- \sum_{i=1}^{t}\log P(\overline{k_i}|\overline{\boldsymbol{R}}) \\ &+ \sum_{i=1}^{t} w_{i,j}\log P(\overline{k_i}|\overline{\boldsymbol{R}}) \end{aligned} \tag{2.18}$$

The terms $\sum_{i=1}^{t}\log P(\overline{k_i}|\boldsymbol{R})$ and $\sum_{i=1}^{t}\log P(\overline{k_i}|\overline{\boldsymbol{R}})$ will be the same for all records in the context of a specific query, and can thus be ignored. Recall that $P(k_i|\boldsymbol{R}) + P(\overline{k_i}|\boldsymbol{R}) = 1$. Defining a new similarity function as the logarithm of the previous similarity function and multiplying with the weights of the query terms to exclude non query terms, we can now define:

$$sim(d_j,q) = \sum_{i=1}^{t} w_{i,q}.w_{i,j}\left(\log\frac{P(k_i|\boldsymbol{R})}{1-P(k_i|\boldsymbol{R})} + \log\frac{1-P(k_i|\overline{\boldsymbol{R}})}{P(k_i|\overline{\boldsymbol{R}})}\right) \tag{2.19}$$

Equation 2.19 is used as the similarity function of the probabilistic model.

To retrieve the first set of records, some initial values for $P(k_i|\boldsymbol{R})$ and $P(k_i|\overline{\boldsymbol{R}})$ must be selected. Normally it is assumed that the probability of randomly selecting a query term

that is in a relevant record is some constant (typically 0.5), thus:

$$P(k_i|\boldsymbol{R}) = 0.5$$

and it is assumed that the distribution of index terms among the non-relevant records is equal to the distribution of the index terms among all the records in the collection, thus:

$$P(k_i|\overline{\boldsymbol{R}}) = \frac{n_i}{N}$$

where n_i is the number of records that contain term k_i and N is the total number of records in the collection. After the initial set of records are retrieved, the estimations made for the values of $P(k_i|\boldsymbol{R})$ and $P(k_i|\overline{\boldsymbol{R}})$ can be improved as follows. Let the set $\mathcal{V}$ be the set of records that the user judged as most relevant of the total set of retrieved records (user intervention is not strictly necessary: $\mathcal{V}$ could be created by simply selecting the top n highest ranked records, where n is a predefined threshold). Let the set $\mathcal{V}_i \subseteq \mathcal{V}$ be the set of records that contain term k_i. We then approximate $P(k_i|\boldsymbol{R})$ by the distribution of term k_i among the records retrieved:

$$P(k_i|\boldsymbol{R}) = \frac{|\mathcal{V}_i|}{|\mathcal{V}|}$$

$P(k_i|\overline{\boldsymbol{R}})$ can be approximated by considering that all non-retrieved records are not relevant:

$$P(k_i|\overline{\boldsymbol{R}}) = \frac{n_i - |\mathcal{V}_i|}{N - |\mathcal{V}|}$$

This process can be repeated recursively. To cater for small values of $|\mathcal{V}|$ and $|\mathcal{V}_i|$, an adjustment factor can be added to the equations:

$$P(k_i|\boldsymbol{R}) = \frac{|\mathcal{V}_i| + \frac{n_i}{N}}{|\mathcal{V}| + 1} \tag{2.20}$$

$$P(k_i|\overline{\boldsymbol{R}}) = \frac{n_i - |\mathcal{V}_i| + \frac{n_i}{N}}{N - |\mathcal{V}| + 1} \tag{2.21}$$

The Probabilistic Model has several disadvantages:

1. The need to guess the initial relevance of the index terms.

2. The fact that all term weights have to be binary.

3. The assumption that terms are independent (whether this is really a disadvantage is debatable).

2.8 Structured Text Retrieval Models

Normally the field of information retrieval is concerned with retrieving textual records. These records are not structured in the same way as records in a relational database would be, where information is already formatted and is meant to be retrieved by means of a key attribute [NBY95]. Some structure can, however, be found in a textual database, for example: chapters, sections, titles, etc. The user may find it useful to be able to specify information on the structure of the record to be retrieved. An example of such a query is: **Chapter**(*information*, *database*, **Title**(*Retrieval*)) i.e., the user is looking for a chapter that contains the words *information* and *database* and with the word *Retrieval* in the title.

To a certain extent the records in the GDB are structured in fields as one would expect in a relational database. However, this structure is not always useful for retrieval purposes, because of inaccuracies in the data and the limited amount of information that the user may have when searching for an individual. For example, the user may know that an individual was baptized in the town Queenstown and may then naturally assume that the individual was also born in the same town. If the individual was in fact born in another town, a search on birth place will not retrieve the desired record. Furthermore, many of the fields in the GDB contain natural language text and can thus not be seen as a field in the same sense as a field in the relational model. By integrating structured text retrieval methods, the above problems may be overcome.

A system that allows the user to specify the content and structure of a record in a query would contain only records that satisfy the query in its result set, and can thus be described as a data retrieval system. By searching for records that match the structure of the query only partially and ranking the result set, the system can be seen as an information retrieval system [RBY99].

It is only in recent years that research has been done in structured text retrieval. According to Navarro and Baeza-Yates in [NBY97], the models proposed are not as mature as some classical models like the Boolean or Vector Space models. Several structured text

retrieval models were compared in [BYN96], and it is interesting to note that only three of the compared models have $O(n)$ efficiency.

2.8.1 Tree Matching Model

After considering several models it was concluded that the Tree Matching Model proposed by Kilpeläinen and Mannila in [KM93] is one of the more versatile models and would be best suited to be implemented in the GIS. This model is similar (but more powerful) to the model proposed by Burkowski [Bur92].

The Tree Matching Model makes use of the fact that information in a database can often be broken up into an hierarchial structure. A document that can be broken up into sections, paragraphs, sentences and words would be represented by a tree structure as shown in figure 2.2.

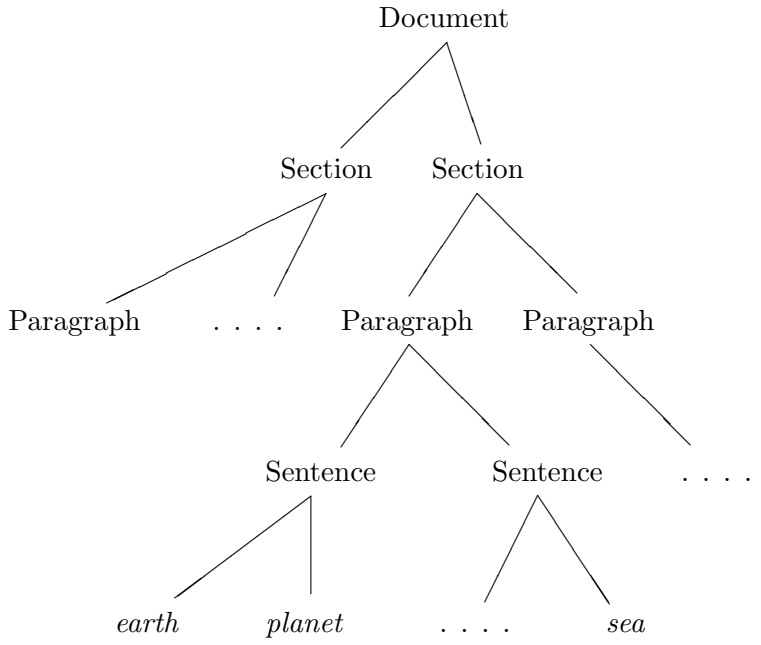

Figure 2.2: The Tree Hierarchy of a simple document.

The tree hierarchy makes it possible to formulate queries as trees, where the sub-nodes for each node are given in brackets, for example:

Paragraph(*sea*, (**Sentence**(*earth*, *planet*)))

The tree for the above query can be seen in figure 2.3.

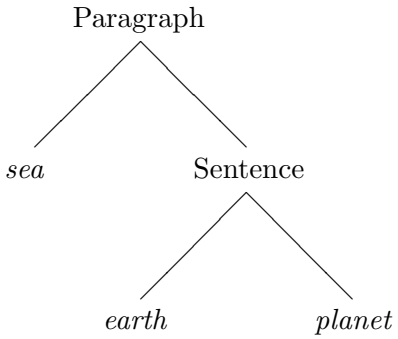

Figure 2.3: The Tree Hierarchy of a Query.

The model allows the query depicted in figure 2.3 to be located in the document depicted in figure 2.2 as shown in figure 2.4. This is called Tree Inclusion. Note that the word *sea* was matched with the corresponding word in the hierarchy, despite the fact that a **Sentence** node exists above *sea* in the document tree, but not in the query tree.

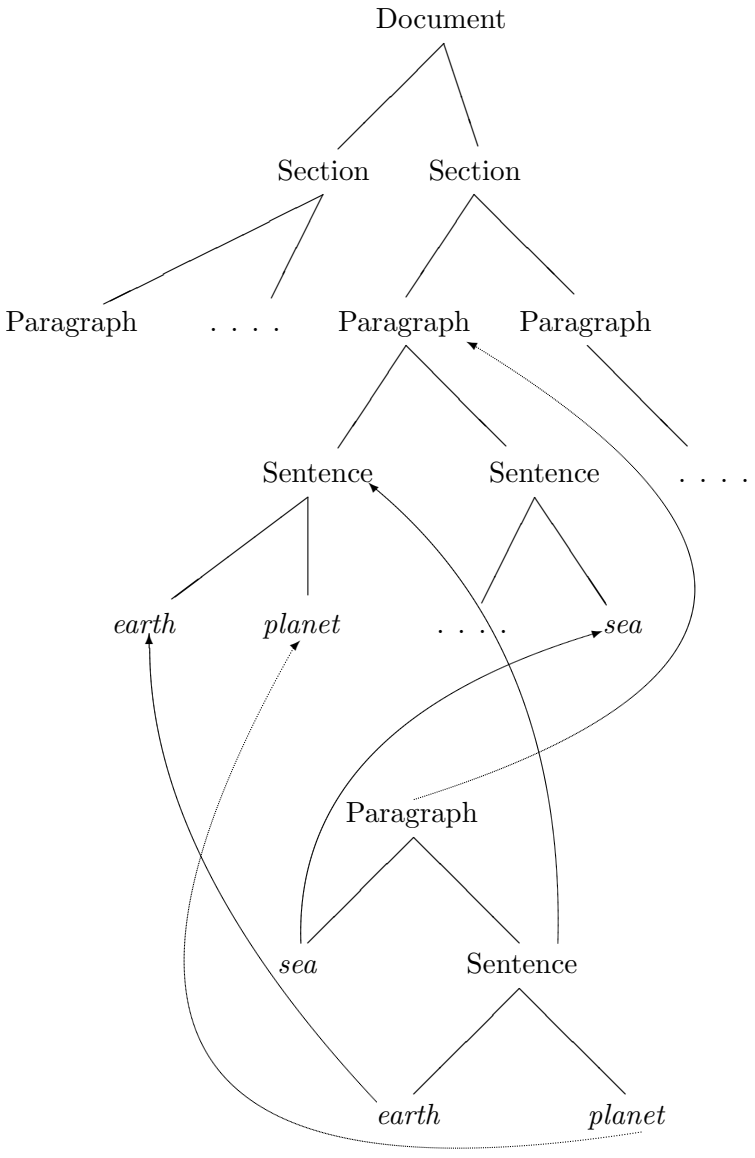

Figure 2.4: Tree Inclusion.

As illustrated in figure 2.4, the goal of the model is identifying ancestorship and labels (i.e. leaf nodes) rather than a direct hierarchial match. The model seeks minimal subtrees of the target. More formally, an *included tree* of a tree T consists of a set of nodes that appear in a T with similar hierarchial relationships. The trees considered here are ordered. We can say that if the left-to-right order of the nodes is the same as in T, the included tree is an *ordered included tree* of T. Note that *ordered tree inclusion* is a stricter form of inclusion than *unordered tree inclusion*. Figures 2.5 and 2.6 illustrate *ordered tree inclusion* and *unordered tree inclusion* respectively.

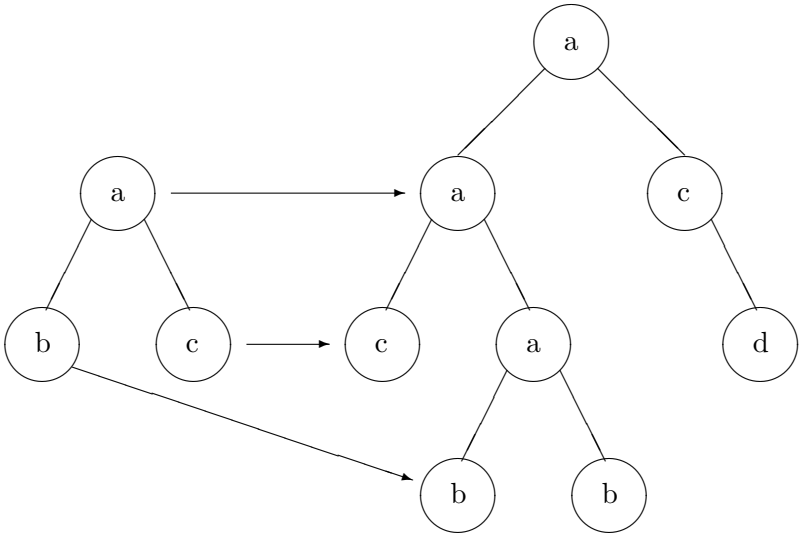

Figure 2.5: Unordered tree inclusion.

The tree matching model also allows for the use of variables in queries, for example

Paragraph(x, (**Sentence**(*earth*, x)))

so that the value of variable x does not explicitly have to be stated. Queries of this form are thus more focused on the structure of the tree than the content.

Unfortunately, even without allowing for variables in queries, unordered tree inclusion is an NP-complete problem. Ordered tree inclusion can be calculated in polynomial time. Tree inclusion is thus a very expensive retrieval model. However, by appropriately indexing the database, and making the assumption that the database does not contain recursive

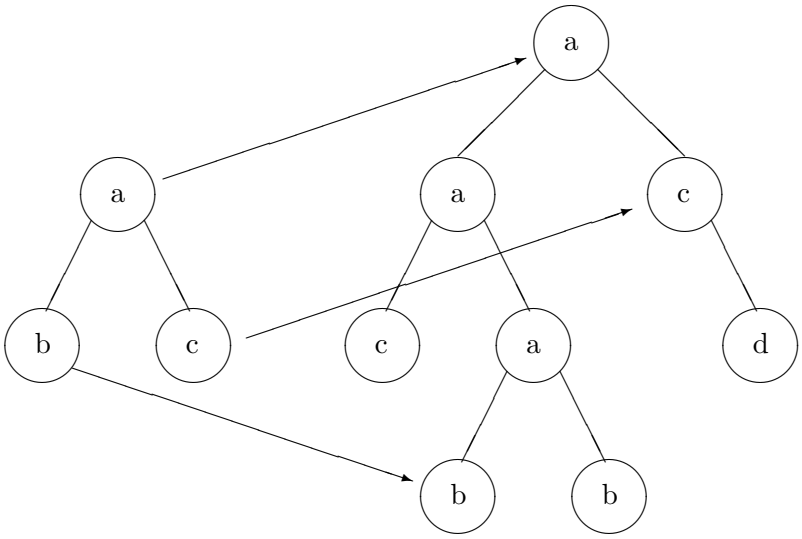

Figure 2.6: Ordered tree inclusion.

hierarchies (i.e. no structure contains itself as a substructure), it has been shown that inclusion queries can be solved in O(n) time.

Yan and Annevelink [YA94] describes the successful implementation of the Tree Matching model in a object oriented database system.

2.9 Neural Network Model

Neural Networks are an attempt to model the human brain using a computer [HK01]. The brain can be very simply seen as a vast interconnected network of cells, called neurons. Each connection to a neuron (called a synapse) can be seen as either an input or output connection. The value of the output is determined by the values of the inputs. An elementary mathematical model for a neuron could thus be that the output is equal to the sum of the inputs, i.e.

$$v = \sum_{i=1}^{n} x_i \tag{2.22}$$

where v is the output, n is the number of inputs and x_i is a specific input.

In practice it was found that the synapses in the brain are of different thicknesses, thus implying that not all inputs are equally important. Equation 2.22 is thus adapted to allow inputs to be weighted:

$$v = \sum_{i=1}^{n} w_i x_i \tag{2.23}$$

where w_i is a weight.

A network is formed by connecting the outputs of some neurons to the inputs of others. Some neurons will receive input from outside the network. These neurons are called input neurons. Other neurons, called the output neurons, provide the output for the entire network. Input is provided to input neurons, whereupon some of them will fire, i.e. provide input to other nodes. The output of the network would be the output of the output node or nodes.

An advantage of neural networks is that they can be *trained* to perform a certain task (as opposed to a human programmer coding a program to perform a task). A network can also be created in such a way as to learn from mistakes and continuously improve after user feedback.

An information retrieval system is implemented in three layers [WH91]: the query term layer, the record term layer and the record layer. The process starts off with the query term nodes sending signals to the record term nodes, which in turn send signals to the records. These signals are weighted (a weighting scheme similar to that of the Vector Space Model can be used). Signals received by the record nodes are summed for each node and the result used in the ranking of the records. It can be shown that the process as described this far is equivalent to the Vector Space Model. The models deviate in that after the initial signals reach the record nodes, the record nodes in turn may send signals back to the record term nodes, starting a cycle. Terms not appearing in the original query may thus be activated. The signals sent out by the record nodes become weaker each cycle so that the process eventually halts.

The Neural Network model is not popular among the information retrieval community, largely because the model provides very little, if any, improvement in retrieval performance over the Vector Space Model. Furthermore, for a database containing a large number of records, a very large number of neurons would be needed, which in turn implies long periods of training time.

2.10 Bayesian Network Model

According to [TC90], a Bayesian inference network is a directed, acyclic dependency graph in which nodes represent propositional variables or constants and edges represent dependence relations between propositions. If a proposition represented by a node p "causes" or implies the proposition represented by node q, we draw a directed edge form p to q. The node q contains a *link* matrix that specifies $P(q|p)$ for all possible values of the two variables. When a node has multiple parents, the link matrix specifies the dependence of that node on the set of parents and characterizes the dependence relationship between that node and all nodes representing its potential causes. Given a set of prior probabilities for the roots of the network, these networks can be used to compute the probability associated with all remaining nodes.

The Bayesian Network Retrieval Model can be very briefly described as follows: A Bayesian inference network is set up with the records in the collection as the root nodes. The child nodes are used to describe the records using a variety of representation techniques. The network just described is called the Document Network. The Document Network is created only once for a specific collection. Edges from the Document Network are connected to the Query Network, which is essentially a representation of the user's information need, which is the final node in the Query Network. The Query Network is modified during query processing as more queries are added or queries are refined. In short, an attempt is made to define a specific information need in terms of a tree of which the roots are the records in which the user is interested. By receiving relevance feedback from the user and by incorporating changes to the user's query, the Query Network can be improved after each search.

A major strength of the Bayesian Network model is the fact that the relevance of a record to a query can be inferred (for example, records that are relevant to query although none of the query terms appear in the record, can be returned). However, in the case of the GIS, where locating a record is essentially a data retrieval problem, inferring relevance can be considered a weakness. This reason, coupled with the fact that the Bayesian Network Model is used very seldom in practice, led to the model not being seriously considered as a model for the search algorithms in the GIS.

2.11 Conclusions

Several well known IR Models were presented in this chapter. Unfortunately, no single model is appropriate for use in the GIS. The functionality provided by Structured Text models, when applied to the fields present in the records contained in the GDB, would greatly enhance a search algorithm for the GIS. The flexibility of the Extended Boolean model makes it a prime candidate for implementation in the GIS. It was thus decided to create a hybrid algorithm of the Extended Boolean and Structured Text models.

Chapter 3

SEARCH ALGORITHMS IN GIS

3.1 Full name

The problem of searching for an individual only on full name has long been investigated at UPE [DK04]. The conclusions of that researched will be discussed in this section.

3.1.1 Definition

The full name search problem can be formally stated as follows [DK02]:

Definition 3.1.1 *Let $\mathcal{N}$ be the universal set of all names (this set includes all surnames and forenames and their variations). A full name is defined as $n = n_1, n_2, n_3, ..., n_k$ with $k \geq 1$, $n_i \in \mathcal{N}$ for $i = 1, 2, ..., k$ and n_1 is the surname. Let $\mathcal{F} \subseteq \mathcal{N} \times \mathcal{N} \times \mathcal{N} \times ... \times \mathcal{N}$ be the set of all full names, and let $\mathcal{V} \subseteq \mathcal{N}$ be the set of all names occurring in the GDB, and let $F \subseteq \mathcal{V} \times \mathcal{V} \times \mathcal{V} \times ... \times \mathcal{V} \subseteq \mathcal{F}$ be the set of full names contained in the database. Given a full name $n \in \mathcal{F}$ a search algorithm should return records containing the closest full names from the set $\mathcal{F}$. Assume a distance function d, between any two full names, i.e. $d : \mathcal{F} \times \mathcal{F} \rightarrow \mathcal{I}$, where $\mathcal{I}$ is a positive closed real interval starting at zero, and $d(n, m) \geq 0$, $d(n, n) = 0$ and $d(n, m) = d(m, n)$.*
The problem can now be defined as: Determine the set of records defined by $M = \{x : x \in F,\ d(n, x) \leq t\}$, for a full name, $n \in \mathcal{F}$, where t is some small threshold value. Sorting the set of all $x \in M$ in increasing order of $d(n, x)$ yields the search results in order of most likely to least likely.

Several attempts were made to define a distance function, see [DK88], [Ple91], [DKDP93], and an attempt by [Chi96] based on the work of [WF74], [LW75] and [Hal80]. Unfortunately an adequate distance function was not found. Failure to do so has prompted a different approach to searching on full name.

3.1.2 Equivalent Classes

The first approach followed involved grouping names into equivalent classes, for example, the names *De Kock*, *Kock* and *Kok* would be classified as being equivalent. This was a manual process, originally assisted by the soundex algorithms developed by Du Plessis [Ple91].

Each equivalent class has a characteristic name, ideally the root name of all the names in the class. Essentially, an index entry can be created for each full name in the database by concatenating the characteristic name of each name in the full name. A full name query would also be converted to a concatenation of characteristic names and the resultant string can then be compared to all index entries to find a match.

Formally: Partition $\mathcal{V}$ into equivalent classes, $\mathcal{V}_i$ for $i = 1, 2, ..., N$, by grouping spelling variations, aliases and derivatives of names, such that

$$\mathcal{V} = \bigcup_{i}^{N} \mathcal{V}_i \text{ where } \mathcal{V}_i \cap \mathcal{V}_j = \phi \ \forall \ i \neq j \tag{3.1}$$

i.e. no name is a member of more than one equivalent class.

The characteristic name of a equivalent class $\mathcal{V}_i$ is denoted by c_i. Let $\mathcal{C} = \{c_1, c_2, ..., c_N\}$ then define the function $C : \mathcal{N} \rightarrow \mathcal{C}$ as

$$\begin{aligned} C(x) &= c_i \text{ if } x \in \mathcal{V}_i \\ &= c_j \text{ if } x \notin \mathcal{V} \text{ where } x \preceq y, y \in \mathcal{V}_j \text{ and there is no other} \\ &\quad z \in \mathcal{V} \text{ such that } x \preceq z \preceq y \end{aligned} \tag{3.2}$$

In the above equation, the lexicographical ordering on the name set, $\mathcal{V}$, is denoted by $\preceq$.

The full name index entry for the full name $n = n_1, n_2, ..., n_k \in F$ consists of the concatenation of $C(n_i)$ for $i = 1$ to k. Each $C(n_i)$ must be padded with blanks until it is equal to some chosen length. Note that the index can contain duplicates. The birthdate of the relevant person can be concatenated to the end of the entry so that people with the same name will be ordered in the index by birthdate.

A query on full name $m = m_1, m_2, ..., m_k \in \mathcal{F}$ is also converted to a search key by concatenating $C(m_i)$ for $i = 1$ to k.

$\mid \mathcal{V}_i \mid$	f	$f\%$	$\sum f\%$	$P = \mid \mathcal{V}_i \mid f$	$P\%$	$\sum P\%$	NO	$\sum NO\%$
≥13	236	1.06%	1.06%	5 236	13.08%	13.08%	828436	52.95%
12	28	0.13%	1.19%	336	0.84%	13.92%	21778	54.34%
11	35	0.16%	1.35%	385	0.96%	14.88%	30066	56.27%
10	58	0.26%	1.61%	580	1.45%	16.33%	36823	58.62%
9	71	0.32%	1.93%	639	1.60%	17.93%	52344	61.96%
8	64	0.29%	2.21%	512	1.28%	19.21%	17835	63.10%
7	133	0.60%	2.81%	931	2.33%	21.53%	46475	66.07%
6	181	0.81%	3.63%	1086	2.71%	24.25%	37825	68.49%
5	313	1.41%	5.03%	1565	3.91%	28.16%	82863	73.79%
4	536	2.41%	7.45%	2144	5.36%	33.51%	55489	77.34%
3	1181	5.31%	12.76%	3543	8.85%	42.37%	119305	84.96%
2	3677	16.54%	29.30%	7354	18.37%	60.74%	132340	93.42%
1	15713	70.70%	100.00%	15713	39.26%	100.00%	102943	100.00%
	22226	100.00%		40024	100.00%		1564522	

Table 3.1: Class size distribution

Table 3.1 depicts the distribution of names into equivalence classes in June 2004 [DK04]. Column $\mid \mathcal{V}_i \mid$ lists the size of the classes, f lists the number of classes of size $\mid \mathcal{V}_i \mid$, $f\%$ lists the number of classes as a percentage, $\sum f\%$ gives an accumulated percentage, $P = \mid \mathcal{V}_i \mid f$ lists the number of names in each set of classes, $P\%$ gives that number as an percentage, $\sum P\%$ gives the accumulated percentage, NO lists the number of times that names from each class occurs in the database and $\sum NO\%$ gives the accumulated percentage of column NO.

There are 22226 different equivalent classes in the GIS. The largest class contains 117 names and there are 15713 classes containing only one name. When a new name is added to the GDB, it is placed in an equivalent class of its own. New names must be manually

placed in existing equivalent classes.

Although only 13.0% of all distinct names in the GDB occur in equivalent classes that contain more than 13 names each, these names account for 52.95% of names occurring in the GDB. In other words, assuming that the occurrence frequency is an indication of the probability that a name will a appear in a query, then most queries will be made more effective by using the equivalent name index.

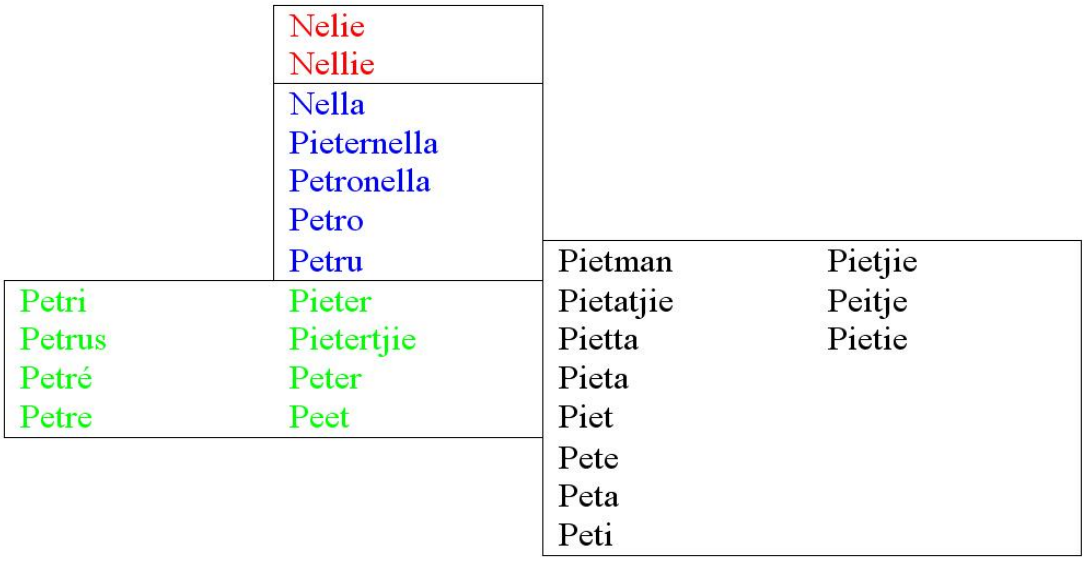

Figure 3.1: An example of equivalent classes

3.1.3 Similarity Sets

A drawback of the equivalent name index is that the equivalent classes are too rigid [DK04]. Consider the equivalent classes depicted in figure 3.1. The name *Pieter* is placed in a different class to *Pieternella*. Clearly these names are very similar, and a search for *Pieter* should also yield hits like *Pieternella*. Simply moving one of these names to the other class is not an option, since this will cause the same problem with different classes of names. Combining both classes into the same class would negatively affect searches,

since a search for *Peet* would, amongst others, yield *Petro*. A solution was proposed by having overlapping *similarity sets* associated with names, a simple example of which can be seen in figure 3.2 (colours indicate the original equivalent classes).

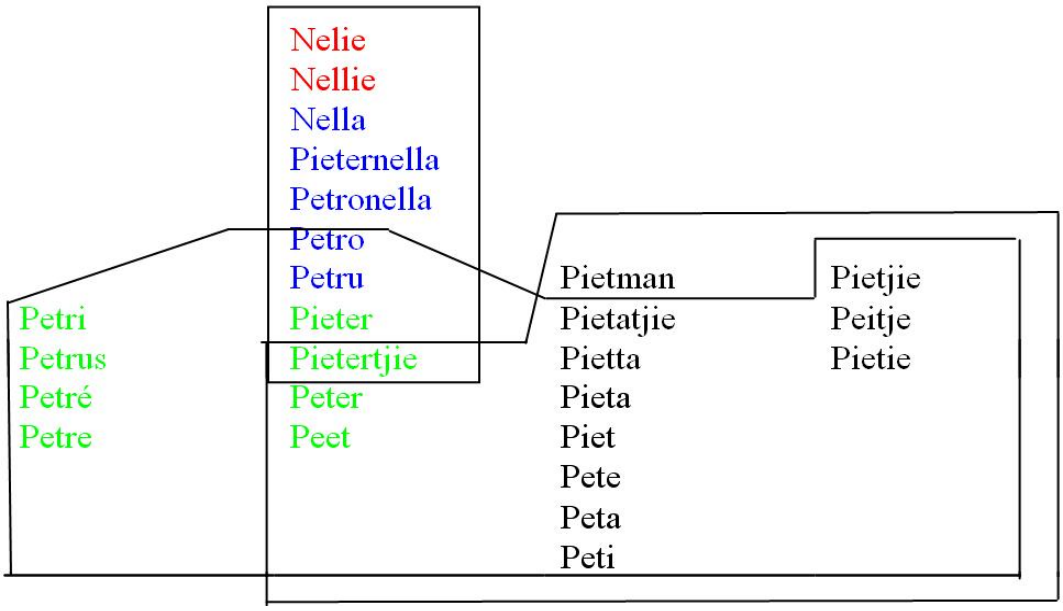

Figure 3.2: An example of similarity sets

Formally: Construct similarity sets, $\{\mathcal{S}_j : j = 1, 2, ..., K\}$, for all names in $\mathcal{V}$ such that

$$\mathcal{V} = \bigcup_{j=1}^{K} \mathcal{S}_j \text{ with } N \leq K \ll \mid \mathcal{V} \mid \tag{3.3}$$

Note that these sets are not disjoint, but it is assumed none of the sets is a subset of another.

An unique characteristic name, sc_j is associated with each S_j. For each name, $x \in \mathcal{V}$, the characteristic names of all the similarity sets of which it is a member, is stored in a database . The function $E : \mathcal{N} \rightarrow 2^{\mathcal{C}}$ is defined as:

$$\begin{aligned} E(x) &= \bigcup_{x \in \mathcal{S}_j} \{sc_j\} \text{ if } x \in \mathcal{V} \\ &= E(y) \text{ if } x \notin \mathcal{V} \text{ where } x \preceq y, y \in \mathcal{V} \text{ and there is no other} \\ &\quad z \in \mathcal{V} \text{ such that } x \preceq z \preceq y \end{aligned} \tag{3.4}$$

The first characteristic name stored for x will be $C(x)$, i.e. the equivalent name for x. Equivalent classes are thus retained. For each full name $n = n_1, n_2, ..., n_k \in \mathcal{F}$, entries are made into the index with the keys $E(n_1) \times E(n_2) \times ... \times E(n_k)$.

The search key for a search on the full name $m = m_1, m_2, ..., m_k \in \mathcal{F}$ is given by $E(m_1) \times E(m_2) \times ... \times E(m_k)$.

3.1.4 Alternative Similarity Sets

The similarity sets as formulated by De Kock [DK04] and described in the previous section have two major drawbacks:

1. Multiple search keys are generated when searching for a search on full name. This makes it impossible to define a lexicographical *next* or *previous* on the search results.

2. The search index no longer contains only equivalent names. As a result, the index is much larger.

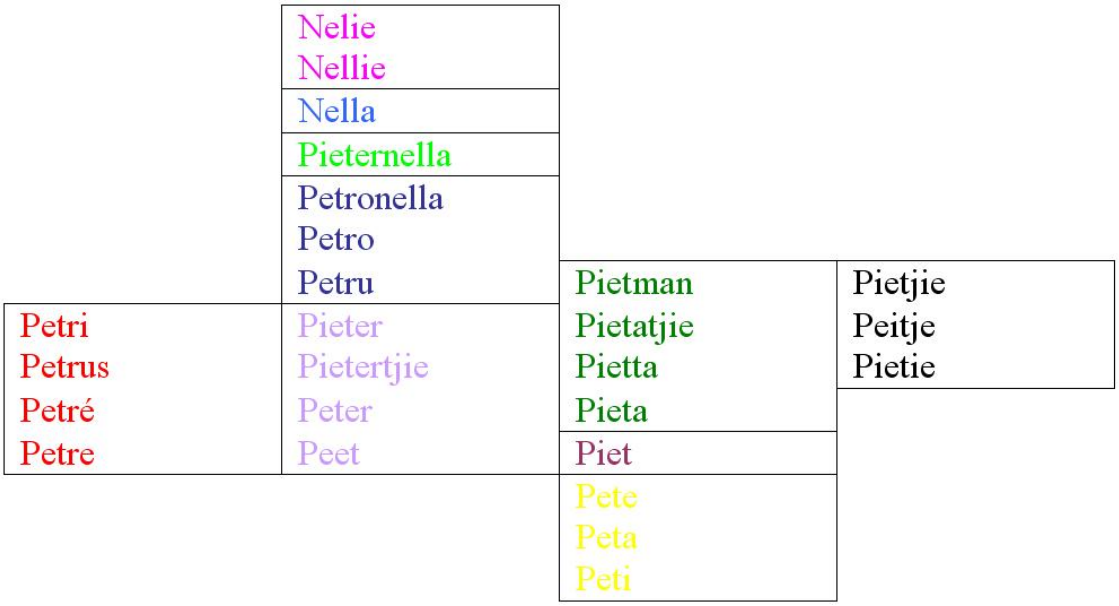

Figure 3.3: Smaller Equivalent Groups

In attempt to address the above problems, an alternative strategy for creating similarity sets will be proposed here. Firstly, reduce the size of equivalent classes, so that they only contain names that are undebatably equivalent. Such a subdivision can be seen in figure 3.3 (compare with figure 3.1).

We now define a similarity set as a set that is formed by associating each name in $\mathcal{V}$ with more than one equivalent class (any name is, by default, associated with its own equivalent class). Venn diagrams of the original and alternative similarity sets can be seen in figures 3.4 and 3.5, where similarity sets are depicted in blue and equivalent classes in yellow.

Formally, equation 3.3 is redefined for the set of similarity sets, $\{S_x : x \in \mathcal{V}\}$, as:

$$\mathcal{V} = \bigcup_{j=1}^{K} \mathcal{S}_j \text{ with } K = |\mathcal{V}| \tag{3.5}$$

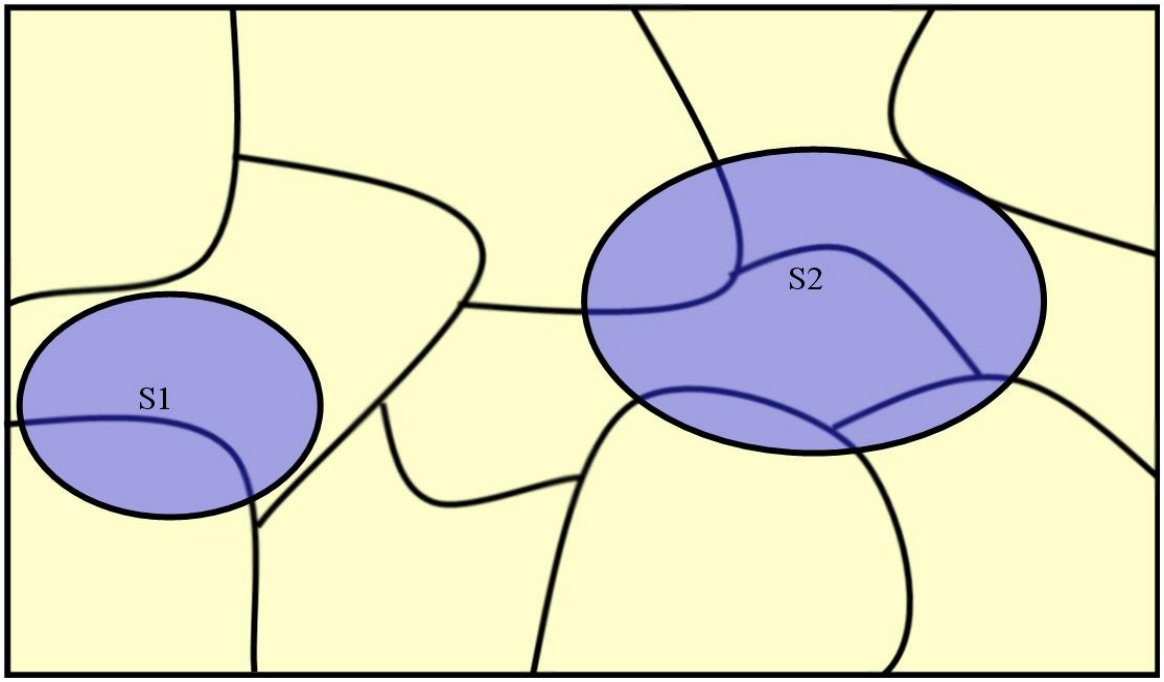

Figure 3.4: Venn diagram of original similarity sets

Each set $\mathcal{S}_i$ is manually created by associating characteristic names from one or more equivalent classes from $\mathcal{V}$ with a name from a class $\mathcal{V}_j$ (note that this will probably not

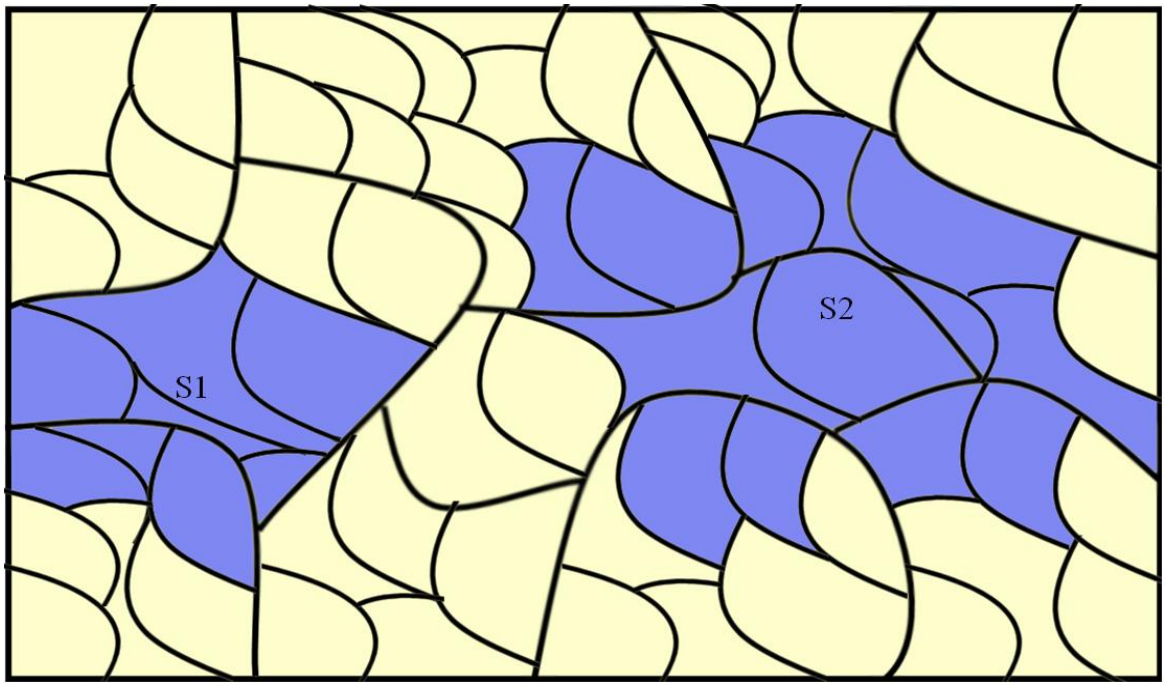

Figure 3.5: Venn diagram of alternative similarity sets

be the characteristic name from $\mathcal{V}_j$). The function $E : \mathcal{N} \rightarrow 2^{\mathcal{C}}$ is defined as:

$$\begin{aligned} E(x) &= \{C(y) : y \in \mathcal{S}_x\} \text{ for } x \in \mathcal{V} \\ &= E(y) \text{ if } x \notin \mathcal{V} \text{ where } x \preceq y, y \in \mathcal{V} \text{ and there is no other} \\ &\quad z \in \mathcal{V} \text{ such that } x \preceq z \preceq y \end{aligned} \tag{3.6}$$

It appears at first glance that by using smaller equivalence classes, the effectiveness of searches will be compromised. However, by selecting appropriate similarity sets, the same results as larger equivalence groups can be achieved. For example, consider the large equivalence class containing the names *Nella*, *Pieternella*, *Petronella*, *Petro* and *Petru* depicted in figure 3.1. Now consider figure 3.3. Here the class is broken into three smaller classes. By creating a similarity set for each of the names in the large equivalent group, containing the characteristic names of all the small equivalent groups, entries will be made in the index resulting in the same hits as for the large equivalent classes (see figure 3.6). There are thus no adverse affects to using smaller equivalence groups.

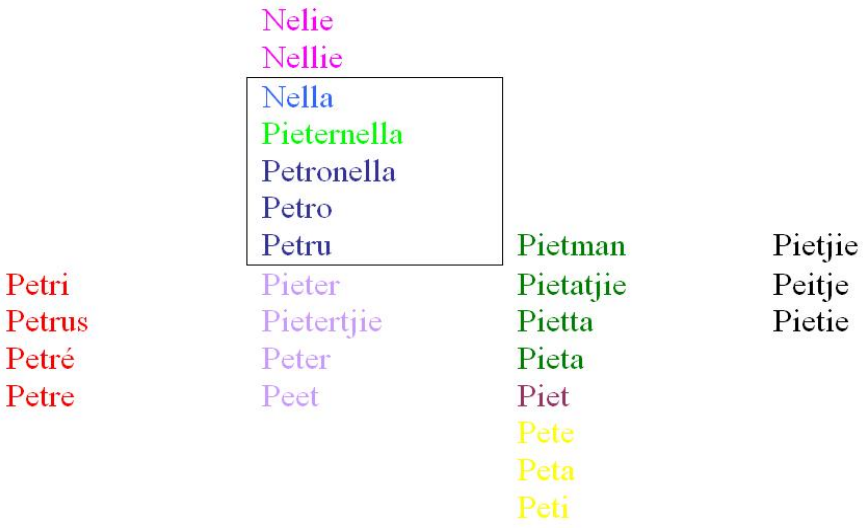

Figure 3.6: Forming of original equivalence class for Pieternella

By making use of smaller equivalent classes and creating similarity sets as defined above, similarity sets closely matching those depicted in figure 3.2 can be created. Figure 3.7 depicts a similarity set that was created by associating several equivalence classes to the name *Pieternalla*. This set is almost identical to one of the sets depicted in 3.2.

The alternative formulation of similarity sets provides a powerful advantage over the original, in that similarity sets need not be symmetric. For example, it is possible to associate *Pieternella* with *Pieter* (i.e. searches for *Pieternella* will also yield records containing *Pieter*) but not associate *Pieter* with *Pieternella*.

For each full name $n = n_1, n_2, ..., n_k \in \mathcal{F}$, entries are made into the index with the keys $E(n_1) \times E(n_2) \times ... \times E(n_k)$. Keys are padded with blanks in the same way as described earlier. The index is significantly larger than when using only an equivalent index, since several entries will be made per full name.

The query full name will be converted to a key in the same way as in section 3.1.2. It is not necessary to convert queries into similarity keys as well, since, if a full name contained in the database has a similarity name equivalent to that of the query, a similarity entry

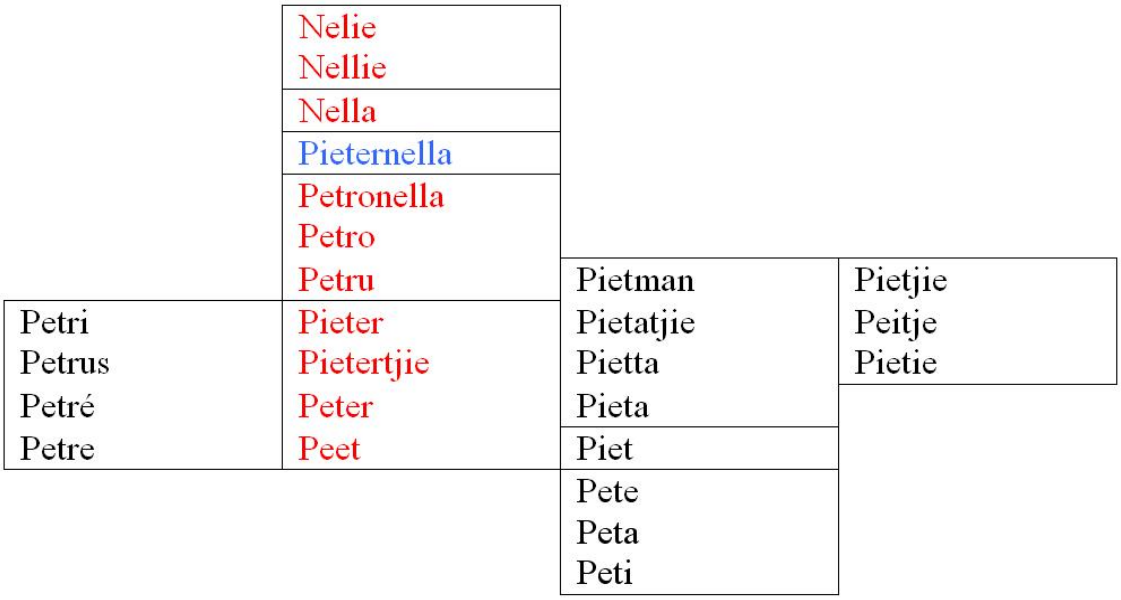

Figure 3.7: Alternative Similarity Set

pointing to the relevant record would already have been made in the index.

A drawback to making entries for each of the similar names in the index is that, for similar names that are lexicographically close, certain blocks of the search results are repeated [dT03]. For example, consider the entries for individuals named *Stephan Francios du Toit* in the database. Assuming the database contains three such individuals, born in 1880, 1920 and 1950 respectively, and ignoring similar names, the index entries would look as follows:

```
.
.
.
Toit______Stephan___Francios__1880
Toit______Stephan___Francios__1920
Toit______Stephan___Francios__1950
.
.
.
```

If search results are presented to the user in lexicographic order, the next results after the three entries will be full names of different individuals, one of which may be the individual

that the user is searching for.

Now consider the situation where the names *Francios* and *Francis* are defined as similar to the name *Frans*. The index entries will now be:

```
.
.
.
Toit______Stephan___Francis__1880
Toit______Stephan___Francis__1920
Toit______Stephan___Francis__1950
Toit______Stephan___Francios__1880
Toit______Stephan___Francios__1920
Toit______Stephan___Francios__1950
Toit______Stephan___Frans_____1880
Toit______Stephan___Frans_____1920
Toit______Stephan___Frans_____1950
.
.
.
```

Certain entries pointing to the a group of individuals are thus repeated. If each name in a full name have similar names associated with it, and if the similar names are all lexicographically close, then the same group of results may be presented to the user many times before an entry in the index is found that represents a new individual.

3.2 General Search Algorithm

In this section a General Search Algorithm (GSA), proposed by De Kock [DK04], making use of not only names but all words (index terms) in the database as well as the structure of the information, is described. A major improvement of the algorithm discussed in this section over those of the previous sections is that it provides a ranked set of possible relevant records in response to a query.

3.2.1 Similarity index on search words

Section 3.1.4 focused on a similarity index for names. This idea can be generalized by now defining $\mathcal{V}$ to be the set of all terms (search words and phrases) contained in the database.

Again equivalent classes and similarity sets are created such that equations 3.3 and 3.6 hold.

Possible search terms are classified into different types, for example, surname, male name, female name, place name, occupation type, etc. Let $T = \{t_1, t_2, ..., t_p\}$ be the set of all possible types. This information can be used for validity checks (for example, gender checking).

For each term, w, the following information is stored: The frequency with which it occurs as a specific type, i.e. $f_1, f_2, ..., f_p$ and a number of pairs $\{(h_i, \sigma_i) : \forall h_i \in E(w)\}$, where σ_i with $0 \leq \sigma_i \leq 1$ indicates the similarity of w to each element of $E(w)$. For the equivalent term in $E(w)$, $\sigma_1 = 1$.

3.2.2 Events

Let the set of event types, $E = \{E_1, E_2, ..., E_q\}$ be the events described in section 1.2 on page 1. We number the subfields, period (date or period, i.e. start and end dates), place and details; 1, 2 and 3 respectively. Let $E_j.r$ refer to subfield r of event E_j.

For the sake of simplicity, we consider the *surname* and *first names* fields to be events with no place or dates associated with it. *Nicknames* is considered to be part of the details field of a first names event. The *general information* field will be considered to be an information event with no dates. Relationships classify as relationship events, with the relationship date as the start date, the separation date as end date and the names of the spouse in the details field.

The correlation between any two event fields must be manually defined in a table that will give the relevance between any two fields. The correlation table can be used to define a relevance function $R : (E \times \{1, 2, 3\}) \times (E \times \{1, 2, 3\}) \longrightarrow [0, 1]$ to indicate relevance between fields. Note that $R(E_i.r, E_i.r) = 1$.

Examples of values between events would be:

- $R(birth.r, baptism.r) = 0.9$ i.e. the same fields in *birth* and *baptism* events are very relevant to each other, since these events normally occur close to each other and are likely to occur in the same place.
- $R(birth.1, baptism.2) = 0$ i.e. there is no relevance between a place name and a date.
- $R(birth.2, baptism.3) = 0.7$ and $R(birth.2, birth.3) = 0.8$ i.e. if a term is expected in a place field, but occurs in a details field, it is still considered relevant.
- $R(birth.r, death.r) = 0.2$ i.e. terms expected to be in a *birth* event, but that occur in a *death* event, are not very relevant since the likelihood that a person was born and died in the same place and that these two events occurred very close together is low (This assumption does not hold for children who died shortly after birth. However, in general, individuals mostly appear in the database if they are part of a family line, hence children who died shortly after birth occur infrequently).

3.2.3 Similarity in dates

In this section a similarity function between dates is defined. Any date can be represented as a real number, where the integer part represents the year and the decimal part represents the months and days. A period, $p = (s, e)$ consists of a start date, s, and an end date, e. If $e = 0$ then p represents a single date. If $s = 0$ it implies that $e = 0$ and p then represents an unknown period. The distance between two periods $p_1 = (s_1, e_1)$ and $p_2 = (s_2, e_2)$, $d(p_1, p_2)$ is given by:

$$d(p_1, p_2) = \begin{cases} 100 \text{ either } s_1 = 0 \text{ or } s_2 = 0 \\ 0 \ e_1 \neq 0, \ e_2 = 0 \ \& \ s_2 \in [s_1, e_1] \\ 0 \ e_2 \neq 0, \ e_1 = 0 \ \& \ s_1 \in [s_2, e_2] \\ 0 \ e_1, e_2 \neq 0 \text{ and} \\ \qquad [s_1, e_1] \cap [s_2, e_2] \neq \phi \\ g(p_1, p_2) \text{ otherwise} \end{cases}$$

where

$$g(p_1, p_2) = \min(\mid e_1 - s_2 \mid, \mid e_2 - s_1 \mid)$$

The similarity $0 \leq f(d(p_1, p_2), a_j) \leq 1$ between two periods p_1 and p_2 is calculated as follows:

$$f(x,a) = \begin{cases} 0 & x < -a \\ \frac{(x+a)(a-x)}{a^2} & -a \leq x \leq a \quad \text{or} \\ 0 & x > a \end{cases} \tag{3.7}$$

$$f(x,a) = e^{-4a^{-2}x^2} \tag{3.8}$$

The two alternative formulas described above are depicted in figure 3.8 for $a = 10$.

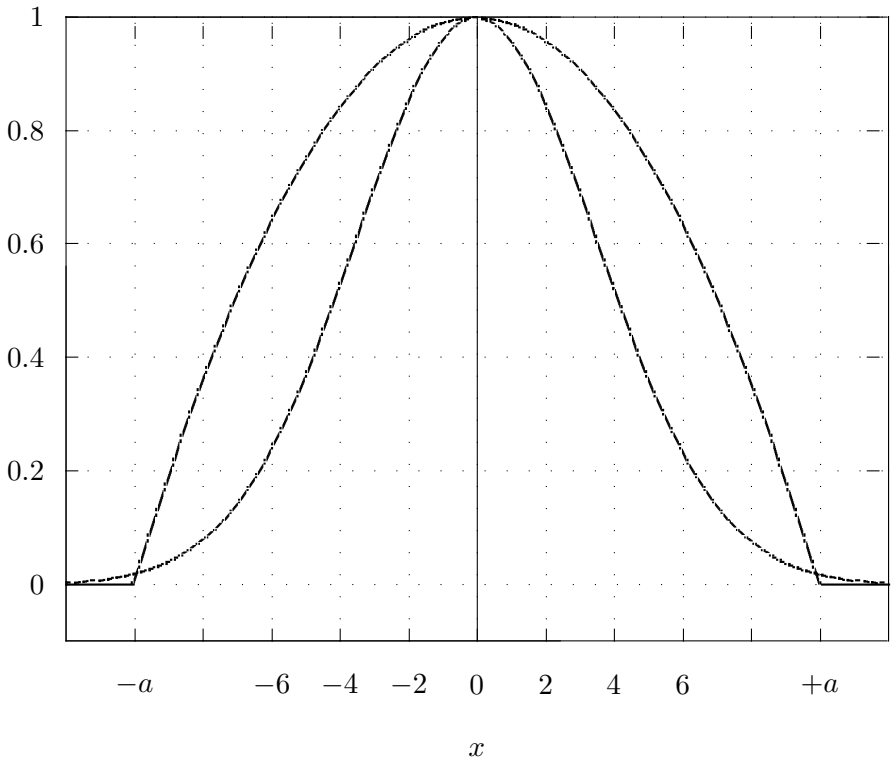

Figure 3.8: $y = f(x, a)$ with $a = 10$

A separate value for a, a_j is defined for each type of event. For events where the date can be expected to be a specific date, for example birth, baptism or death, a_j can be given a low value, for example, $a_j = 10$. For events were the date is less specific, for example, residence or occupation, a_j can be given a high value, for example, $a_j = 20$.

3.2.4 Search index and ranking

A general search algorithm will now be defined using the concepts described in the previous sections.

An entry is made in the search index for each element in the set $\{(h_i, \sigma_i) : \forall\ h_i \in E(w)\}$ of term w in subfield r_j of event E_j of record D. Each entry contains (h_i, σ_i) and a pointer, $(PD, PE_j.r_j)$, pointing to subfield $E_j.r_j$ of record D. This is done for all terms in the database, except dates, so $r_j \in \{2, 3\}$.

A query, Q, consists of a set of tuples

$$Q = \{(v_1, E_{k_1}, t_{k_1}, p_{k_1}), (v_2, E_{k_2}, t_{k_2}, p_{k_2}), ..., (v_L, E_{k_L}, t_{k_L}, p_{k_L})\}$$

with $v_\ell \in \mathcal{V}$, p_{k_ℓ} a period and t_{k_ℓ} a subfield of event E_{k_ℓ}.

The search index is used as follows: For each v_ℓ with $\ell \in L$, $C(v_\ell)$ (see section 3.1.2) is located in the search index. Assume that a set of hits is obtained:

$$H_\ell = \{(C(v_\ell), \sigma_i), (PD_i, PE_{j_i}.r_{j_i})\ :\ i = 1, 2, ..., h\}$$

Assume the set of all possible records to be returned by the search algorithm to be:

$$\{D_1, D_2, \ldots, D_p\} \ = \ \bigcup_{\ell=1}^{L} \bigcup_{i=1}^{h} D_i$$

A ranking number $N(D_k)$ can be calculated for each record D_k for $k = 1, 2, ..., p$, as follows: Let record D_k have events E_i for $i = 1, 2, ..., e_k$ and let $n_{ri\ell}$ indicate the number of occurrences of term v_ℓ in subfield $r \in \{2, 3\}$ of event E_i, then

$$\begin{aligned} N(D_k) \ = \ \sum_{\ell=1}^{L} \sum_{i=1}^{e_k} \sigma_i \Big[& R(E_{k_\ell}.t_\ell, E_i.2) n_{2i\ell} w_{j_i k_\ell}^{t_\ell 2} \\ & + R(E_{k_\ell}.t_\ell, E_i.3) n_{3i\ell} w_{ik_\ell}^{t_\ell 3} \\ & + w_{ik_\ell}^{1} f(d(p_{k_\ell}, p_{j_i}), a_{k_\ell}) \Big] \end{aligned}$$

where $w_{j_i k_\ell}^{t_\ell 2}$ with $r, t \in \{2, 3\}$ is a weight indicating the importance of a hit of the search term in field $E_{j_i}.r_{j_i}$ and field $E_{k_\ell}.t_{k_\ell}$. For example, the weights for a hit of a place name in similar events will be high. If the number of hits of term v_ℓ in an event is deemed irrelevant, $n_{ri\ell}$ can be set to 0 or 1 to indicate a hit or not.

3.3 Conclusions

The General Search Algorithm provides most of the functionality that is required for the GIS. It provides a realistic means of dealing with similarities in terms. To a large extent, the structure of queries and records are taken into account. Furthermore, searches on dates are made possible.

A significant drawback of the GSA is that it does not allow the boolean formulation of queries. Another drawback is that the main term index would contain almost 3500000 entries (the total number of words in the GDB; see chapter 6 for further information), which would make searches slow.

Chapter 4

INFORMATION RETRIEVAL MODEL FOR THE GIS

4.1 Introduction

The aim of this chapter is to motivate the selection of a proposed and implemented IR Model for the GIS.

Of the retrieval models described in chapter 2, the most natural option is the Extended Boolean Model. This model is unique in that it provides the power and functionality of both the Boolean model and the Vector Space model.

The semi structured nature of the information in the GDB makes functionality as is provided by the Tree Matching model essential. Unfortunately, the information in the GIS is structured in recursive hierarchies. This implies that ordered tree inclusion algorithms have polynomial time efficiency. By placing restrictions on queries, it will be shown that the Tree Matching model can be very efficiently implemented and furthermore integrated with the Extended Boolean model.

The General Search Algorithm discussed in chapter 3 takes into account spelling variations and incorporates searches on dates, but does not allow for boolean formulation of

queries. It will be shown how the advantages of the General Search Algorithm can be incorporated into the Extended Boolean Model with searches based on structure.

4.2 Adaptation of Extended Boolean Model

The ranking formula of the Extended Boolean model in its general form is given by:

$$sim(d_j, q) = \sqrt[p]{\frac{w_{1,q}^p.(w_{1,j})^p + ... + w_{t,q}^p.(w_{t,j})^p}{w_{1,q}^p + ... + w_{t,q}^p}} \quad \text{for } q = [(w_{1,q}, k_1) \vee^p ... \vee^p (w_{t,q}, k_t)] \tag{4.1}$$

$$sim(d_j, q) = 1 - \sqrt[p]{\frac{w_{1,q}^p.(1 - w_{1,j})^p + ... + w_{t,q}^p.(1 - w_{t,j})^p}{w_{1,q}^p + ... + w_{t,q}^p}} \quad \text{for } q = [(w_{1,q}, k_1) \wedge^p ... \wedge^p (w_{t,q}, k_t)] \tag{4.2}$$

By following the recommendations of Salton et al [SB88], described in section 2.4.1 on page 11, an appropriate query term weight would be **nfx**, i.e.:

$$w_{i,q} = \log \frac{N}{f_i} \tag{4.3}$$

since short query vectors can be expected.

An appropriate record term weight would be **tfc**:

$$\frac{tf. \log \frac{N}{f}}{\sqrt{\sum_{vector} \left(tf_i. \log \frac{N}{f_i} \right)^2}}$$

because of the varied vocabulary and the fact that record vectors are comparatively short and are of homogeneous length. It was decided not to make use of the normalization term, **c**, for the following reasons:

1. Including the term implies an extra pass of all term vectors after the **tf** components of the weights have been calculated. Apart from the computing time involved in this process, the memory requirements would be significant, especially for long query vectors.

2. The following sections will describe how the term weight will also be used as a measure of how well the structure of a target record matches the structure specified by the user in the query. Normalization with respect to other terms in the query would be pointless since we are interested in how well the record matches the structure with respect to other records.

For the same reason as point 2 above, it was decided to divide each term weight by the maximum weight found for that term in all records. The weight of index term k_i in record d_j is thus:

$$w_{i,j} = \frac{F_{i,j}.\log\frac{N}{f_i}}{\max_p\left(F_{i,p}.\log\frac{N}{f_i}\right)} \tag{4.4}$$

where $F_{i,j}$ is the frequency with which index term k_i occurs in record d_j, N is the total number of records and f_i is the number of records that contain term k_i.

Equations 4.3 and 4.4 are standard weighting techniques and should perform well in any archival database. The significance of the scale factor in equation 4.4 will only become clear after the following sections. This is a different approach, but should not negatively affect the final ranking since terms will still be ranked in the same order relative to each other.

4.3 Adaptation of Structured Text Retrieval

The motivation behind the use of structured text retrieval models, discussed in section 2.8, is that queries specify where in a record a certain term must be found. The data in the GIS is far more structured than information in a normal text document. Often the expected location of a term in a record is known, for example, it may be known that a person was born in a town called *Tarkastad*. A possible drawback to taking a strict structured view of the information is that some terms may not appear in the exact field where they were expected. For example, a search that attempts to locate a person with the first name *Mathys* may not return a record where *Mathys* appears in the nickname field. It is thus necessary to take into account the hierarchical nature of the information in the GIS.

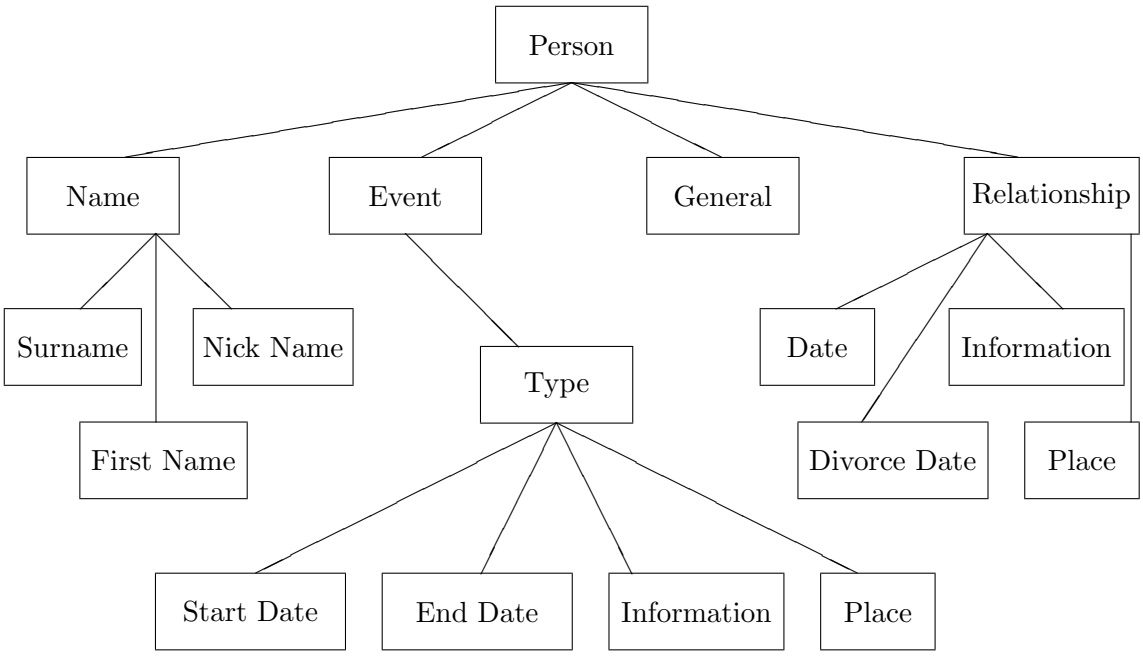

Figure 4.1: Structure of a person record

Consider figure 4.1. Here the information of one person record is structured in a tree. Queries specify the leaf or subtree where a term is expected. Consider the following 3 queries (See appendix B on page 128 for the final EBNF of queries.):

```
Person(Mathys) AND Person(Tarkastad)
First Name(Mathys) AND Birthplace(Tarkastad)
Name(Mathys) AND Event(Tarkastad)
```

The first query is looking for the terms *Mathys* and *Tarkastad* anywhere in a person record. The second query is looking for the term *Mathys* specifically in a first name field and the term *Tarkastad* specifically in a birth place field. The third query only requires the term *Mathys* to be in a name subtree (first name, surname or nickname) and the term

Tarkastad to be in an event subtree.

Note that it was not necessary to allow for recursive type queries, for example:

```
Person(Birth(Place(Tarkastad)), Name(Mathys))
```

for a single person. This makes it possible that the Tree Matching algorithms discussed in section 2.8 to be implemented efficiently.

Although tree matching would be a large improvement on the searching capability of the system, a broader approach is necessary for the GIS. Consider the following query:

```
Birthplace(Tarkastad)
```

If a pure Tree Matching algorithm was employed, only records that contain the term *Tarkastad* in a birth place field would be returned. Records where the term appeared in the baptism place field would not be returned. There is a strong relationship between these two fields. If a child was baptized in a certain town, it is very likely that that child was also born there. Therefore records containing term *Tarkastad* in the baptism field should also be returned to the user. The user could change the query to:

```
Eventplace(Tarkastad)
```

but then all records that have the term *Tarkastad* associated with the place field in an event would be returned.

Let $H = \{H_1, H_2, ..., H_n\}$ be the set of all fields (for example, Surname) and field hierarchies (for example, Name) within a person record. Let $H_L = \{H_{L_1}, H_{L_2}, ..., H_{L_n}\}$ be the set of all leaf fields in the tree hierarchy and let $H_N = \{H_{N_1}, H_{N_2}, ..., H_{N_n}\}$ be the set of all internal nodes (with their respective subtrees) in the tree hierarchy. Thus $H = H_L \cup H_N$.

If $H_j \in H$ is a node in the subtree $H_i \in H_N$ then we denote it with $H_j \preceq H_i$. We define a relevance function $\mathcal{D} : (H \times H) \longrightarrow [0, 1]$ indicating the relevance between any two fields. Let the first parameter of $\mathcal{D}$ be the field as specified in the query, and the second

parameter be the field where a term was found in a record. Note that $\mathcal{D}(H_i, H_i) = 1$ and that $\mathcal{D}(H_i, H_j) = 1$ if $H_j \preceq H_i$.

$\mathcal{D}$ is not symmetric, i.e. in general $\mathcal{D}(H_i, H_j) \neq \mathcal{D}(H_j, H_i)$. Obviously this is the case where H_j is a node in the subtree denoted by H_i (i.e. $H_j \preceq H_i$). If the user specified that a term should fall within a name field and the term was found in the surname field, $\mathcal{D}$ should return a much higher relevance value than if the user specified a term should be found in the surname field but it was found in some other name field. For situations where neither $H_j \preceq H_i$ or $H_j \succeq H_i$, $\mathcal{D}$ is also not symmetric. For example, if the user specified in a query that a term should be found in a residence place field, and the term is then found in a baptism place field, it is likely that that person lived in that place in his or her early years. $\mathcal{D}$ should thus return a reasonably high value. Conversely, if the user specified that a term should be located in a birth place field, but the term was found in a residence field, it does not really imply that that person was born there. $\mathcal{D}$ should thus return a reasonably low value.

$\mathcal{D}$ makes use of a manually drawn up relevance table, from which the relevance of any two leaf fields can be read.

Often the only information known about an individual is details of that person's parents, spouses, relationships or children. It is thus very useful to allow queries as follows:

```
Person(Mathys) AND Mother(Tarkastad)
Father(Firstname(Mathys)) AND Mother(Birthplace(Tarkastad))
Child(Name(Mathys)) AND Spouse(Event(Tarkastad))
```

For the sake of efficiency it is necessary to restrict the user to queries one generation from the person to be searched for, i.e. not allow recursive generational queries, like:

```
Mother(Father(Surname(Mathys)))
```

For the same reason we define $\mathcal{D}(H_i, H_j) = 0$ if H_i and H_j are not from the same person, for example, if a term to be found in a field in a spouse record is found in a field in a child

record then there is no relevance between the query and the record.

4.4 Incorporation of Structured Text Retrieval

Two approaches to incorporating structured text retrieval into equations 4.1 and 4.2 will be discussed here. The first approach was designed specifically with an efficient implementation in mind. This approach will be referred to as the **Fast Matching** approach. A second approach was devised through communications with the study leader. The second approach, though it may not be as fast as the first, should yield better retrieval results. The second approach will be referred to as the **Complete Matching** approach. The reasons for the first approach being faster than the second will be discussed in chapter 7.

The major difference between the two ranking approaches is how multiple occurrences of terms in the same record are dealt with. Records in which a specific term only appears once, will be ranked exactly the same by the Fast and the Complete matching algorithms.

4.4.1 Fast Matching

This approach assumes that the ranking depends only on the best structure match found for the term in the record. To incorporate structured text retrieval into equations 4.1 and 4.2 we simply multiply the record term weight (given in equation 4.4) by the relevance function:

$$w_{i,j} = \frac{F_{i,j}.\log\frac{N}{f_i}.\mathcal{D}(H_{i,q},H_{i,j})}{\max_p\left(F_{i,p}.\log\frac{N}{f_i}.\mathcal{D}(H_{i,q},H_{i,p})\right)} \tag{4.5}$$

where $H_{i,q}$ is the query specified location of term k_i and $H_{i,j}$ is field where term k_i was found in record d_j.

4.4.2 Complete Matching

The problem with equation 4.5 is that, for situations where a term occurs several times in a record but few of those times in the desired field, the term will be assigned a weight as

if all its occurrences were in the correct field. A more accurate approach would be to look at the value given by the distance function for each time the term appears in a different field. Equation 4.4 is adapted by replacing the term frequency component, $F_{i,j}$, by the sum of the distance between the field specified in the query and each occurrence in the record:

$$w_{i,j} = \frac{\left(\sum_{(H_k \in H_L) \preceq H_{i,q}} \sum_{(H_l \in H_L) \preceq H_{i,j}} \mathcal{D}(H_k, H_l) \right) .\log \frac{N}{f_i}}{\max_p \left[\left(\sum_{(H_k \in H_L) \preceq H_{i,q}} \sum_{(H_l \in H_L) \preceq H_{i,p}} \mathcal{D}(H_k, H_l) \right) .\log \frac{N}{f_i} \right]} \tag{4.6}$$

In effect, the Complete Matching approach assigns a weight according to how the multiple occurring term is distributed in the record with respect to the query.

The role of the denominator in equation 4.5 and 4.6 can now be better explained. Consider a situation where a query consists of several terms. Assume the record d_j contains only one of the terms, but it occurs several times and in the desired location. If we were to normalize the weight assigned to that term in d_j with respect to the Euclidian length of the weight vector for d_j then record d_j would be ranked too high (i.e. it may outrank records that contain d_j the same amount of times and also in the correct field, merely because the other records contain some of the other terms as well). The denominator that was finally decided on is an attempt to ensure that no single term ever dominates the ranking of a multiple term query.

4.5 Combination with General Search Algorithm

The algorithm discussed in this chapter provides all the functionality of the General Search Algorithm discussed in chapter 3, except for accounting for equivalent and similar terms and dates. In the following section it will be shown how the ideas presented in chapter 3 can be introduced into the information retrieval algorithms for the GIS.

4.5.1 Similarity Sets

The General Search Algorithm makes use of similarity sets to circumvent the problem of spelling variations in terms. This approach can easily be incorporated into our algorithm

so far by first finding $E(k_i)$, the set of similarity names for term k_i, and then calculating $w_{i,j}$. For the Fast Matching approach (equation 4.5) the final weight will be given by:

$$w_{i,j} = \frac{\sum_{k_m \in E(k_i)} F_{m,j}.\log\frac{N}{f_i}.\max_{k_m \in E(k_i)}(\mathcal{D}(H_{i,q}, H_{m,j}))}{\max_p\left(\sum_{k_m \in E(k_i)} F_{m,p}.\log\frac{N}{f_i}.\max_{k_m \in E(k_i)}(\mathcal{D}(H_{i,q}, H_{m,p}))\right)} \tag{4.7}$$

For the Complete Matching approach (equation 4.6) the final weight will be given by:

$$w_{i,j} = \frac{\sum_{k_m \in E(k_i)}\left(\sum_{(H_k \in H_L) \preceq H_{m,q}}\sum_{(H_l \in H_L) \preceq H_{m,j}}\mathcal{D}(H_k, H_l)\right).\log\frac{N}{f_i}}{\max_p\left[\sum_{k_m \in E(k_i)}\left(\sum_{(H_k \in H_L) \preceq H_{m,q}}\sum_{(H_l \in H_L) \preceq H_{m,p}}\mathcal{D}(H_k, H_l)\right).\log\frac{N}{f_i}\right]} \tag{4.8}$$

Presumably similar terms will occur in the same record very infrequently.

4.5.2 Dates

Discussion so far focused only on index terms and made no mention of dates. Allowing for the inclusion of dates in queries would be a powerful addition to the search algorithm. The user should be able to enter queries like the following:

```
Firstname(Mathys) AND Birthdate(1980.01.12)
Name(Mathys) AND Birthdate(1980-1981)
```

It is important to note here that searching on a date alone would not be very useful in locating an individual in the GDB. The field matching can be achieved using function $\mathcal{D}$ defined above. The general weight of a date should not be calculated as in equation 4.4, rather, records should be ranked in order of how close their relevant dates are to the query date. An approach similar to that followed by [DK04] (discussed in section 3.2.3 on page 48) will thus be followed. The selected approach will be given first and be motivated below.

Let a date be represented as the number of days that have passed since 1 January 1 A.D.. Define a period p as the tuple (*start date*, *end date*). The distance between two periods $p_1 = (s_1, e_1)$ and $p_2 = (s_2, e_2)$, $d(p_1, p_2)$ is given by:

$$g(p_1, p_2) = \begin{cases} 999999 \text{ either } s_1 = 0 \text{ or } s_2 = 0 \\ \mid s_1 - s_2 \mid e_1 = 0 \text{ and } e_2 = 0 \\ 0.5\ e_1 \neq 0,\ e_2 = 0\ \&\ s_2 \in [s_1, e_1] \\ 0.5\ e_2 \neq 0,\ e_1 = 0\ \&\ s_1 \in [s_2, e_2] \\ 0\ e_1, e_2 \neq 0 \text{ and} [s_1, e_1] \cap [s_2, e_2] \neq \phi \\ \min(\mid e_1 - s_2 \mid, \mid e_2 - s_1 \mid) \text{ otherwise} \end{cases} \tag{4.9}$$

The similarity $0 \leq s(g(p_1, p_2)) \leq 1$ between two periods p_1 and p_2 is calculated as follows:

$$s(x) = e^{-4.(3650)^{-2}x^2}$$

which can be rewritten as:

$$s(x) = e^{x^2/-3330625}$$

It was decided use equation 3.8 rather than equation 3.7 because of the sharper gradient near zero, i.e. for p_1 to have have a high similarity to p_2 it must be closer to to p_1 than it would have to be if equation 3.7 was used. Furthermore 3.8 only approaches zero would thus provide a useful similarity for any two dates (rather than assigning the similarity value to zero for large differences).

Note that, in equation 4.9, a value of 0.5 (or half a day) is assigned to g when one and only one of the end dates is zero. This is to ensure that, when a query only specified a start date, occurrences of single dates that match the query date exactly will rank very slightly higher than records that merely contain a period in which the query date falls.

In the discussion in section 3.2.3, a variable, a, could be changed to give different similarity values for different queries. It was decided to set a to 3650 (about 10 years) for all queries, since the location relevance of a date occurrence can be found with $\mathcal{D}$.

Given a query, q, that contains a period, p_q, we define the weight of a period term, p_{i_j}, in record d_j as:

$$w_{i,j} = s(g(p_q, p_{i_j})).\mathcal{D}(H_{i,q}, H_{i,j}) \tag{4.10}$$

To be consistent with the weights calculated for normal terms, the weight for each period term is divided by the maximum weight found for the period in all records:

$$w_{i,j} = \frac{s(g(p_q, p_{i_j})).\mathcal{D}(H_{i,q}, H_{i,j})}{\max_r \left(s(g(p_q, p_{i_r})).\mathcal{D}(H_{i,q}, H_{i,r})\right)} \tag{4.11}$$

This weight can be used in equations 4.1 and 4.2 in the same way as the weight for normal terms, given by equation 4.7, is used.

4.6 Conclusions

The algorithms presented in this chapter provide essential functionality for a search algorithm for the GIS. These algorithms were designed with efficient implementations in mind, but no implementation details have been discussed. The following chapters gives some background on possible implementation techniques and motivates the decisions made in the final implementation.

Chapter 5

INDEXES

5.1 Introduction

Consider the problem of finding all occurrences of term k_i in a record set $\boldsymbol{D}$. The simplest method would be to do a *sequential search* through all the records d_j in $\boldsymbol{D}$, comparing all the terms in d_j with the term k_i. Although this method guarantees success, it is not practical due to the amount of time such a search takes. For example, consider searching for a word in the GDB. It currently contains about 3500000 words (names of individuals are included in this number). A sequential search would thus imply 3500000 word comparisons. Without using efficient text matching algorithms, such a search takes about 20 minutes on a modern PC. This is clearly not acceptable.

If the search was restricted to only one field in a record, for example the surname of an individual, the problem could be simplified by sorting the records in alphabetical order of surname and then using a binary search to locate the desired individuals. The GDB contains about 600000 records, so a sequential search for a surname would take about 600000 comparisons. Using a binary search on a sorted record set, the search would only need $\lceil \log_2 600001 \rceil = 20$ comparisons plus the number of comparisons it would take to identify how many other records contain an individual with the same surname. Unfortunately, sorting the record set is very expensive, so it is not practical to re-sort the record set on a different field each time a search should be performed.

Searching time can be improved by building efficient data structures over the record set, called indexes. Apart from providing a means of limiting the number of comparisons during a search, indexes also provide a means of limiting the number of disk accesses (which is, by comparison, an extremely time consuming operation) needed to locate information. Although indexes add to the disk space and are relatively expensive to maintain (each time a record is added, edited or deleted, the changes have to be reflected in the indexes), it is worth while for large semi-static collections. The GDB can be described as semi-static since a large percentage of the records will rarely be changed.

Indexes have been the focus of intensive research because of their usefulness. The three most powerful indexing methods will be discussed in this chapter.

5.2 Inverted File

The *inverted file* or *inverted index* draws its name from the fact that it represents a situation where the roles of the records and the roles of the attributes are reversed [Knu73]. Instead of listing the attributes of a given record, we list the records having given attributes. An example of an inverted file in every day life is the glossary or index appearing in the back of most academic books.

An inverted file consists of a list of words (called index terms) in alphabetic order that appear in the lexicon (the list of all terms that can be searched on) [TCB99]. See chapter 6 for a discussion on which terms should be included in the lexicon (for the moment, assume that all terms will be included). Each index term points to an *inverted list*, that contains all the occurrences or *postings* of the term in the record set.

For example, consider a record set where each record is a verse of poetry:

Record 1

```
Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand or eye
Could frame thy fearful symmetry?
```

Record 2

```
In what distant deeps or skies
Burnt the fire of thine eyes?
On what wings dare he aspire?
What the hand dare seize the fire?
```

An inverted index for the record set given above can be seen in table 5.1. Note that each posting is given in the form *Record, Line, Position*; where *Record* is the record number of the record in which the term appears, *Line* is the line number on which the term appears, and *Position* is the position of the term in the line. As is the practice when indexing a record, punctuation has been left out.

The *addressing granularity* of the postings is how accurately the position of a term in a record is specified. This normally depends on the application. Often only the record number is specified. It is also common practise to break the records up into logical blocks, for example blocks of 10 characters each. The posting list would thus require less space because there are fewer blocks than positions and a posting can thus be represented by a smaller number, and more than one occurrence of a term inside a block would now be reduced to a single posting.

Searching for a term using an inverted index can be summed up in three steps [RBY99]:

1. **Lexicon search** Terms in the query are isolated and located in the lexicon.

2. **Retrieval of postings** Lists of the occurrences of all terms found are retrieved.

Index Term	**Postings** (*Record, Line, Position*)
Burnt	(2,2,1)
Could	(1,4,1)
In	(1,2,1) (2,1,1)
On	(2,3,1)
Tyger	(1,1,1) (1,1,2)
What	(1,3,1) (2,4,1)
aspire	(2,3,6)
bright	(1,1,4)
burning	(1,1,3)
dare	(2,3,4) (2,4,4)
deeps	(2,1,4)
distant	(2,1,3)
eye	(1,3,5)
eyes	(2,3,6)
fearful	(1,4,4)
fire	(2,2,3) (2,4,7)
forests	(1,2,3)
frame	(1,4,2)
hand	(1,3,3) (2,4,3)
he	(2,3,5)
immortal	(1,3,2)
night	(1,2,6)
of	(1,2,4) (2,2,4)
or	(1,3,4) (2,1,5)
seize	(2,4,5)
skies	(2,1,6)
symmetry	(1,4,5)
the	(1,2,2) (1,2,5) (2,2,2) (2,4,2) (2,4,6)
thine	(2,2,5)
thy	(1,4,3)
what	(2,1,2) (2,3,2)
wings	(2,3,3)

Table 5.1: An Inverted Index

3. **Manipulation of occurrences** Occurrences are processed to solve query (for example, Boolean) operations.

Inverted files provide a very flexible and efficient means to locate relevant records, and are thus the most frequently used indexing method.

5.3 Suffix Trees and Suffix Arrays

Suffix trees provides an efficient way to search for phrases, words or prefixes of words in a record set [GT02]. A Suffix Array provides the same functionality as a Suffix Tree but requires less space.

When a Suffix tree is used, all the data in the record set is seen as one long string of text. A suffix tree is essentially a *trie* data structure built over all the suffixes of the text. For example, consider the following string of text:

```
In what distant deeps or skies
```

The above text will have the following suffixes:

```
In what distant deeps or skies
what distant deeps or skies
distant deeps or skies
deeps or skies
or skies
skies
```

In this example we will only index words, providing similar functionality as inverted files. A suffix tree is created by first noting the position of each term in the record set (in this case the position of the first character of the term in the line), for example:

1	4	9	17	25	28
In	what	distant	deeps	or	skies

A Suffix Tree can be drawn up for the text as seen in figure 5.1.

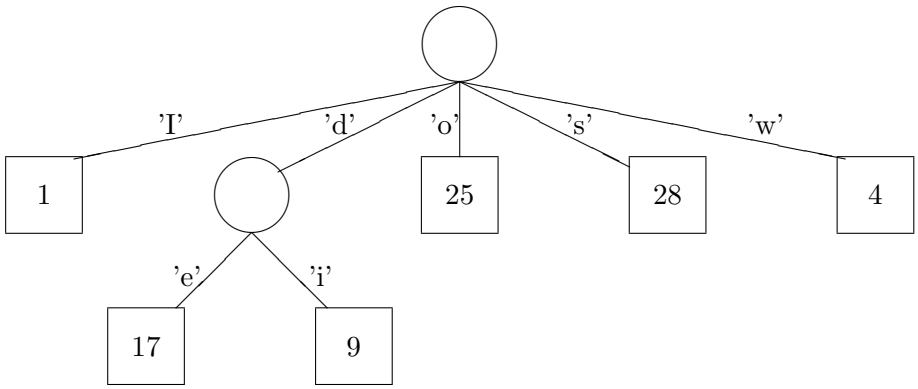

Figure 5.1: A Suffix Tree.

A search for a word (the model can be extended to allow for phrases) can now be performed by using the characters in the search word to trace a path in the tree. Leaves in the suffix tree store the position of the search word in the text.

Suffix Trees allow very efficient searching (ignoring disk accesses), but they are not space efficient. The size of a Suffix Tree can vary between 120% to 240% of the size of the original collection [RBY99].

A similar approach is a Suffix Array. It is created by performing an in-order traverse on a Suffix Tree and storing the leaves in an array. The Suffix Array for the Suffix Tree in figure 5.1 would thus be:

1	17	9	25	28	4

Note that the elements of the array is stored in lexicographical order. A word can be located using a binary search by comparing words at the respective locations with the search word. Unfortunately, in most cases a disk read will have to be performed for each comparison (to locate the word to compare).

By using a *supra-index*, the problem can be reduced. A supra-index is a second index that points to locations in the Suffix Array. A search would first be done in the supra-index to reduce the amount of disk reads. Not all index terms have to appear in the supra-index. To make the supra-index more space efficient, only every x words represented in the Suffix Array need appear. To improve space efficiency still further, only the first y characters of each word need to be taken into account. The supra-index is likely to be small enough to be stored in memory. This means that the closest match to a term is first found in the supra-index, which points to some location in the Suffix Array. The exact match of the term is then searched for from that position in the Suffix Array, hence reducing the number of disk reads.

A Suffix Array provides the same functionality as a Suffix Tree but requires less space.

The distinction between a Suffix Tree (when implemented as a Suffix Array) and a Inverted File is very small, since they both rely on an index to point towards a list of postings. The most significant difference is that the occurrences of a single term in an Inverted File is sorted by text position, while in a Suffix Tree it is sorted in lexicographical order of the words following the term.

5.4 Signature Files

Signature Files work on the principle of assigning a descriptor to each record [TCB99]. The descriptor of a record depends on the terms that appear in the record. In the following example each record contains a line of poetry:

Record 1

```
In what distant deeps or skies
```

Record 2

```
Burnt the fire of thine eyes?
```

Record 3

```
On what wings dare he aspire?
```

Record 4

```
What the hand dare seize the fire?
```

A 16 bit hash code is created for each word contained in the records, as shown in table 5.2.

Index Term	**Hash Code**
Burnt	1000 0000 0010 0100
In	0010 0100 0000 1000
On	0000 1010 0000 0000
What	0000 1001 0010 0000
aspire	0000 1000 1000 0010
dare	0100 0010 0000 0001
deeps	0010 1000 0000 0100
distant	1000 1000 0100 0000
eyes	0000 0101 0000 0001
fire	0100 0100 0010 0000
hand	0000 0010 0110 0000
he	0100 0100 0000 0001
of	1010 1000 0000 0000
or	0010 0001 0000 0010
seize	0001 1000 0000 1000
skies	0000 0100 1000 0100
the	0110 0000 0100 0000
thine	0000 0001 0010 0010
what	0001 1000 0000 1000
wings	0100 0000 1100 0000

Table 5.2: Hash Codes for Index Terms

The record descriptors are created by performing a logical OR on the hash codes of terms contained in each record. Thus, the descriptor for record 1 is the disjunction of 0010 0100 0000 1000 (hash code for *In*), 0001 1000 0000 1000 (hash code for *what*), 1000 1000 0100 0000 (hash code for *distant*), 0010 1000 0000 0100 (hash code for *deeps*), 0010 0001 0000 0010 (hash code for *or*) and 0000 0100 1000 0100 (hash code for *skies*), that is,

1011 1101 1100 1110. The descriptors for all four records can be seen in table 5.3.

Record	Text	Descriptor
1	In what distant deeps or skies	1011 1101 1100 1110
2	Burnt the fire of thine eyes?	1110 1101 0110 0111
3	On what wings dare he aspire?	0101 1110 1100 1011
4	What the hand dare seize the fire?	0111 1110 0110 1001

Table 5.3: Signature file

A search for term k_i can be conducted by finding the hash code term k_i and then doing a sequential search for k_i through all records that have corresponding bits set in their descriptors as the hash code for k_i. Note that the fact that the same bits are set in the descriptor as in the hash code does not guarantee that the term is present in the record. For example, despite that fact that the appropriate bits are set for the term *On* in record 4, the term does not appear in that record.

The drawback of Signature files is that, if records contain many terms, the record descriptors will contain many 1's. This results firstly in too many hits for a search and secondly in inaccurate results.

A great advantage in using Signature files is that phrase searching is improved [RBY99]. This is because by searching for more terms, the number of bits that are set is greater, and thus fewer records would have to be sequentially searched for terms.

Signature files normally take up only 10% to 20% as much space as the original record set. However, because the searching time of a Signature file is linear, Signature files are only appropriate for small text collections.

5.5 Conclusions

Signature files can be dismissed as a possible implementation technique in the GIS simply because of the possibility of inaccurate results. Very specific searches are done on the GDB

and as a result the returning of irrelevant records by a search algorithm should be avoided. Furthermore, the advantage of efficient phrase searching by signature files is redundant for the searches to be supported in the GIS. Signature files are only appropriate for small collections, not for a database the size of the GDB.

The large space requirements of Suffix Trees coupled with the fact that they do not provide any additional functionality makes inverted files the only realistic option.

An inverted file with a B-tree implementation will be used for implementing the search algorithm. A B-tree is a balanced search tree and was created with the aim of minimizing the amount of disk reads necessary to locate an item [FMC98].

Chapter 6

LEXICON REFINEMENT

6.1 Introduction

The list of terms on which a record set is indexed is called the lexicon. Since all the terms in the lexicon appear in the index, it is preferable to keep the lexicon as small as possible. Furthermore, it is also important to remove words from the lexicon unlikely to improve query results. Punctuation (with a few exceptions) is commonly disregarded when setting up the lexicon. In this chapter several techniques used to minimize the size of the lexicon are discussed. An analysis of the distribution of the words in the GIS will be done in section 6.5.

6.2 Case Folding

One of the simplest ways to reduce the number of terms in the lexicon is a technique called *case folding*, whereby all capitalization is ignored. The words **House**, **HOUSE** and **house** will thus all be mapped to **house**. Not only is the number of words in the lexicon reduced, but search recall is improved since more records are retrieved in response to a query. Furthermore, in most cases, search precision is also improved since in the absence of case folding, case mismatch often causes queries to fail [TCB99].

6.3 Stop words

Terms that appear too many times in a record set do not make good search words since too many records are returned by queries including the term. For example, consider the term **van** in the GDB. The term occurs a total of 85 221 times in the GIS, making the term useless as a search term. Furthermore, if an inverted index is used to index the GDB, the term **van** would account for 4.41% of the postings in the index. Clearly search effectiveness can be improved and disk space saved by removing certain terms from the lexicon. Terms that are not indexed are added to a *stop list*.

The choice of terms to add to the stop list has been the subject of debate, but normally consists of articles, prepositions and conjunctions [RBY99]. Other candidates for inclusion in the stop list are terms that appear too frequently (since too many records would be returned), terms that appear very infrequently (words that are likely to be spelling mistakes) and very short terms (for example, one letter terms).

When indexing general text collections, the exclusion of stop words may be problematic since searches for phrases that consists entirely of stop words, for example the verb **to be**, are not entirely unlikely. The GIS especially lends itself to the removal of stop words since a search for an individual is in its nature specific, and thus unlikely to consist entirely of terms that have been designated as stop words.

6.4 Stemming

Stemming is a technique whereby words are broken down to their stems. Thus, terms like **connected**, **connecting**, **connection** and **connections** would all be broken down to **connect**. Queries are also broken down to their stems. Thus a search for the term **connected** would also return records containing, for example, **connection**. The main motivation behind stemming is that, if several terms can be mapped to one term, the size of the lexicon can be significantly reduced.

Although it seems obvious that stemming should improve searches, it has been found in practice that there are no clear benefits to stemming [Fra92].

Several stemming techniques exists, the most prominent of which is affix stemming. Mostly only suffixes of words are removed since most variants of words are generated by adding suffixes to the stem word. It is important to note that the stems that are found by removing affixes do not necessarily have to be a valid word, since the stems are only used for indexing. Unfortunately ambiguity sometimes arise since unrelated words frequently share a common stem.

The most popular suffix removal algorithm is Porter's algorithm ([Por97] in [RBY99]) which reduces words to their stems by a set of production rules to a word. Examples of the rules employed by Porter's algorithm are:

$$sses \rightarrow ss$$
$$ies \rightarrow i$$
$$s \rightarrow \lambda$$

where λ represents the empty string.

It is not recommended to apply Porter's algorithm blindly to any lexicon. The language use and context of a record set has serious implications regarding the stemming algorithm that should be used [XC98]. The GDB contains mostly Afrikaans words, but also English words. Significant changes would have to be made to the rules of Porter's algorithm if stemming were to be applied in the GIS, since the algorithm was specifically designed for stemming English words.

6.5 GIS Term Frequencies

At the time of this analysis (July 2003), the GDB contained a total of 551440 records. The average number of terms (words and names) per record is 6.88 with a standard deviation of 12.38.

6.5.1 Words

In this section the words in the unstructured data (i.e. Data in the Person Details, Person Footnote, Event Place, Event Details, Event Comment, Relationship Details, Relationship Comment and Information categories) were considered. The words were extracted and sorted in ascending order of number of occurrences. There are currently 82289 distinct words occurring a total of 1931578 times.

All the words can unfortunately not be listed, but the words occurring more than three thousand times each can be seen in Table 6.1. These words make up only 0.12% of the list of distinct words.

The presence of many of the words in table 6.1 can be easily explained. The most common word, **van**, is a word often used in Afrikaans. It also occurs in many surnames, for example, *van der Merwe*, *van Antwerpen* and *van Rensburg*. The one letter word **X** was until recently used by genealogists to indicate a marriage in the information field. The words **J**, **D** and **M** results from entries of the age at death, for example, *80j 9m 18d*. The words **S** and **K** represents references to death notices, *S/K* (from the afrikaans word *sterfkennis*).

The words in the GIS are distributed very unevenly. Table 6.2 was created by grouping words into groups of 1000, in sequence of their number of occurrences. The percentage of the total word occurrences of each group was then tabulated. Note that only 1000 words make up more than 76.7% of the words in the database. The information in table 6.2 can be seen graphically in figure 6.1.

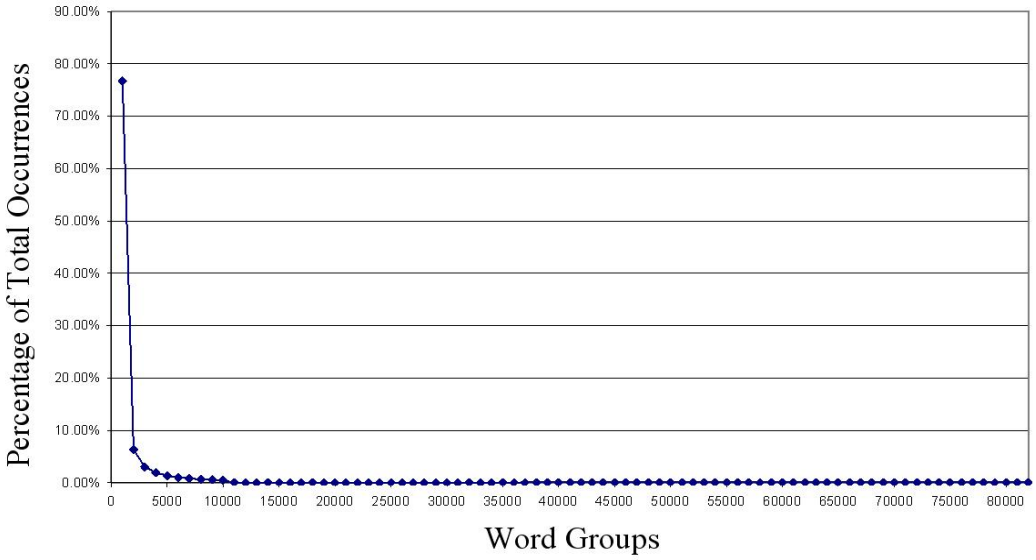

Figure 6.1: Distribution of Word Occurrences in Groups of 1000

Word	# Occurrences	Word	# Occurrences
VAN	85221	DIST	6308
SY	34274	V	6057
EN	30249	KINDERS	6025
DIE	29530	WORD	5948
IN	27902	AAN	5772
MET	24662	PORT	5715
OP	22875	THE	5640
X	22652	A	5436
JR	17697	TEN	5411
BOER	16148	TYE	5263
HY	13990	JOHANNES	5116
KAAPSTAD	13744	SOMERSET-OOS	5090
PRETORIA	13351	G	4960
HAAR	13197	POTCHEFSTROOM	4909
N	11833	OF	4905
J	11741	WORCESTER	4773
TOE	11741	TULBAGH	4662
GRAAFF-REINET	11661	MIDDELBURG	4490
WOON	11582	BEAUFORT-WES	4263
IS	11376	COLESBERG	4236
S	10689	BURGER	4201
AS	10290	ONGETROUD	4151
KERK	9945	PLAAS	4064
DISTRIK	9468	KAAP	3995
DOOD	9443	GERMANY	3990
JOHANNESBURG	9374	AND	3988
SWELLENDAM	8980	HUIS	3928
TROU	8970	JONK	3898
UITENHAGE	8929	ENGLAND	3789
SE	8565	HUMANSDORP	3764
HUWELIK	8550	OORLEDE	3737
PAARL	8451	MARIA	3627
GEORGE	8348	SWANEPOEL	3581
HET	8250	TVL	3402
CALEDON	8247	HOSPITAAL	3392
M	7924	JACOBUS	3344
WAS	7895	CAPE	3298
BY	7688	JOHANNA	3296
ELIZABETH	7543	OOK	3284
GENOEM	7244	VIR	3190
BLOEMFONTEIN	7217	WOONAGTIG	3172
STELLENBOSCH	7126	STERFKENNIS	3133
K	6926	VOOR	3119
DE	6738	ROBERTSON	3098
NIE	6670	KROONSTAD	3081
D	6568	LATER	3080
TE	6556	RUSTENBURG	3051
NA	6477	AANKOMS	3049
CRADOCK	6320	VOLGENS	3016

Table 6.1: Words Occurring more than 3 000 times

Word Group	% of Occurrences	Words	% of Occurrences
1 - 1000	76.70%	41001 - 42000	0.05%
1001 - 2000	6.35%	42001 - 43000	0.05%
2001 - 3000	3.00%	43001 - 44000	0.05%
3001 - 4000	1.92%	44001 - 45000	0.05%
4001 - 5000	1.36%	45001 - 46000	0.05%
5001 - 6000	1.02%	46001 - 47000	0.05%
6001 - 7000	0.81%	47001 - 48000	0.05%
7001 - 8000	0.67%	48001 - 49000	0.05%
8001 - 9000	0.57%	49001 - 50000	0.05%
9001 - 10000	0.49%	50001 - 51000	0.05%
10001 - 11000	0.43%	51001 - 52000	0.05%
11001 - 12000	0.37%	52001 - 53000	0.05%
12001 - 13000	0.33%	53001 - 54000	0.05%
13001 - 14000	0.31%	54001 - 55000	0.05%
14001 - 15000	0.26%	55001 - 56000	0.05%
15001 - 16000	0.26%	56001 - 57000	0.05%
16001 - 17000	0.22%	57001 - 58000	0.05%
17001 - 18000	0.21%	58001 - 59000	0.05%
18001 - 19000	0.21%	59001 - 60000	0.05%
19001 - 20000	0.18%	60001 - 61000	0.05%
20001 - 21000	0.16%	61001 - 62000	0.05%
21001 - 22000	0.16%	62001 - 63000	0.05%
22001 - 23000	0.16%	63001 - 64000	0.05%
23001 - 24000	0.16%	64001 - 65000	0.05%
24001 - 25000	0.00%	65001 - 66000	0.05%
25001 - 26000	0.10%	66001 - 67000	0.05%
26001 - 27000	0.10%	67001 - 68000	0.05%
27001 - 28000	0.10%	68001 - 69000	0.05%
28001 - 29000	0.10%	69001 - 70000	0.05%
29001 - 30000	0.10%	70001 - 71000	0.05%
30001 - 31000	0.10%	71001 - 72000	0.05%
31001 - 32000	0.10%	72001 - 73000	0.05%
32001 - 33000	0.10%	73001 - 74000	0.05%
33001 - 34000	0.10%	74001 - 75000	0.05%
34001 - 35000	0.10%	75001 - 76000	0.05%
35001 - 36000	0.10%	76001 - 77000	0.05%
36001 - 37000	0.07%	77001 - 78000	0.05%
37001 - 38000	0.05%	78001 - 79000	0.05%
38001 - 39000	0.05%	79001 - 80000	0.05%
39001 - 40000	0.05%	80001 - 81000	0.05%
40001 - 41000	0.05%	81001 - 82000	0.05%

Table 6.2: Distribution of Words in Groups of 1000

The word distributions can be clarified by tabulating the data in occurrence ranges (i.e. grouping the words on word occurrences). Table 6.3 shows word occurrence ranges, along with the number of distinct words in the range, the percentage of distinct words represented by the range, the number of occurrences in the database of the words in the range and the percentage of the total number of word occurrences in the database. It can be seen from the more intuitive graphical representation in Figures 6.2 and 6.3 that some of the words that represent a very small percentage of the distinct words, represent a very large percentage of the word occurrences in the GDB (Specifically note that one word represents 4.41% of all the word occurrences in the database).

Word Occur Range	**# Distinct**	**% Distinct**	**# Occur**	**% Occur**
1	45963	55.85%	45963	2.38%
2	11487	13.96%	22974	1.19%
3	5346	6.50%	16038	0.83%
4	3282	3.99%	13128	0.68%
5 - 6	3782	4.60%	20554	1.06%
7 - 11	4219	5.13%	18236	0.94%
12 - 29	4091	4.97%	16340	0.85%
30 - 3000	4026	4.89%	14970	0.77%
3001 - 6025	47	0.06%	194332	10.06%
6026 - 9374	26	0.03%	197921	10.25%
9375 - 17697	17	0.02%	207896	10.76%
17698 - 34274	7	0.01%	192144	9.95%
85221	1	0.00%	85221	4.41%
Total:	82289	100.00%	1931578	100.00%

Table 6.3: Distribution of Words according to Occurrence Range

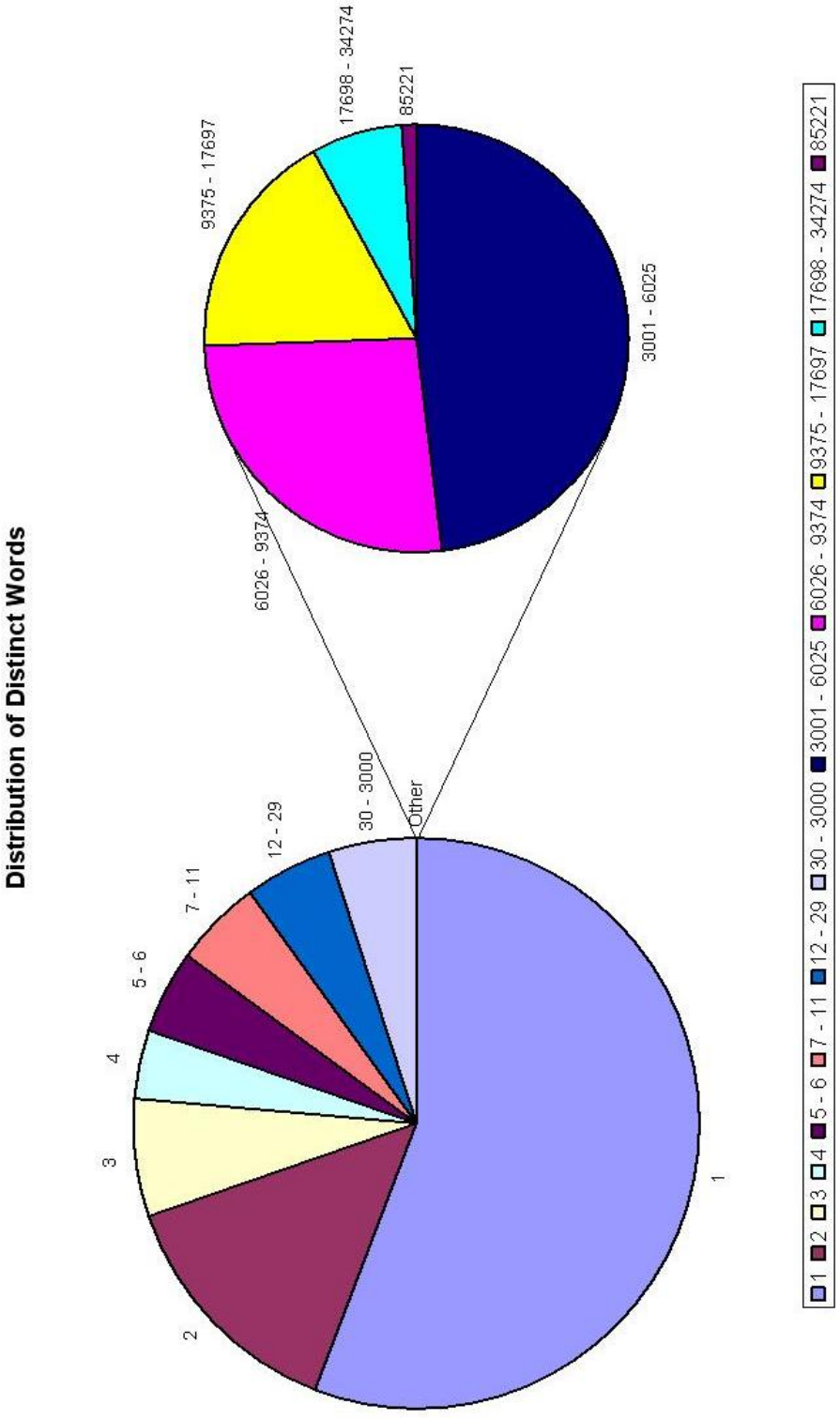

Figure 6.2: Distribution of Distinct words

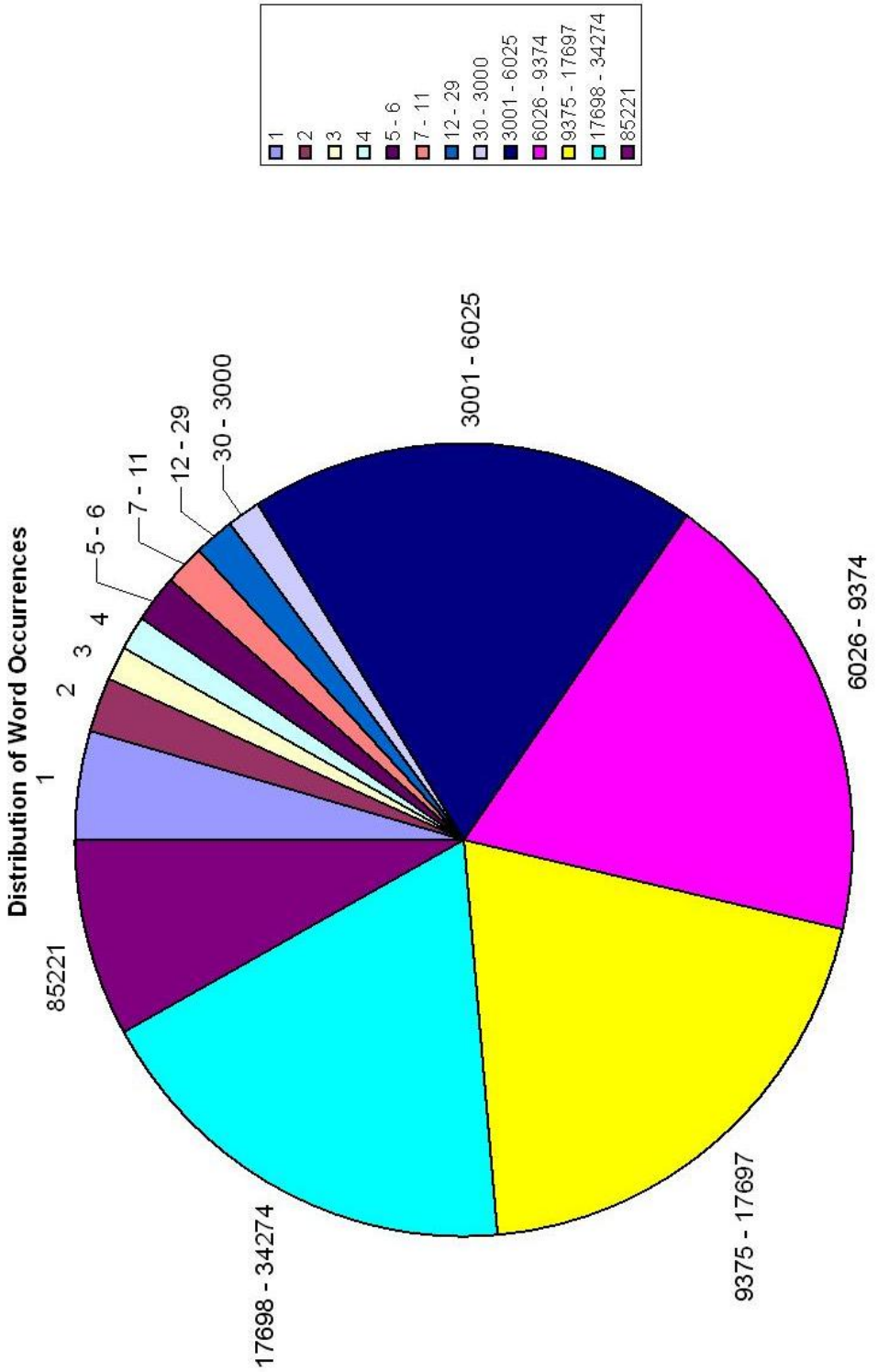

Figure 6.3: Distribution of Word Occurrences

Table 6.4 shows the distribution of words that occur less than 50 times each. Every row in the table lists the number of word occurrences (for example, words occurring 5 times), the number of words that occur the given number of times, the percentage of the total number of distinct words that these words constitute, a cumulative percentage of the previous field, the number of times that these words occur in the database, the percentage of words in the database that are represented by the words in the row and an accumulative percentage of the previous column. These words make up 96.68% of the distinct words and 14.61% of the total word occurrences in the database. A graph of table 6.4 can be seen in figure 6.4.

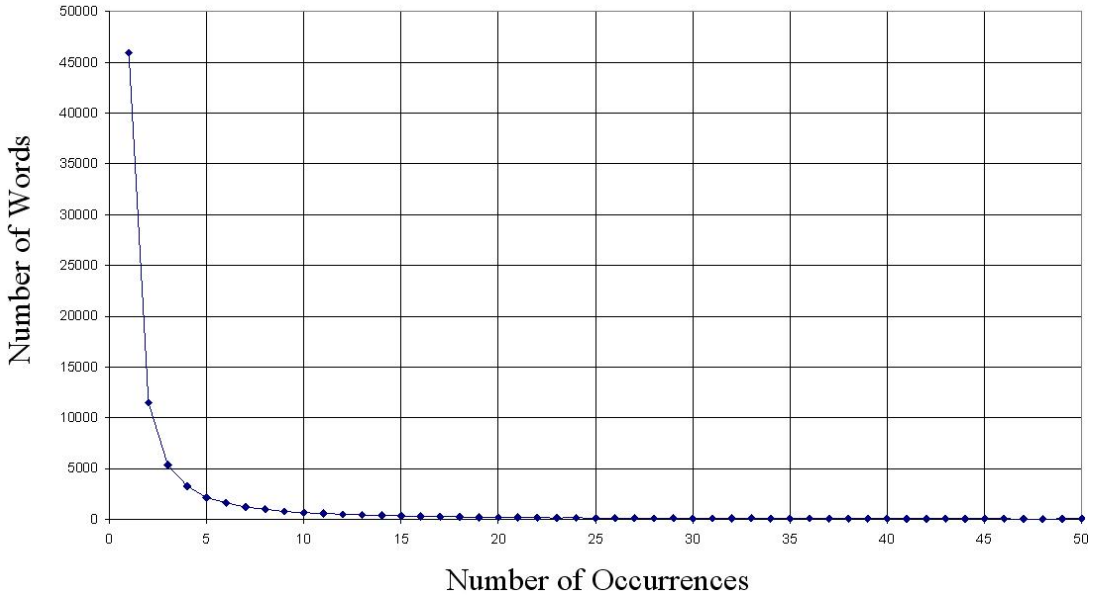

Figure 6.4: Distribution of Words Occurring less than 50 times.

# Occur	# Words	% Distinct	Accu %	Σ # Occur	% Occur	Accu %
1	45963	55.85%	55.85%	45963	2.38%	2.38%
2	11487	13.96%	69.81%	22974	1.19%	3.57%
3	5346	6.50%	76.31%	16038	0.83%	4.40%
4	3282	3.99%	80.30%	13128	0.68%	5.08%
5	2138	2.60%	82.89%	10690	0.55%	5.63%
6	1644	2.00%	84.89%	9864	0.51%	6.14%
7	1196	1.45%	86.34%	8372	0.43%	6.58%
8	996	1.21%	87.55%	7968	0.41%	6.99%
9	778	0.95%	88.50%	7002	0.36%	7.35%
10	651	0.79%	89.29%	6510	0.34%	7.69%
11	598	0.73%	90.02%	6578	0.34%	8.03%
12	489	0.59%	90.61%	5868	0.30%	8.33%
13	440	0.53%	91.15%	5720	0.30%	8.63%
14	391	0.48%	91.62%	5474	0.28%	8.91%
15	354	0.43%	92.05%	5310	0.27%	9.19%
16	278	0.34%	92.39%	4448	0.23%	9.42%
17	262	0.32%	92.71%	4454	0.23%	9.65%
18	251	0.31%	93.01%	4518	0.23%	9.88%
19	217	0.26%	93.28%	4123	0.21%	10.10%
20	196	0.24%	93.51%	3920	0.20%	10.30%
21	175	0.21%	93.73%	3675	0.19%	10.49%
22	164	0.20%	93.93%	3608	0.19%	10.68%
23	147	0.18%	94.11%	3381	0.18%	10.85%
24	155	0.19%	94.29%	3720	0.19%	11.04%
25	124	0.15%	94.44%	3100	0.16%	11.20%
26	110	0.13%	94.58%	2860	0.15%	11.35%
27	129	0.16%	94.73%	3483	0.18%	11.53%
28	100	0.12%	94.86%	2800	0.14%	11.68%
29	109	0.13%	94.99%	3161	0.16%	11.84%
30	93	0.11%	95.10%	2790	0.14%	11.98%
31	93	0.11%	95.21%	2883	0.15%	12.13%
32	95	0.12%	95.33%	3040	0.16%	12.29%
33	101	0.12%	95.45%	3333	0.17%	12.46%
34	80	0.10%	95.55%	2720	0.14%	12.60%
35	67	0.08%	95.63%	2345	0.12%	12.73%
36	86	0.10%	95.74%	3096	0.16%	12.89%
37	63	0.08%	95.81%	2331	0.12%	13.01%
38	58	0.07%	95.88%	2204	0.11%	13.12%
39	76	0.09%	95.98%	2964	0.15%	13.27%
40	72	0.09%	96.06%	2880	0.15%	13.42%
41	49	0.06%	96.12%	2009	0.10%	13.53%
42	61	0.07%	96.20%	2562	0.13%	13.66%
43	69	0.08%	96.28%	2967	0.15%	13.81%
44	46	0.06%	96.34%	2024	0.10%	13.92%
45	47	0.06%	96.39%	2115	0.11%	14.03%
46	53	0.06%	96.46%	2438	0.13%	14.15%
47	44	0.05%	96.51%	2068	0.11%	14.26%
48	41	0.05%	96.56%	1968	0.10%	14.36%
49	51	0.06%	96.62%	2499	0.13%	14.49%
50	45	0.05%	96.68%	2250	0.12%	14.61%

Table 6.4: Distribution of Words Occurring less than 50 times.

The length of the words is another factor that is considered. Table 6.5 shows the distribution of words according to length. Note that 18.3% of the total word occurrences are one or two letter words. A graphical comparison of the percentages of distinct words and word occurrences according to word length can be seen in figure 6.5.

Word Length	**# Distinct**	**% Total Distinct**	**# Occur**	**% Total Occur**
1	26	0.03%	107989	5.59%
2	454	0.55%	245409	12.71%
3	1563	1.90%	277701	14.38%
4	3288	4.00%	206146	10.67%
5	5713	6.94%	114275	5.92%
6	8303	10.09%	151047	7.82%
7	9480	11.52%	194686	10.08%
8	9719	11.81%	149026	7.72%
9	9591	11.66%	162681	8.42%
10	8452	10.27%	110496	5.72%
11	6997	8.50%	69698	3.61%
12	5358	6.51%	69767	3.61%
13	4037	4.91%	35552	1.84%
14	2838	3.45%	12291	0.64%
15	2048	2.49%	8182	0.42%
16	1431	1.74%	9180	0.48%
17	944	1.15%	2771	0.14%
18	629	0.76%	1735	0.09%
19	445	0.54%	1099	0.06%
20	329	0.40%	886	0.05%
21	173	0.21%	307	0.02%
22	145	0.18%	191	0.01%
23	102	0.12%	122	0.01%
24	68	0.08%	81	0.00%
25	51	0.06%	64	0.00%
26	23	0.03%	71	0.00%
27	24	0.03%	31	0.00%
28	21	0.05%	39	0.00%
29	18	0.04%	31	0.00%
30	7	0.01%	9	0.00%
31	12	0.02%	15	0.00%
Total:	82289	100.00%	1931578	100.00%

Table 6.5: Distribution of words according to length

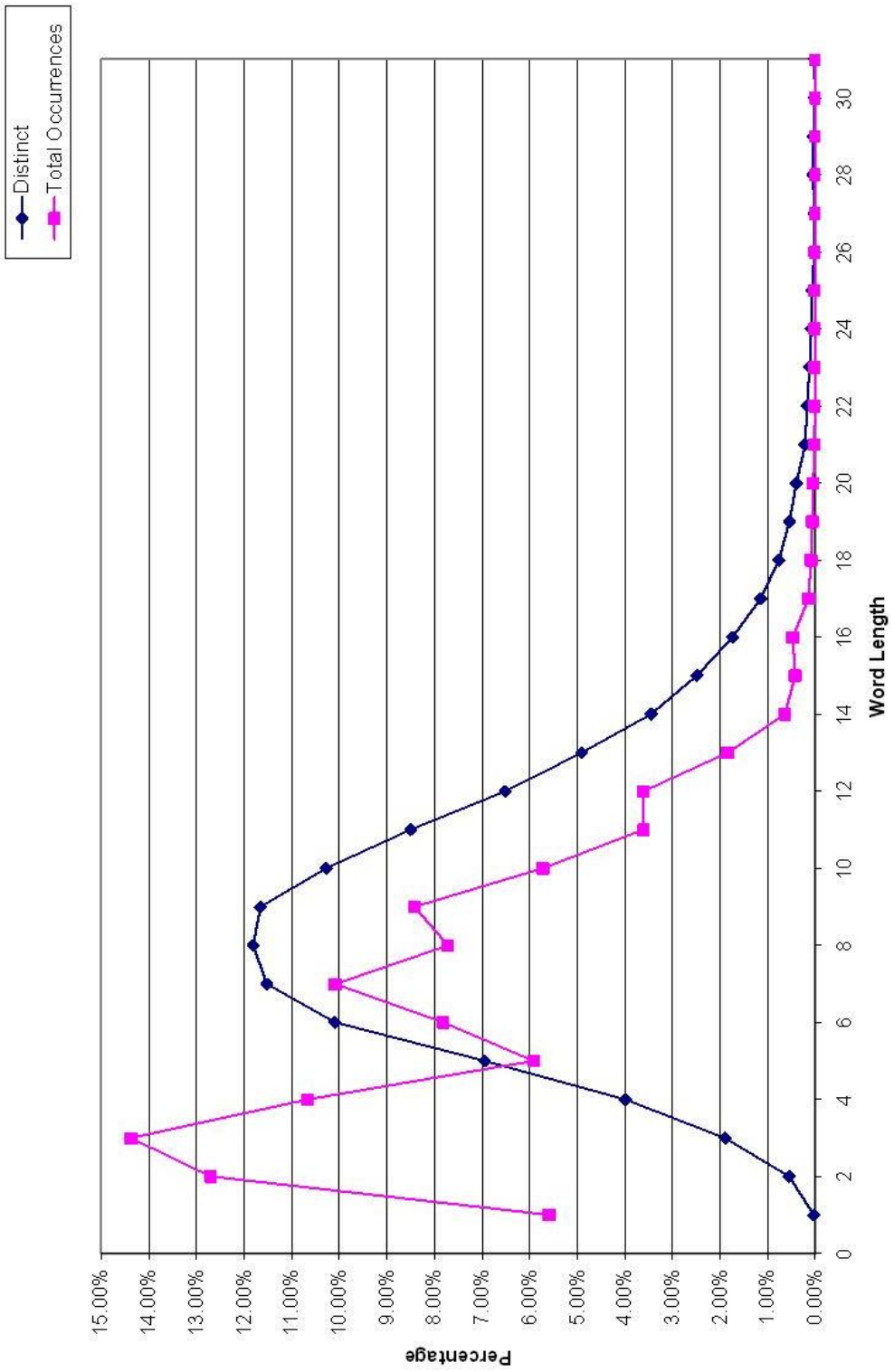

Figure 6.5: Distribution of words according to length

6.5.2 Dates

In July 2003 the GDB contained 909440 dates. Most dates fall in the period 1600 to 2003. There are some dates outside this range, but they are assumed to be incorrect. Table 6.6 and 6.7 shows the distribution of dates from 1600 to 2003. See figure 6.6 for a graphical representation. The two unusually high spikes at 1983 and 1990 represents the intensive use of voter's roles to capture residence and occupation details for many of the individuals in the GDB.

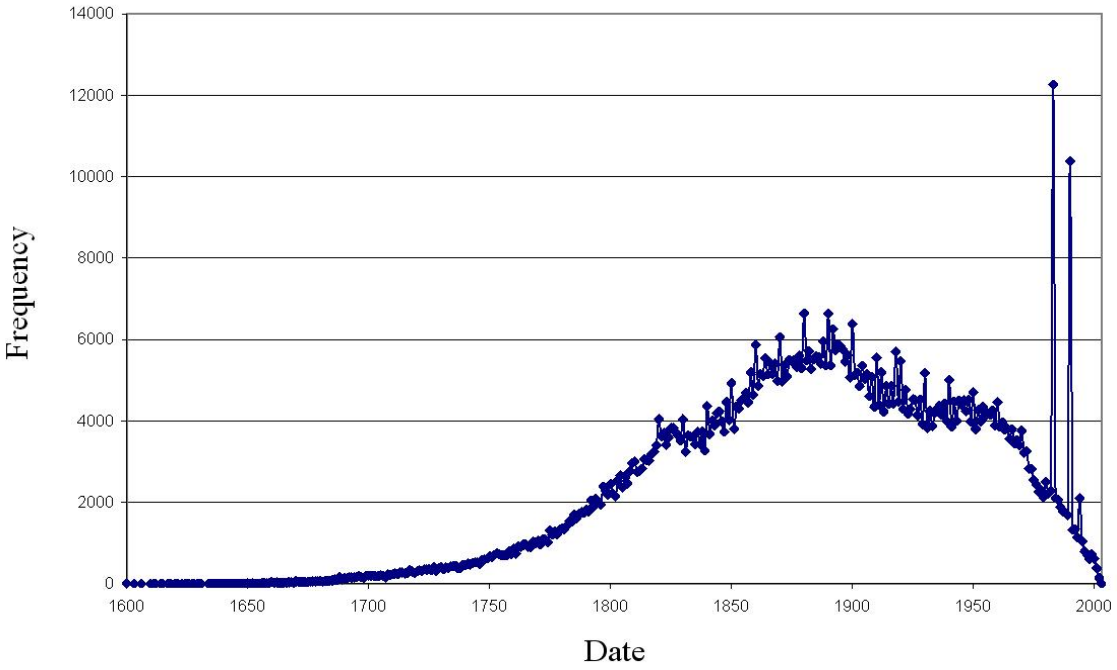

Figure 6.6: Distribution of Dates

Date	Frequency	Date	Frequency	Date	Frequency	Date	Frequency
1600	4	1662	36	1711	254	1760	864
1603	1	1663	22	1712	237	1761	741
1606	2	1664	17	1713	279	1762	927
1610	1	1665	34	1714	277	1763	913
1611	1	1666	28	1715	254	1764	961
1612	2	1667	37	1716	275	1765	971
1614	1	1668	42	1717	338	1766	908
1615	1	1669	31	1718	301	1767	907
1617	1	1670	60	1719	267	1768	1033
1618	1	1671	50	1720	307	1769	1009
1620	2	1672	48	1721	331	1770	1053
1621	1	1673	54	1722	315	1771	968
1623	2	1674	43	1723	356	1772	1088
1624	1	1675	59	1724	348	1773	1095
1625	2	1676	51	1725	371	1774	1025
1626	2	1677	62	1726	343	1775	1315
1627	2	1678	61	1727	406	1776	1206
1628	2	1679	67	1728	319	1777	1288
1629	1	1680	70	1729	389	1778	1216
1630	2	1681	57	1730	405	1779	1336
1631	1	1682	64	1731	364	1780	1356
1634	2	1683	64	1732	401	1781	1337
1635	7	1684	74	1733	393	1782	1429
1636	4	1685	86	1734	432	1783	1543
1637	3	1686	111	1735	435	1784	1524
1638	2	1687	96	1736	443	1785	1697
1639	4	1688	171	1737	386	1786	1619
1640	5	1689	107	1738	389	1787	1723
1641	1	1690	145	1739	462	1788	1759
1642	3	1691	137	1740	464	1789	1748
1643	4	1692	161	1741	499	1790	1826
1644	4	1693	154	1742	473	1791	1786
1645	8	1694	154	1743	510	1792	2051
1646	4	1695	167	1744	529	1793	1889
1647	3	1696	193	1745	525	1794	2085
1648	5	1697	162	1746	497	1795	1990
1649	6	1698	147	1747	592	1796	1946
1650	12	1699	192	1748	594	1797	2384
1651	5	1700	205	1749	625	1798	2291
1652	15	1701	187	1750	681	1799	2187
1653	8	1702	202	1751	662	1800	2453
1654	9	1703	183	1752	703	1801	2212
1655	10	1704	186	1753	748	1802	2145
1656	16	1705	211	1754	718	1803	2534
1657	18	1706	202	1755	704	1804	2665
1658	20	1707	156	1756	706	1805	2361
1659	25	1708	233	1757	694	1806	2654
1660	46	1709	227	1758	803	1807	2463
1661	24	1710	236	1759	731	1808	2767

Table 6.6: Date distributions 1600 to 1808

Date	Frequency	Date	Frequency	Date	Frequency	Date	Frequency
1809	2964	1858	5200	1907	4606	1956	4191
1810	3004	1859	4639	1908	5086	1957	4149
1811	2742	1860	5873	1909	4345	1958	4261
1812	2773	1861	4857	1910	5555	1959	3885
1813	2830	1862	5147	1911	4403	1960	4462
1814	3071	1863	5105	1912	5197	1961	3857
1815	3040	1864	5552	1913	4229	1962	3965
1816	3022	1865	5144	1914	4858	1963	3783
1817	3185	1866	5432	1915	4425	1964	3813
1818	3251	1867	5158	1916	4861	1965	3569
1819	3400	1868	5407	1917	4424	1966	3797
1820	4046	1869	4987	1918	5708	1967	3445
1821	3625	1870	6058	1919	4468	1968	3536
1822	3703	1871	4971	1920	5471	1969	3405
1823	3413	1872	5374	1921	4295	1970	3760
1824	3591	1873	5100	1922	4758	1971	3216
1825	3828	1874	5496	1923	4178	1972	3258
1826	3827	1875	5463	1924	4291	1973	2827
1827	3713	1876	5497	1925	4531	1974	2813
1828	3644	1877	5324	1926	4508	1975	2548
1829	3517	1878	5603	1927	4143	1976	2435
1830	4044	1879	5296	1928	4524	1977	2258
1831	3240	1880	6643	1929	3927	1978	2303
1832	3654	1881	5484	1930	5171	1979	2128
1833	3625	1882	5722	1931	3830	1980	2502
1834	3619	1883	5270	1932	4251	1981	2222
1835	3432	1884	5517	1933	3875	1982	2294
1836	3733	1885	5581	1934	4199	1983	12261
1837	3430	1886	5575	1935	4307	1984	2104
1838	3741	1887	5410	1936	4377	1985	2073
1839	3277	1888	5951	1937	4148	1986	1889
1840	4360	1889	5363	1938	4435	1987	1774
1841	3675	1890	6639	1939	3987	1988	1745
1842	4011	1891	5366	1940	5016	1989	1682
1843	3900	1892	6254	1941	3858	1990	10379
1844	4194	1893	5740	1942	4478	1991	1328
1845	4233	1894	5889	1943	4003	1992	1344
1846	3991	1895	5842	1944	4499	1993	1144
1847	3736	1896	5750	1945	4400	1994	2106
1848	4463	1897	5452	1946	4511	1995	1047
1849	4021	1898	5615	1947	4237	1996	798
1850	4936	1899	5064	1948	4523	1997	717
1851	3804	1900	6384	1949	3985	1998	622
1852	4352	1901	5138	1950	4709	1999	722
1853	4310	1902	5184	1951	3806	2000	616
1854	4510	1903	4854	1952	4268	2001	383
1855	4538	1904	5358	1953	4000	2002	149
1856	4696	1905	5034	1954	4348	2003	11
1857	4453	1906	5157	1955	4135		

Table 6.7: Date distributions 1809 to 2003

6.5.3 Names

Unlike general words, all the names in the name fields of the GDB are relevant to searching. Almost all the stored individuals have at least one name associated with them (it is very seldom that a person was added without knowing at least their surname or one first name) and most have more than one name. In July 2003 there were 36297 different names in the GDB. These names occur a total of 1557560 times in the database, 520403 times as surnames, 512669 times as male names, and 524488 times as female names.

The same name sometimes occurs as more than one type of name (for example, the name *Pieter* occurs as both a male and female name, and the name *Wessel* occurs as both a surname and a male name). There is thus no distinction made between different types of names. Table 6.8 shows how names are distributed in the different categories. See figure 6.7 for a graphical representation.

Name Occurring As	**# Names**	**% Names**	**# Occur**	**% Occur**
Surname	14127	38.92%	58974	3.79%
Male	6000	16.53%	34772	2.23%
Female	10907	30.05%	143336	9.20%
Surname and Male	2391	6.59%	77137	4.95%
Surname and Female	922	2.54%	8895	0.57%
Male and Female	487	1.34%	578746	37.16%
Surname, Male and Female	1463	4.03%	655700	42.10%
Total:	36297	100.00%	1557560	100.00%

Table 6.8: Distribution of Names according to Type of Name

6.6 Removed Terms

A stop list for the words in the GIS was drawn up manually. All words with length less than 3 characters were added to the stop list. From table 6.5 it can be seen that 480 distinct words and were thus removed (i.e. the total number of words with length less than 3) . Accordingly a total of 353398 occurrences will not have to be stored in the index.

From the 98 words that occurred more than 3000 times (see table 6.1), 53 were added to the stop list. The 9 most frequently occurring words were among the invalid words.

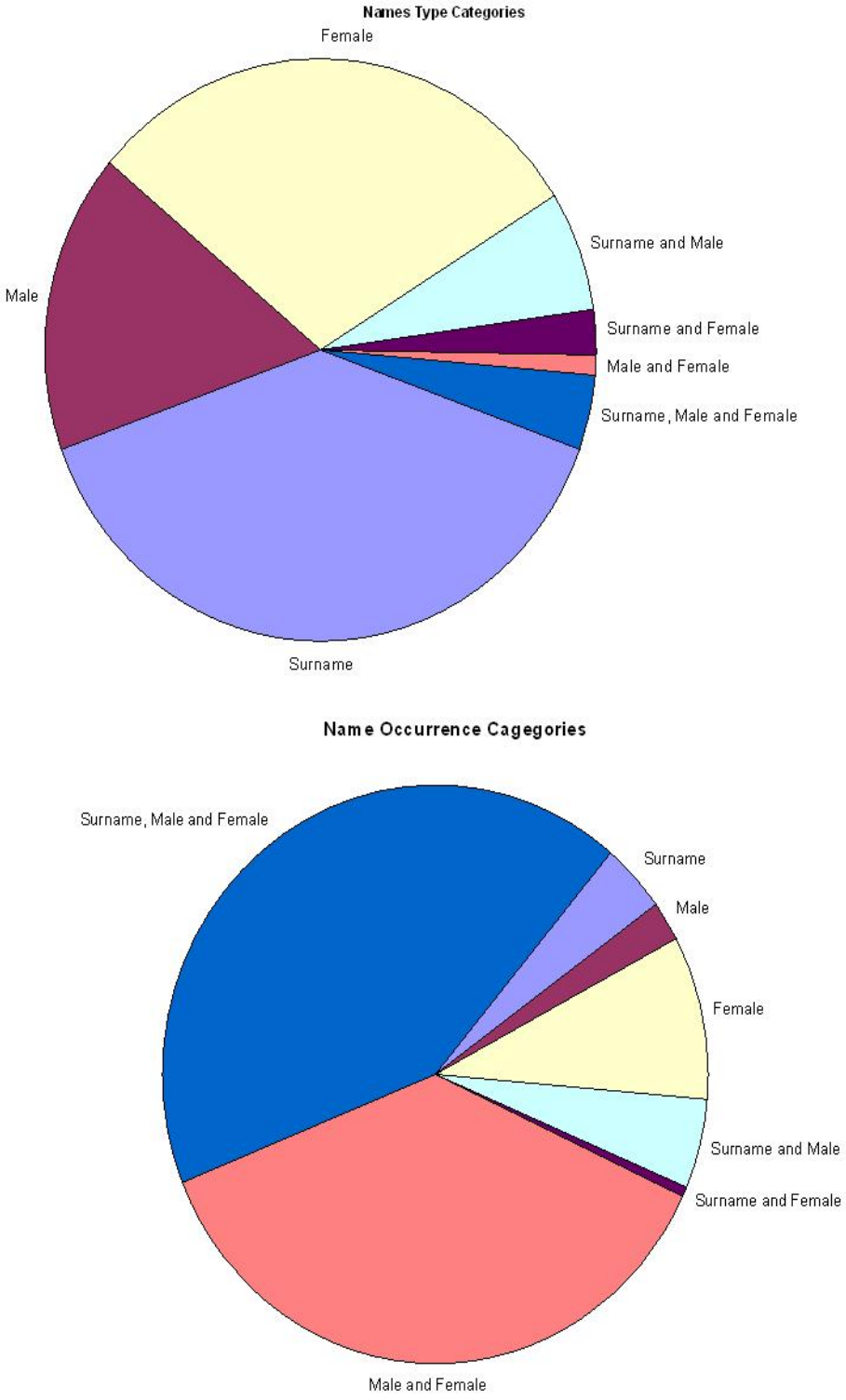

Figure 6.7: Distribution of Names

The new list of terms occurring more than 3000 times after the removal of some terms can be seen in table 6.9.

Word	**# Occurrences**	**Word**	**# Occurrences**
BOER	16148	MIDDELBURG	4490
KAAPSTAD	13744	BEAUFORT-WES	4263
PRETORIA	13351	COLESBERG	4236
HAAR	13197	BURGER	4201
GRAAFF-REINET	11661	ONGETROUD	4151
DOOD	9443	KAAP	3995
JOHANNESBURG	9374	GERMANY	3990
SWELLENDAM	8980	JONK	3898
UITENHAGE	8929	ENGLAND	3789
PAARL	8451	HUMANSDORP	3764
GEORGE	8348	MARIA	3627
CALEDON	8247	SWANEPOEL	3581
ELIZABETH	7543	TVL	3402
BLOEMFONTEIN	7217	HOSPITAAL	3392
STELLENBOSCH	7126	JACOBUS	3344
CRADOCK	6320	CAPE	3298
AAN	5772	JOHANNA	3296
PORT	5715	OOK	3284
JOHANNES	5116	VIR	3190
SOMERSET-OOS	5090	VOOR	3119
POTCHEFSTROOM	4909	ROBERTSON	3098
WORCESTER	4773	KROONSTAD	3081
TULBAGH	4662	RUSTENBURG	3051

Table 6.9: Words Occurring more than 3 000 times after removing non-index terms

The stop list contains a total of 4475 words. Combined they constitute a total number of 937008 occurrences. By making use of a stop list, a reduction of 48.5% in the occurrences to be stored in the index was achieved.

Investigations using stemming with a simplified version of Porter's algorithm were performed. It was found that small reductions could be made to the size of lexicon, but that the risk of terms with different meanings being indexed as one term was too great. A safer approach to achieving some of the benefits of stemming would be to make words with the same stem part of the same similarity set (see sections 3.1.4 and 3.2.1). A stemming algorithm was used to group certain surnames into equivalent groups, for example, removing surname prefixes like *van der*, *de* and *du*.

6.7 Equivalent and Similarity Groups

Populating the database is not part of the scope of this project. Therefore, although the functionality is provided in the implementation of the GIS, not all terms have been grouped into equivalent or similarity groups.

Names and surnames have been grouped into equivalent groups, with a few similarity groups for testing purposes. It is envisioned that words in Afrikaans and English that have the same meaning will be grouped into equivalence groups, for example, *geneesheer*, *dokter* and *doctor*. Apart from better search results, such groupings will greatly reduce the size of the index.

6.8 Conclusions

In this chapter a clear picture of the terms appearing the the GDB was given. Methods to refine the lexicon were described and techniques used to do so were discussed. The number of index terms were significantly reduced.

Chapter 7

IMPLEMENTATION

7.1 Introduction

In chapter 4 an information retrieval model for the GIS was discussed. The focus of this chapter will be the algorithms used to implement the retrieval model. The approach followed will be motivated, and restrictions on the model necessary to make fast retrieval possible will be discussed.

7.2 Retrieval Approach

The goal of the search algorithm is to evaluate equations 4.1 and 4.2, on page 53, for each record. As was mentioned in chapter 5, a sequential search through all records would be ineffective. To speed up search times, an index must be used that point to some data structure that contains information that can be used to evaluate equations 4.1 and 4.2. There are two approaches that may be followed:

1. Use an index on record number that points to term postings sorted in lexicographic order. For each posting all information needed to calculate the term weight, given by equations 4.7 and 4.8, must be stored. The search process will consist of doing an intersection merge on the terms appearing in the query with the postings for each record, by traversing the index linearly. The resultant list can then be used to evaluate equations 4.1 and 4.2.

2. Make use of an index on terms that points to postings of the records that contain the term. Each posting contains the information needed to calculate the weight of the index term in that specific record. During searching, the record postings for each term will undergo a union merge, and the resultant list will be used to calculate the rank of each record.

Both approaches will require about the same amount of disk space. The second approach will provide a faster search algorithm since in most cases not all records will have to be evaluated. The drawback of the second approach is that the indexing process will take longer. This is not seen as a significant problem since the GDB is semi-static (it grows by a relatively small percentage every month) so it will not be necessary to rebuild the index often (at the moment dynamic updating of the index is not supported). The second approach was followed to implement the search algorithm in the GIS.

7.3 Data structures

In the section the data structures used by the search algorithm will be discussed. The actual use of these data structures is discussed later in this chapter. In all cases when the word *index* is used, it refers to an inverted file as discussed in chapter 5. To facilitate fast location, the index terms are stored in a B-tree.

7.3.1 Equivalent and Similarity Database

The creation and retrieval of equivalent and similarity sets (see section 3.2.1) is done through an equivalent and similarity term database. For each distinct term in the GDB the following record is stored:

```
TTermRecord = record
  Status : integer;                  {  4 bytes}
  Term : String;                     { 32 bytes}
  Code : integer;                    {  4 bytes}
  EquivalentTerm : String;           { 32 bytes}
  EquivalentCode : integer;          {  4 bytes}
  SimilarTerms : array[1..5] of String;  {5 x 32 bytes}
  SimilarCodes : array[1..5] of integer; {5 x  4 bytes}
  TotalFrequency : integer;          {  4 bytes}
```

```
    MaleNameHz : integer;              {  4 bytes}
    FemaleNameHz : integer;            {  4 bytes}
    SurnameHz : integer;               {  4 bytes}
    PlaceNameHz : integer;             {  4 bytes}
    ValidPersonName : boolean;         {  1 bytes}
    SearchTerm : boolean;              {  1 bytes}
  end;                                 {278 bytes}
```

`Term` contains the actual term. Note that a maximum of 31 characters are allowed per term. Each term is assigned an unique 4-byte code. This code has the same lexicographical order as the term it represents. In essence the codes are generated by sorting the terms alphabetically and then numbering them (more detail will be given in a later section). Each term has an equivalent or characteristic term associated with it, stored in `EquivalentTerm`. The code of the equivalent term is stored in `EquivalentCode`. There can be a maximum of five similar terms associated with each term, stored in the `SimilarTerms` array. The code for each similar term is stored in the `SimilarCode` array. By storing the codes for term equivalent term and the similar terms, information is duplicated. The reason for this will be discussed later.

The total number of occurrences, number of occurrences as a male, female, place name and surname are all stored for statistical purposes and validity checks. `ValidPersonName` is used for validity checks. The field `Status` is used by the database system, and is not used by the search algorithm.

The similarity database also acts as the stop list. If the field `SearchTerm` is set to *false*, then the term should not occur in the search index.

Two indexes are used in the similarity database. The first index, the similarity term index (STI), contains all the terms in the GDB and points to the record for each term. The second index, the similarity code index (SCI), contains the unique code for each term and points to the record associated with that term. Note that it is faster to locate the record for a term using the code index since the term index must make provision for 31 byte entries, while each entry in the code index is only 4 bytes long. There are thus fewer

comparisons and less disk reads involved in searching the code index.

7.3.2 Term Search Index

The term search index (TI) consists of the codes of all search terms (i.e. the equivalent term codes of all terms that are not in the stop list). Each entry points to a linked list of postings, one for each record in which the term occurs. Each posting (element in the linked list) has the following form:

```
TPosting = record
  Status : integer;              { 4 bytes}
  RecordID : integer;            { 4 bytes}
  Frequency : integer;           { 4 bytes}
  Occurrences : int64;           { 8 bytes}
  Next : integer;                { 4 bytes}
end;                             {24 bytes}
```

The `Status` field is used by the database management system. `RecordID` is the record number of a record that contains the term at least once. `Frequency` is the total number of times that the term appears in the record.

The `Occurrences` field stores the fields in which the term occurs in the record. The system currently caters for 64 possible fields. The current fields can be seen in appendix D on page 131. Each field is represented by a unique 64-bit integer that is an exponent of 2. In other words each field is assigned a 64-bit binary code of which only one bit is set to 1. For example, the *Surname* field is represented by 2 and the *Information* field is represented by 8. In binary, these numbers are 10 and 1000 respectively. The `Occurrences` field stores the bit-wise **OR** of all the fields in the record where the term is located. For example, if the term was located in both the *Surname* and the *Information* fields, then `Occurrences` would store 1010. It may be tempting to think that the `Occurrences` field makes the `Frequency` field unnecessary since the number of occurrences can be found by counting the number of bits set to 1 in `Occurrences`. This would not work in situations where a term occurs more than once in the same field. The total number of occurrences must thus still be stored separately.

Note that the approach followed above is not perfect. Information will be lost if a term appears in several different fields and in some of those fields more than once. This will negatively affect the Complete Matching weighting scheme described in section 4.4.2 on page 58. Despite the drawback, it was decided to use this approach for the following reasons:

1. This approach makes the calculation of the distance function very fast.
2. Storing all information about a term's occurrence would imply a dramatic increase in the space used on disk (and hence also imply more disk reads).
3. Only very few records contain a term several times in the same field. The ranking algorithm will thus be affected very infrequently.

The field `Next` stores the location of the next posting in the linked list. By using a linked list, it is not necessary to have more than one entry for the same term in the B-tree (some terms appear in thousands of different records). Creating one big record to store all postings is impractical because of the large difference in the number of occurrences of terms (see table 6.3 and figures 6.2 and 6.3). The linked list is sorted in ascending order of `RecordID`.

The first posting in the linked list for any term contains the total number of records in which the term occurs in the database (i.e. the number of posting records). This number is used as the f_i of equations 4.7 and 4.8. It is not strictly necessary to store this value, since it can be calculated by counting the postings. However, counting the postings and afterwards evaluating equations 4.7 and 4.8 would imply two passes over the linked list.

A separate term index is created for Person, Mother, Father, Spouse and Children. In each case the `RecordID` that is used in each posting is the Person's record number.

7.3.3 Date Search Index

The weight of a date in a record is calculated according to its proximity to the query date. An indexing approach similar to terms would imply an index consisting of dates

pointing to postings of records in which each date occurs. From equation 4.11 on page 62 it can be seen that any date will contribute some weight. A search for a date would imply visiting every posting of every date and calculating a weight for that posing. All linked lists must then be merged to find the final linked list containing the weight of the date in each record. As stated in section 6.5.2, there are almost 1000000 dates in the GDB. Clearly this approach would be very time consuming and thus impractical.

A more efficient approach is to have an index on record number pointing to postings of dates and their locations in the records. This approach implies a linear iteration over all record numbers, but not a merge of all linked lists. The fact that the final linked list containing weights for a query date in each record can be found after one pass over the index, makes this approach far superior to the first approach discussed.

Search times can be considerably improved by asserting that searching for dates should only be seen as a refinement on an existing query. We assume that a query will never consist of dates alone. The lists of records containing the terms can then be found first and only those records are then considered for the date search. Note that searching for dates is a far more computationally expensive task than searching for normal terms, since far more linked lists must be evaluated. The number of linked list for terms is equal to the number of terms in the query, whereas the number of linked list for dates is equal to the number of records that contained any of the terms in the query. Typically there would be several thousand date linked lists to evaluate.

The date search index (DI) consists of record numbers, each pointing to a date posting linked list. Each element will be of the following form:

```
TDatePosting = record
   Status : integer;              { 4 bytes}
   Field : integer;               { 4 bytes}
   StartDate : integer;           { 4 bytes}
   EndDate : integer;             { 4 bytes}
   Next : integer;                { 4 bytes}
 end;                             {20 bytes}
```

As before, `Status` is only used by the database management system. `Field` stores a 32-bit integer indicating the field in which the date occurred. This integer is similar to the 64 bit integer used to store the field for normal terms. See appendix E on page 133 for a list of all possible date field values. `StartDate` and `EndDate` store the start and the end date for a period. Each date is represented by an integer which stores the number of days that have elapsed between 1 January 1 A.D and the date to be stored. The `Next` field stores the position of the next posting in the posting linked list.

A separate date index is created for Person, Mother, Father, Spouse and Children. In each case the record number that is used in the index is the Person's record number.

7.3.4 Relevance Table

The function $\mathcal{D}$, discussed in section 4.3, makes use of a relevance table to determine how relevant the field where a term was located is to the field specified in the query. The relevance table was drawn up manually by experienced genealogists. A separate relevance table is used for term and dates, since a term can never be located in a date field or vice versa.

The relevance table only stores the relevance values for the leaf fields in the field tree for a person record, i.e. only values for H_i where $H_i \in H_L$ (see section 4.3 on page 54).

An extract of the term relevance table can be seen in table 7.1, where the vertical axis represents the required field and the horizontal axis represents the field where the term was located.

Field	**First Name**	**Surname**	**Nickname**	**Information**	$\cdots$
First Name	1	0.7	0.9	0.8	$\cdots$
Surname	0.7	1	0.7	0.7	$\cdots$
Nickname	1	0.6	1	0.9	$\cdots$
Information	0.8	0.8	0.8	1	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Table 7.1: Extract of the term relevance table

Note that the table is not symmetric. For example, if a query states that a term must be found in the *Nickname* field, and it is located in the *FirstName* field then the relevance is 1. If the query states that a term must be located in the *FirstName* but it is found in the *Nickname* then the relevance is 0.9. The reason for this seeming discrepancy can be explained as follows: If a user thinks that the name they have is a nickname and it is located in the name field, then it is likely that the name was in fact the person's real name or that the data capturer was under the impression that it was the person's real name. However, if the user specifies that the term is a first name and it is located in the *Nickname* field, it implies that the user did not know the real name of the person, and that person should thus be ranked lower than an individual who actually had the term as a first name.

The relevance tables are implemented as two dimensional arrays where each cell, referenced by `Table[`*column*`][`*row*`]`, contains the relevance of the required field, represented by *column*, to the located field, represented by *row*. Rows and columns are numbered as $\log_2 x + 1$ where x is the integer that represents each field (see appendix D on page 131). For example, the row representing the field *Medical Details* which had the code 268435456, is numbered 29.

Queries can specify that a term belongs to a subtree. Each internal node in the field tree, i.e. for H_i where $H_i \in H_N$, is assigned an integer that is the bit-wise OR of the codes of the fields in its subtree (the codes of all fields $H_j \in H_L$ where $H_j \preceq H_i$). The binary value of the resultant integer is a bit string with all the values of where the term should appear set to 1. This integer will be referred to as the Query Structure Value.

The pseudo-code for the implementation of the relevance function, $\mathcal{D}$ can be found in appendix F on page 134. Firstly, a logical AND is performed on the `Occurrences` field of the term posting and Query Structure Value. If the resultant integer is 1, then the term occurs in one of the relevant locations and a relevance of 1 is returned. If the resultant integer is 0, then the term does not occur into any of the desired locations. The relevance function then compares each desired field with each of the occurrence fields and then re-

turns the highest relevance found in the relevance table.

Although it is not necessary to create a different relevance function for the Complete Matching approach (see section 4.4.2 on page 58), it is used slightly differently. Instead of passing the `Occurrences` field and the Query Structure Value to the relevance function, each code representing leaf nodes is passed to the relevance function individually. The resultant values are then summed (see equation 4.6 on page 59).

7.4 Indexing Process

The goal of the indexing process is to populate the data structures discussed in the previous sections. In so doing, most of the processing is removed from the actual search time.

7.4.1 Equivalent and Similarity Database

The creation of the Equivalent and Similarity database is mostly a manual process. All terms in the database are located by iterating through all records sequentially. Each new term found is placed in a record as described in section 7.3.1. An entry is made into the STI for each new record. If a record was already created for a term, the record is retrieved and the relevant frequency fields updated. A count of the number of unique terms is also kept.

The second phase of the algorithm is to assign unique codes to each term. A step size between terms is calculated as the maximum code (in this case 2147483647) divided by the total number of terms plus 1. The terms are then processed alphabetically and each term sequentially assigned a multiple of the step size, so that the codes maintain the lexicographic order of the terms. The motivation behind assigning codes in steps is that terms are dynamically added to the similarity database. When new terms are added they must be assigned a unique code that maintains the lexicographic order. These terms are assigned a code between the codes of their lexicographic predecessors and successors.

The pseudo-code for the indexing of the Equivalent and Similarity database can be seen

in appendix G.

After all terms are in the Similarity database, the similarity and equivalent groups are set up manually. The stop list is also manually created by setting some terms to non index terms.

As new records are added to the database, a code is assigned to each new term. These terms are manually placed into equivalent or similarity groups periodically. All terms have to be assigned new codes when one of the code gaps between terms are filled with new terms.

7.4.2 Term and Date Search Index

The term and date search indexes (TI and DI) are the most important, as they make the entire retrieval algorithm possible.

To create the indexes, all records in the GDB are traversed sequentially. All fields in each record are considered separately.

Each term in a field is located in the STI (it is important that the equivalent and similarity database should be up to date before creating the term and date search index), and the relevant `TTermRecord` read. Each equivalent and similarity code of the term is then located (or a new entry created if it does not yet exist) in the TI. The posting linked list that the term points to is then retrieved. If the posting linked list already contains a posting for the current record, the `TPosting` record is retrieved, the `Frequency` field incremented and a logical OR performed on the 64-code representing the current field and the `Occurrences` field. If no postings for the current record exist, a new posting is created containing the relevant data for the term.

For each period in a field, the record number is located in the DI. If no entry exists, a new entry is made. The linked list for the record number is then retrieved. A `TDatePosting`

record is created containing the relevant field, start date and end date. This posting is inserted in the linked list.

The above process is repeated for the individual in the current records' mother, father, spouses and children, with the entries made in the relevant indexes.

The pseudo-code for the term and date indexing can be seen in appendix H on page 137.

Note that, because the `TPosting` linked list is sorted in ascending order of record number, the algorithm is much faster if the records in the GDB is traversed in descending order of record number. New postings can then always be added to the beginning of each linked list, making inserting much faster.

7.5 Optimization

7.5.1 Repacking Postings

It was mentioned earlier that an efficient search algorithm should minimize the number of disk reads. The algorithm described in section 7.4.2 does not index all postings for a specific term before indexing the next term. Records are processed sequentially and terms are indexed as they occur in the records. As a result the postings for a specific term are not written to disk next to one another. Figure 7.1 depicts what the postings for a database containing 5 records may look like on disk. Note that each block contains a tuple of the form (*term number*, *posting number*).

3,1	3,2	2,1	5,1	5,2
3,3	1,1	2,2	4,1	3,4
5,3	5,4	2,3	4,2	1,2
2,4	5,5	1,3	5,6	5,7
3,4	1,4	3,4	1,5	3,6

Figure 7.1: Unpacked Postings

Assume that in one disk read three postings are read from the disk. To retrieve all the postings for term 1, it would thus require 4 separate disk reads. Furthermore, after each disk read, the reading head must be moved to the location from which it must read the next term. This is a time consuming operation.

The disk read problem can be alleviated by repacking the postings after the indexing process. Although this may take a considerable amount of time, it only has to be done once after the index is created. Figure 7.2 depicts the same postings as in figure 7.1 after they have been repacked. Note that all the postings for term 1 can now be read with only 2 disk reads and very little movement of the read head. In an actual computer where far more than 3 postings are read during each disk read, repacking postings makes a considerable difference to the algorithm's response time.

1,1	1,2	1,3	1,4	1,5
2,1	2,2	2,3	2,4	3,1
3,2	3,3	3,4	3,4	3,4
3,6	4,1	4,2	5,1	5,2
5,3	5,4	5,5	5,6	5,7

Figure 7.2: Packed Postings

7.6 Search Process

Queries are assumed to be in disjunctive normal form. This approach was also followed by [CKR01]. Define the set of individuals that can be used in queries to locate a person to be: $\Delta = \{$`Person`, `Mother`, `Father`, `Spouse`, `Child`$\}$. Let Λ be the set of all fields that can be used in a query (these fields are listed in appendix C). Formally, queries have the following form:

$$\bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} \delta_{i,j}(\lambda_{i,j}(k_{i_j})) \tag{7.1}$$

where $\delta_{i,j} \in \Delta$, $\lambda_{i,j} \in \Lambda$ and k_{i_j} is a term or a period. See appendix B for the EBNF of queries. The evaluation of a query will consist of first evaluating equation 4.1 for the

conjunctive components and then evaluating equation 4.2 for the disjunctive components.

It is assumed that a value for p in equations 4.1 and 4.2 is specified by the user before the search process starts (see section 2.6.4 on page 22 for how varying p changes the retrieval algorithm). The user can also specify if weighted query vectors should be used (it was found in practice that better results are obtained from using using un-weighted query vectors, see chapter 8) and if the Fast Matching or the Complete Matching weighting technique should be used (see sections 4.4.1 and 4.4.2).

Records are ranked in response to a query by first locating each term in the STI and obtaining its equivalent code. The conjunctive components are evaluated first. Each equivalent code in a conjunctive component is then located in the TI and the relevant posting linked list retrieved. This linked list is used to calculate a weight (using equation 4.7 or 4.8) for the current term in every record in which it occurs. This information is stored in a new linked list in memory.

After a weight linked list has been found for each term in the conjunctive components, equation 4.1 is evaluated for conjunctive component by means of a union merge on all the weight linked lists. The resultant linked lists will contain a ranking of each of the records in which terms occurred. If a conjunctive component contained any periods, the record number of each record in which terms were located is used to locate the date posting linked lists using the DI. An new ranking for each record is then calculated using equation 4.1 with equation 4.11 giving the weights for date terms.

Once ranking is complete for all conjunctive components, the disjunctive component can be evaluated. The result of equation 4.1 for each conjunction acts as a weight in equation 4.2. The weight for the terms and periods in the disjunctive component is found in the same way as in the conjunctive components. Equation 4.2 is also evaluated by performing a union merge of the different weight linked lists.

The result of the above process is a linked list containing the final ranking of each record

to be retrieved. The linked list is sorted in ascending order of record number.

The pseudo-code for the retrieval and ranking algorithm can be seen in appendix I on page 139.

In the algorithm just described, two passes are made over each linked list. The first pass is made when the postings are read from disk and the linked list containing the record occurrences of the term in the GDB is created. During this pass the numerator of equation 4.7 or 4.8 is calculated and the maximum numerator (the denominator) found. A second pass is made when the linked lists are merged. Equation 4.1 or 4.2 is evaluated during the merge after the numerator of equation 4.7 is divided by the denominator for each term occurrence. If the resultant linked list represents the ranking of a conjunction of terms, a third pass may be made over it as it is merged with the other linked lists in the disjunction.

Date searches are far less efficient than normal term searches. A date or period search implies an extra traverse of the linked list of the terms in the conjunction or disjunction where the period is specified. For each record number in the linked list, the numerator of equation 4.11 is calculated for each period posting and the maximum stored in the period linked list. This requires many disk reads since the record numbers are not known before query times and thus no repacking of postings can be performed. Furthermore, *all* date postings for the relevant records must be considered. A final merge must be performed on the linked list to divide the numerator of equation 4.11 by the denominator and to calculate equation 4.1 or 4.2.

Note that the final linked list is not sorted in order of ranking. Even a very efficient sort algorithm would require several passes of the list. Rather than sorting the entire list in one go, the top 10 highest ranking records are removed and added to a separate sorted list (a process that can be done in a single pass). As the user views the results 10 at a time, the sorted list will grow longer until it contains all the record numbers. It is unlikely that this will ever happen since the user would usually only look at the first few highest ranked records before refining his or her query. Although the total sort time would be

longer than a complete sort at the beginning if all records are eventually viewed by the user, a single pass over the linked list takes a short enough time not to be noticed and so no time is wasted in unnecessarily sorting the list.

7.7 Implementation Performance

Reporting on retrieval times is difficult because of the factors that influence response times.

Firstly there is *caching*, the process whereby data read from the hard drive is automatically stored in memory. When the data is needed again, it can be accessed from memory, making several disk reads unnecessary. As a result of caching, retrieval times depends on previous queries.

The second factor that affect response times is the hardware on which the GIS is run. Processor speed and the configuration of the hard drive can greatly affect the time that a search takes.

7.7.1 Indexing

For the 551440 records that are currently in the GDB, indexing takes over 4 hours. This number includes the time it takes to repack the postings. Records are thus indexed at a rate of around 34 per second, a reasonable rate considering that no further sorting is required.

7.7.2 Searching

Table 7.2 lists the queries used in searches that were performed. The computer was rebooted after each query (for searches using both packed and unpacked postings). Some of the terms, for example, **MERWE** and **WILLEM** was selected because of the large number of times they occur in the GDB, thus constituting a worst case scenario.

Table 7.3 lists the statistics for queries that were performed. The column labelled # represents the number of records that were returned in response to a query. Groups of

	Query
A	(FirstName Willem) **AND** (Surname Merwe)
B	(BirthPlace Willem) **AND** (BaptismPlace Merwe)
C	(FirstName Cornelis) **AND** (Surname Plessis)
D	(BirthPlace Cornelis) **AND** (BaptismPlace Plessis)
E	(FirstName Willem) **AND** (Surname Merwe) **AND** (BirthDate 1880)
F	(FirstName Cornelis) **AND** (Surname Plessis) **AND** (BirthDate 1980.01.12)
G	((Mother) Surname Merwe) **AND** ((Father) FirstName Cornelis) **AND** ((Father) Surname Plessis) **AND** ((Child) FirstName Willem)

Table 7.2: Queries

columns labelled **I** represents initial queries and groups **C** represents queries that have been repeated (i.e. where caching play a role). Columns labelled **T** represent the total time for a search, and columns labelled **D** represents time spent on disk reads. All times are in seconds. Note that only the Fast Matching algorithm was used in the tests represented in table 7.3

		Unpacked				**Packed**			
		I		**C**		**I**		**C**	
Query	**#**	**T**	**D**	**T**	**D**	**T**	**D**	**T**	**D**
A	53424	16.623	16.561	0.313	0.25	0.579	0.532	0.298	0.235
B	53424	17.444	17.397	1.282	1.219	1.571	1.455	1.283	1.236
C	15345	17.0	16.969	0.11	0.094	0.313	0.297	0.109	0.094
D	15345	21.304	21.272	0.359	0.344	0.579	0.563	0.359	0.329
E	53424	24.798	24.72	1.375	1.297	8.272	8.194	1.392	1.329
F	15345	24.173	24.157	0.5	0.484	7.23	7.199	0.485	0.454
G	61136	31.074	31.012	0.375	0.296	0.715	0.704	0.36	0.281

Table 7.3: Retrieval times in seconds of various queries

The effect of repacking postings can be clearly seen. Note that for query **G** an improvement in search time of 30.36 seconds was found.

Table 7.3 also highlights the negative effect on search time due to using dates in queries. In both cases were dates were used (queries **E** and **F**), searches took almost 7 seconds longer than the same queries without the dates (queries **A** and **C**).

In queries **B** and **D**, terms were searched for in fields where they are not expected to occur. In such cases the term matching was not found by the logical AND performed on

the desired and actual locations of the term by the relevance function (see section 7.3.4) had to be evaluated for almost every hit. This had a considerable negative effect on search times for query **B**. The terms in query **D** does not occur as frequently in the GDB as the terms in query **B**, and was thus not as negatively affected.

Query **G** makes use of the relative indexes (Father, Mother, Child and Spouse) to locate a person. It is clear that, although longer search times were found, search times are reasonable and well worth the added functionality to the search algorithm.

A second set of tests were conducted to compare the Fast Matching with the Complete Matching algorithm. The results can be seen in table 7.4. The results shown are for the same queries as were used in the first test. The retrieval times are listed in seconds, and are all for repacked postings and cashed searches.

Query	**#**	**Fast Matching**	**Complete Matching**
A	53424	0.298	1.281
B	53424	1.283	1.297
C	15345	0.109	0.375
D	15345	0.359	0.375
E	53424	1.392	2.407
F	15345	0.485	0.74
G	61136	0.36	1.469

Table 7.4: Comparing Fast and Complete Matching algorithms

For the most part, the retrieval time for the Fast Matching algorithm is much faster than for the Complete Matching algorithm. This is to be expected, since far more processing time goes into the Complete Matching algorithm than into the Fast Matching algorithm. Note that the search times for queries **B** and **D** is virtually the same. These are the queries for which the fields specified for terms where not the fields where it was expected the terms would appear. In these cases the Fast and Complete Matching algorithms had to perform virtually the same computations.

All the search times for repacked postings in table 7.3 and 7.4 (even queries that contain

dates) are comparable to search times reported by Brin and Page for the first version of Google [BP98].

7.8 Conclusions

This chapter described how the retrieval model discussed in chapter 4 can be efficiently implemented. The algorithm was evaluated with respect to search times and it was found that it compares favorably to other retrieval systems.

Chapter 8

EVALUATION OF IMPLEMENTED ALGORITHMS

8.1 Introduction

The evaluation of an information retrieval system can be very difficult, and no perfect methods have emerged [RBJ89]. The most popular method of retrieval evaluation is *Recall* and *Precision* [VR89]. In general, the goals of an information retrieval system is firstly to retrieve as many relevant records as possible, and secondly to retrieve a minimal number of non relevant records. These two goals are often conflicting, as a system that retrieves *all* records will completely satisfy the first goal, while a system that retrieve *no* records will satisfy the second goal.

We can formalize the notion of the first goal by defining the concept of *Recall* as the ratio of the number of relevant records retrieved to the total number of relevant records. The second goal can be define as *Precision*, the ratio of the number of relevant records retrieved to the total number of records retrieved. A recall-precision graph is created by plotting the precision values versus different recall values.

Unfortunately, the nature of searches in the GIS makes the above approach inappropriate. When searching for an individual, only one record is relevant, therefore the Recall

value will only be 1 or 0, depending whether the individual is found of not (ignoring the possibility of duplicate individuals). The Precision value can be useful, if we define it as the number of records retrieved before the relevant record was found. In this chapter this new definition of precision will be the measure of how well a search algorithm performs.

8.2 Retrieval in the GIS

Apart from using the precision value to evaluate the search algorithm, it will be compared to the original search algorithms in the GIS. The original algorithm allowed for searches on name and birthdate. The results are sorted in alphabetical order and then on birthdate. For example, for the query `du Plessis, Mathys Cornelius`, the first 10 results are:

```
1. Du Plessis, Mathys Cornelis 1837.09.00
2. Du Plessis, Matthys Cornelis 1864.09.11
3. Du Plessis, Matthys Cornelis 1873.09.20
4. Du Plessis, Mathys Cornelius 1904
5. Du Plessis, Mathys Cornelius 1923.12.06
6. Du Plessis, Mathys Cornelius 1980.01.12
7. Du Plessis, Matheus Gideon 1944.08.18
8. Du Plessis, Matthys Heyns 1906.11.24
9. Du Plessis, Matthys 1865
10. Du Plessis, Matthys 1872
```

The results of the new search algorithm are displayed in order of the ranking assigned by the weighting formulas. The user can specify the p value (see section 2.3), whether weighted query vectors should be used, and whether the Fast or the Complete Matching algorithm should be used. The first 10 results of the query `{FirstName Mathys} AND {FirstName Cornelius} AND {Surname du Plessis}` are (a p of 3 and a un-weighted fast matching search was used):

```
1. Du Plessis, Matthys Cornelis 1864.12.04
2. Du Plessis, Mathys Cornelis 1837.09.00
3. Du Plessis, Mathys Cornelius 1980.01.12
4. Du Plessis, Mathys Cornelius 1923.12.06
5. Du Plessis, Mathys Cornelius 1904
6. Du Plessis, Matthys Cornelis 1874.01.12
7. Mattheus, Cornelius Johannes 1885.06.20
8. Du Plessis, David Gerhardus Cornelius 1872
9. Du Plessis, Cornelis Johannes 1804.11.25
10. Du Plessis, Cornelis Johannes Hendrik 1814.04.09
```

Note that, if a date was specified, then many of the relevant results would not have been returned by the original algorithm. For example, the query `du Plessis, Mathys Cornelius 1904` would yield:

```
1. Du Plessis, Mathys Cornelius 1904
2. Du Plessis, Mathys Cornelius 1923.12.06
3. Du Plessis, Mathys Cornelius 1980.01.12
4. Du Plessis, Matheus Gideon 1944.08.18
5. Du Plessis, Matthys Heyns 1906.11.24
6. Du Plessis, Matthys 1865
7. Du Plessis, Matthys 1872
8. Du Plessis, Mattheus Jacobus 1851.05.19
9. Du Plessis, Mattheus Johannes Andries 1839.12.03
10. Du Plessis, Mattheus Johannes 1810.05.04
```

To the query `{FirstName Mathys} AND {FirstName Cornelius} AND {Surname du Plessis} AND {BirthDate 1904}`, the new search algorithm would yield:

```
1. Du Plessis, Mathys Cornelius 1904
2. Du Plessis, Matthys Cornelis 1864.12.04
3. Du Plessis, Mathys Cornelis 1837.09.00
4. Du Plessis, Matthys Heyns 1907.02.03
5. Du Plessis, Mathys Cornelius 1980.01.12
6. Du Plessis, Mathys Cornelius 1923.12.06
7. Du Plessis, Matthys Cornelis 1874.01.12
8. Van Rensburg, Cornelius Mattheus 1900
9. Du Plessis, Gerrit Thomas Cornelius Ferreira 1899
10. Moggee, Charles Matthys Cornelis 1900.08.15
```

8.3 Evaluation

In this section searches will be performed and the results reported on and discussed. Actual case studies (not selected by the author) from genealogical sources [vH04] will ensure an accurate representation of the search capabilities of the GIS.

The tables reporting the results for the new search algorithm will be labelled in the following way: First the p value will be shown. If weighted query vectors is used, a **W** will appear in the label. A **F** will indicate the Fast Matching algorithm and a **C** will indicate the Complete Matching algorithm.

8.3.1 Case Study 1

Known Information

The person to be located is **Lance Lindenberg Tomlinson**, born 1884.10 and married in 1900 to **Sarah Johanna Uys**, born 1879.1.7 and died in 1918. He had a daughter named **Francina**.

Original search algorithm

The query used was `Tomlinson, Lance Lindenberg 1884.10`. The correct record contained the person's name as **Lance Miles Tomlinson**, born in 1885. The correct record was the third record retrieved.

New search algorithm

The following query was used:

```
{FirstName Lance} AND {FirstName Lindenberg} AND {Surname
Tomlinson} AND {(Spouse) FirstName Sarah} AND {(Spouse) FirstName
Johanna} AND {(Spouse) Surname Uys} AND {(Child) FirstName
Francina} AND {RelationshipDate 1900} AND {(Spouse) BirthDate
1976.1.7} AND {(Spouse) DeathDate 1918} AND {BirthDate 1884.10}
```

The following results were obtained for the different user settings:

	3C	**3F**	**3WF**	**3WC**	**1WF**	**1WC**
Position	1	1	3	3	1	2

The two queries not using weighted query vectors both listed the correct record first. The weighted query vectors listed the record third, but changing the p value to 1 made the weighted queries list the record as first and second for Fast and Complete Matching respectively.

8.3.2 Case Study 2

Known Information

The person to be located is **Sarah Crous**, born in 1848 and married to **Ephraim Ferreira**. Her father was **Pieter Arnoldus Crous** and her mother was **Johanna Aletta**

van Rooyen.

Original search algorithm

The first query used was `Crous, Sarah 1848`. The correct record was not found in the list of records. The date was then left out of the query and the correct record, (`Sarah Elizabeth Crouse 1840`) was the second item in the list. The reason why the first query did not find the correct person is that the birth date specified was later than the birth date in the record.

New search algorithm

The following query was used:

```
{FirstName Sarah} AND {Surname Crous} AND {(Spouse) FirstName
Ephraim} AND {(Spouse) Surname Ferreira} AND {BirthDate 1848}
```

The following results were obtained for the different user settings:

	3C	**3F**	**3WF**	**3WC**
Position	1	1	1	1

The record was listed first for all the different user settings. The person's parents do not occur in the database. If that information was included in the query, the record would have been the 9th record listed.

8.3.3 Case Study 3

Known Information

The person to located is **Henriëtta de Kock** who was married to **Matthys Johannes Somerset** and was the widow of **Matthee**.

Original search algorithm

The query used was `Kock, Henrietta` and the correct record was returned second.

New search algorithm

The second husband of the person is not in the database, but since this was not known to the user before the person was located, the information was included in the query:

```
{FirstName Henrietta} AND {Surname Kock} AND {(Spouse) FirstName
Mattys} AND {(Spouse) FirstName Johannes} AND {(Spouse) Surname
Somerset} AND {(Spouse) Surname Matthee}
```

The following results were obtained for the different user settings:

	3C	**3F**	**3WF**	**3WC**
Position	1	1	3	3

The correct record was the first returned record for un-weighted query vectors and third for weighted query vectors. Varying the p value had no positive effect on where the record as ranked.

8.3.4 Case Study 4

Known Information

The person to be found is **Johanna Maria De Kock**, married to **Johan Godried Bam** who was born on 1802.11.7.

Original search algorithm

The following query was used: `De Kock, Johanna Maria`. The correct person was the seventh record found.

New search algorithm

The query used for the new search algorithm was:

```
{FirstName Johanna} AND {FirstName Maria} AND {Surname Kock} AND
{(Spouse) FirstName Johan} AND {(Spouse) FirstName Godfried} AND
{(Spouse) Surname Bam} AND {(Spouse) BirthDate 1802.11.7}
```

The following results were obtained for the different user settings:

	3C	3F	3WF	3WC
Position	1	1	1	1

The correct record was listed first for all the different user settings.

8.3.5 Case Study 5

Known Information

The person to be found is **Jan Hendrik Badenhorst**, son of **Bernardus Gerhardus Badenhorst**, born 1811.3.17 and **Susara Susanna Magdalena de Kock**. **Jan Hendrik Badenhorst** was married in 1851 to **Hendrina Wilhelmina Swart**, born 1833.1.15, who should also be located in the search.

Original search algorithm

The query used to search for **Jan Hendrik Badenhorst** was `Badenhorst, Jan Hendrik`. Thirteen incorrect records were retrieved before the correct record was found. To locate **Hendrina Wilhelmina Swart**, the query `Swart, Hendrina Wilhelmina 1833.1.15` was used. The correct record was the first record retrieved.

New search algorithm

The marriage between **Jan Hendrik Badenhorst** and **Hendrina Wilhelmina Swart** is not recorded in the GDB, but the information was nonetheless included in the query:

```
{FirstName Jan} AND {FirstName Hendrik} AND {Surname Badenhorst}
AND {(Mother) FirstName Susara} AND {(Mother) FirstName Susanna}
AND {(Mother) FirstName Magdalena} AND {(Mother) Surname Kock} AND
{(Father) FirstName Bernardus} AND {(Father) FirstName Gerhardus}
AND {(Father) Surname Badenhorst} AND {(Spouse) FirstName
Hendrina} AND {(Spouse) FirstName Wilhelmina} AND {(Spouse)
Surname Swart} AND {RelationshipDate 1851} AND {(Spouse) BirthDate
1833.1.15} AND {(Father) BirthDate 1811.3.17}
```

The following results were obtained for the different user settings:

	3C	3F	3WF	3WC	1WF
Position	1	1	2	1	1

Despite the misleading information, the record was the first retrieved for all user settings except for weighted query vectors using the Fast Matching algorithm. When the p value was dropped to 1, the search with weighted query vectors using the Fast Matching algorithm also returned the correct record first.

Knowing that the marriage is not recorded in the GDB, it was not included in the query when searching for **Hendrina Wilhelmina Swart**:

```
{FirstName Hendrina} AND {FirstName Wilhelmina} AND {Surname
Swart} AND {BirthDate 1833.1.15}
```

The correct record was returned first for all the user settings.

8.3.6 Case Study 6

Known Information

The person to be found is **Jan Crous** who was married to **Maria de Kock**. This scenario is complicated by the fact that both the first names do not match the first names of the correct individuals in the GDB (The actual records was **Johannes Hendrikus Gerhardus Samuel Crouse** and **Catharina Petronella Wilhelmina De Kock**).

Original search algorithm

The query used was: `Crous, Jan`. The correct record was located only after 16 incorrect records were retrieved.

New search algorithm

The following query was used:

```
{FirstName Jan} AND {Surname Crous} AND {(Spouse) FirstName Maria}
AND {(Spouse) Surname Kock}
```

The following results was obtained for the different user settings:

	3C	**3F**	**3WF**	**3WC**	**20F**
Position	10	2	22	1	1

The correct record was located first by the query using weighted query vectors and Complete Matching. It was located second by the query using only Fast Matching. This position was moved to 1 by changing the p value to 20. No significant improvements were achieved for the other queries by changing the p value.

8.3.7 Case Study 7

The search presented here can not be performed using the original search algorithm, since the maiden name of the individual was not known . Only results for the new search algorithm will be presented.

Known Information

The person to be located is **Eunice Swart**, where **Swart** is her surname after marriage.

Results

The query used was:

```
{FirstName Eunice} AND {(Spouse) Surname Swart}
```

The correct record was that of **Eunice Susanna Saayman**, and was located first for all the possible user settings. This is a significant result as it clearly illustrates not only the effectiveness of the new search algorithm, but also its capacity to perform searches that were previously impossible.

8.3.8 Case Study 8

This scenario is another example of a search not possible using the original search algorithm.

Known Information

The person to be located is **Engela Catherina Christina** who was married to a **Rossouw**. Her maiden name was unknown.

Results

The query used was:

```
{FirstName Engela} AND {FirstName Catherina} AND {FirstName
Christina} AND {(Spouse) Surname Rossouw}
```

The correct record was that of **Engela Christina Catharina Van Graan**, and was located first for all the possible user settings.

8.3.9 Case Study 9

The original search algorithm could also not be used for this scenario.

Known Information

The person to be located is **Aletta Gertruida Susanna** who was married to a **Van der Merwe**. Her maiden name was unknown. She was born on 1891.9.24 and died on 1950.5.12 in the town *Edenburg*.

Results

The query used was:

```
{FirstName Aletta} AND {FirstName Gertruida} AND {FirstName
Susanna} AND {DeathPlace Edenburg} AND {(Spouse) Surname Merwe}
AND {BirthDate 1891} AND {DeathDate 1950}
```

The correct record was that of **Aletta Gertruida Susanna Cloete**, with a recorded birthdate of 1892, and was located first for all the possible user settings.

8.3.10 Case Study 10

This is an artificial scenario, designed to illustrate the power of the new search algorithm. A search will be done for **Luine Stefanie Stapelberg**, but only minimal information will be used.

Known Information

Assume that only the following information is known: The individual's father was born in **Barkley-Oos**, her mother maiden surname is **Seeber**, and she has a child who is named **Mathys**.

Results

The query used was:

```
{(Mother) Surname Seeber} AND {(Father) BirthPlace Barkley-Oos}
AND {(Child) FirstName Mathys}
```

The correct record was located first for all the possible user settings. This scenario shows that the correct individuals can be located using only the bare minimum of information.

8.3.11 Case Study 11

In this second artificial scenario, a search for **Gideon de Villiers de Kock** will be performed.

Known Information

Assume that only the following information is known: We know that the person to be searched for has the first name **Deon** and that he married to a lady named **Magdalena** in **Pretoria-Oos**.

Results

The query used was:

```
{FirstName Deon} AND {(Spouse) FirstName Magdalena} AND
{RelationshipPlace Pretoria-Oos}
```

The correct record was located first for all the possible user settings.

8.4 Conclusions

The case studies presented in this chapter clearly show the versatility and effectiveness of the new search algorithm. In each case the individual searched for was located and found

with fewer incorrect records than the original search algorithm. Some searches that were not possible using the original search algorithm were performed successfully with the new algorithm.

Despite the heuristics laid down by Salton and Buckley [SB88], it was found that unweighted query vectors outperforms weighted query vectors. This is mainly because each term used to search the GDB can be seen as equally important.

In general, results using the Complete Matching algorithm was superior to the Fast Matching algorithm.

Chapter 9

CONCLUSIONS

9.1 Summary of Research

This project consisted of a literature study of Information Retrieval models. Several models were investigated, and both the most novel and commonly used were reported. For genealogical information, it was concluded that the Extended Boolean retrieval model would be best suited. Structured text retrieval models were considered with the idea of hybridization with other retrieval models.

Several algorithms proposed specifically for the GIS were investigated and adapted. A new retrieval model, that draws ideas from the Extended Boolean model, Structured Text models and the algorithms specifically designed for the GIS, was proposed.

Several implementation techniques were investigated, and the lexicon to be used by the retrieval algorithm refined. The efficient implementation of the new retrieval algorithm was described in detail.

The new retrieval algorithm was evaluated with respect to search times and retrieval effectiveness. It was compared to the original search algorithm where possible.

9.2 Future Research

The focus of this research was the development of the search algorithm. Several factors were not considered, and could be topics of future research.

The relevance values between different fields, that are used by the relevance function, can only be shown to be correct through many searches. The refinement of these relevance values could present opportunities for further research.

No compression was done on the posting linked lists that are stored for each term. By using an efficient compression algorithm the disk space used could be significantly reduced. Perhaps more importantly, if less disk space were to be used, it would require fewer disk reads to retrieve information and would thus result in faster retrieval times.

9.3 Conclusions

This research cumulated in the design and implementation of a fast, effective search algorithm. The algorithm allows searches for words and dates, and takes into account the structure of the query and records. Functionality is provided that allows the user to search on information about an individual's relations (parents, children, etc.). Search results are ranked in order of most relevant to the user's query.

The algorithm can be applied to any database that stores textual information on individuals and their relations.

Appendix A

Data Structures

```
AltparTipe = Record
             Altfat : IDTipe;
             Altmot : IDTipe;
             ATipe : str4;  {0 Adopted, 1 Foster Parents}
             end;

EventTipe = Record
            ETipe      : str4;
            StartDat, EndDat : DatumTipe;
            StartDatC, EndDatC : str4;
            EPlekC : str4;
            EPlek : PlekTipe;
            EInfo  : String;
            EVoetNota : String;
            References : String;
            end;

MInfoRekordTipe  = Record
             Inf : String;
             IVoetNota : String;
             end;

EventArray = Array[1..MaxEvents] of EventTipe;
ReferenceArray = Array[1..MMaxVerwys] of VerwysTipe;
AlternateParentArray = Array[1..MaxAltPar] of AltparTipe;

MensRekord = record
(*------*)   Id,PaId,MaId      : IDTipe    ;

             Geslag            : str4    ;  {0 male, 1 female, 2 unknown}
             GebDat,
```

```
              SterfDat          : DatumTipe;
              Van               : String[VanLen];
              VNaam             : String[VoornaamLen];
              BNaam             : String[Bynaamlen];
              AltParent         : AlternateParentArray;
              Events            : EventArray;
              MVerwys           : ReferenceArray;
              MInfoRek          : MInfoRekordTipe;
              MRecordHis        : Array[1..MaxRecordHis] of RecordHistType;
              NrAltPar, NrEvents, NrMVerwys  : integer;
end;

HInfoRekordTipe  = Record
              Inf : String;
              HVoetNota : String;
              end;

ReferenceArray = Array[1..HMaxVerwys] of VerwysTipe;

HuwRekord  = record
(*-----*)     ManId,VrouId      : IDtipe;
              HuwDat,SkeiDat    : Datumtipe;
              HuwDatC, SkeiDatC : str4;
              HPlekC            : str4;
              HPlek             : PlekTipe;
              SkeiTipe,         (* 0 Marriage, 1 Common Law*)
              TrouTipe          : str4;  (*0 Divorce, 1 Annulment, 2 Seperation*)
              HVerwys           : ReferenceArray;
              HInfoRek          : HInfoRekordTipe;
              HRecordHis        : Array[1..MaxRecordHis] of RecordHistType;
              NrHVerwys         : integer;
              end;
```

Appendix B

EBNF for Query Language

The query language for the GIS is given below. **Value** means a name, term or date that a user would like to search for.

Query	::= *Disjunction*
Disjunction	::= *Conjunction* {`OR` *Conjunction*}
Conjunction	::= *Term* {`AND` *Term*}
Term	::= {*Individual Field* **Value**}
Individual	::= `(Father)` | `(Mother)` | `(Person)` | `(Spouse)` | `(Child)` | λ
Field	::= `firstname` | `surname` | `nickname` | `place` | `event` | `birth`
	| `birthplace` | `birthdetails` | `baptism` | `baptismplace` | ...
	| `eventdate` | `birthdate` | ...

The rest of the possible search fields can be seen in appendix C.

Appendix C

Search Fields

All fields that can be searched for are listed in tables C.1 and C.2.

Person	eventdate	birthdate
OR Mother		baptismdate
OR Father		deathdate
OR Child		deathdate
OR Spouse		burialdate
		immigrationdate
		residencedate
		willdate
		adoptiondate
		militarydate
		medicaldate
		educationdate
		occupationdate
		otherdate
		relationshipdate

Table C.1: Available date search fields

Person	name	firstname	
OR Mother		surname	
OR Father		nickname	
OR Child	event	birth	birthplace
OR Spouse			birthdetails
		baptism	baptismplace
			baptismdetails
		death	deathplace
			deathdetails
		burial	burialplace
			burialdetails
		immigration	immigrationplace
			immigrationdetails
		residence	residenceplace
			residencedetails
		will	willplace
			willdetails
		adoption	adoptionplace
			adoptiondetails
		military	militaryplace
			militarydetails
		medical	medicaldetails
			medicalplace
		education	educationplace
			educationdetails
		occupation	occupationplace
			occupationdetails
		other	otherplace
			otherdetails
		relationship	relationshipplace
			relationshipdetails
		place	baptismplace
			birthplace
			deathplace
			burialplace
			immigrationplace
			residenceplace
			willplace
			adoptionplace
			militaryplace
			educationplace
			occupationplace
			otherplace
			relationshipplace
	information		

Table C.2: Available term search fields

Appendix D

Record Fields

Provision is made for terms appearing in 64 different field. Each field has a 64-bit integer associated with it. The code for each field corresponds to one bit of the bit-string representing the integer being set to 1.

```
First Name          : 1;                    //1
Surname             : 2;                    //2
Nickname            : 4;                    //3
Information         : 8;                    //4
Relationship Type   : 16;                   //5
Relationship Place  : 32;                   //6
Relationship Info   : 64;                   //7
Separation Type     : 128;                  //8
Birth Place         : 512;                  //10
Birth Details       : 1024;                 //11
Baptism Place       : 2048;                 //12
Baptism Details     : 4096;                 //13
Death Place         : 8192;                 //14
Death Details       : 16384;                //15
Burial Place        : 32768;                //16
Burial Details      : 65536;                //17
Immigration Place   : 131072;               //18
Immigration Details : 262144;               //19
Residence Place     : 524288;               //20
Residence Details   : 1048576;              //21
Will Place          : 2097152;              //22
Will Details        : 4194304;              //23
Adoption Place      : 8388608;              //24
Adoption Details    : 16777216;             //25
Military Place      : 33554432;             //26
Military Details    : 67108864;             //27
```

```
Medical Place       : 134217728;                //28
Medical Details     : 268435456;                //29
Education Place     : 536870912;                //30
Education Details   : 1073741824;               //31
Occupation Place    : 2147483648;               //32
Occupation Details  : 4294967296;               //33
Other Place         : 8589934592;               //34
Other Details       : 17179869184;              //35
Event 13 Place      : 34359738368;              //36
Event 13 Details    : 68719476736;              //37
Event 14 Place      : 137438953472;             //38
Event 14 Details    : 274877906944;             //39
Event 15 Place      : 549755813888;             //40
Event 15 Details    : 1099511627776;            //41
Event 16 Place      : 2199023255552;            //42
Event 16 Details    : 4398046511104;            //43
Event 17 Place      : 8796093022208;            //44
Event 17 Details    : 17592186044416;           //45
Event 18 Place      : 35184372088832;           //46
Event 18 Details    : 70368744177664;           //47
Event 19 Place      : 140737488355328;          //48
Event 19 Details    : 281474976710656;          //49
Event 20 Place      : 562949953421312;          //50
Event 20 Details    : 1125899906842624;         //51
Event 21 Place      : 2251799813685248;         //52
Event 21 Details    : 4503599627370496;         //53
Event 22 Place      : 9007199254740992;         //54
Event 22 Details    : 18014398509481984;        //55
Event 23 Place      : 36028797018963968;        //56
Event 23 Details    : 72057594037927936;        //57
Event 24 Place      : 144115188075855872;       //58
Event 24 Details    : 288230376151711744;       //59
Event 25 Place      : 576460752303423488;       //60
Event 25 Details    : 1152921504606846976;      //61
Event 26 Place      : 2305843009213693952;      //62
Event 26 Details    : 4611686018427387904;      //63
```

Appendix E

Date Fields

```
Birth Date        :  1;
Baptism Date      :  2;
Death Date        :  4;
Burial Date       :  8;
Immigration Date  :  16;
Residence Date    :  32;
Will Date         :  64;
Adoption Date     :  128;
Military Date     :  256;
Medical Date      :  512;
Education Date    :  1024;
Occupation Date   :  2048;
Other Date        :  4096;
Event 13 Date     :  8192;
Event 14 Date     :  16384;
Event 15 Date     :  32768;
Event 16 Date     :  65536;
Event 17 Date     :  131072;
Event 18 Date     :  262144;
Event 19 Date     :  524288;
Event 20 Date     :  1048576;
Event 21 Date     :  2097152;
Event 22 Date     :  4194304;
Event 23 Date     :  8388608;
Event 24 Date     :  16777216;
Event 25 Date     :  33554432;
Event 26 Date     :  67108864;
Relationship Date :  134217728;
```

Appendix F

Relevance Function

The relevance function $\mathcal{D}$ is implemented using the following algorithm that calculates the distance between fields `Required` and `Located`. Note that these fields are integers encoded as described in sections 7.3.3 and 7.3.2. In the algorithm `AND` and `OR` is the bit-wise logical AND and OR of the integers.

```
function Relevance(Required, Located) begin
    RELEVANCE = 0
    if (Required AND Located) > 0 then
    begin
        RELEVANCE = 1
    end
    else begin
        for each non-zero bit position, RP, in Required
        begin
            for each non-zero bit position, LP, in Located
            begin
                RELEVANCE = Max(Table[RP][LP], RELEVANCE)
            end
        end
    end

    return RELEVANCE
end
```

In other words, if a term is located in the required field (or fields), the relevance is 1. Otherwise, the relevance is the maximum relevance between all the fields where the term was located and required.

Appendix G

Equivalent and Similarity Database Indexing

The equivalent and similarity database is created by sequentially processing the entire GDB. The following algorithm is used:

```
COUNT = 0 For each person record in GDB: begin
    Read person record
    For each term in the record:
        if Term is not in STI then
        begin
            Create a TTermRecord
            Assign Term to current term
            Write record to disk
            Create STI entry pointing to record
            Increment COUNT
        end
        else begin
            Use STI to locate record
            Increment TotalFrequency
            Increment other relevant frequency fields
        end
end

STEP = MAX_CODE_SIZE / (COUNT+1) TEMP = STEP For each term in STI
(traversed alphabetically): begin
    Read term record
    Assign Code to TEMP
    TEMP = TEMP + STEP
    Save changes to record
```

```
    Create SCI entry, using Code, pointing to the record
end
```

The above code is run only once. Note that the codes conform to the lexicographical order of the terms. A gap is left between codes so that new terms can be added to the database without assigning a new code to all terms.

Appendix H

Term and Date Indexing

The following algorithm is used to create the term and the date index:

```
For each person record in GDB begin
    Read record
    For each field in record
    begin
        For each term in field
        begin
            Locate term in STI
            Read TTermRecord
            With EquivalentCode and each SimilarCode do
            begin
                if Code is in TI then
                begin
                    Locate TPosting linked list
                    if there exist a TPosting with
                            (RecordID = record number) then
                    begin
                        Read TPosting record
                        Occurrences = Occurrences OR field code
                        Increment Frequency
                        Save changes
                    end
                    else begin
                        Create TPosting record
                        Assign RecordID to person record number
                        Assign Occurrences to field code
                        Assign Frequency to 1
                        Insert TPosting into linked list
                    end
                end
```

```
                else begin
                    Create TPosting record
                    Assign RecordID to person record number
                    Assign Occurrences to field code
                    Assign Frequency to 1
                    Store record
                    Insert Code into TI pointing to TPosting record
                end
            end
        end
        For each date in field
        begin
            if record number is in DI then
            begin
                Locate TDatePosting linked list
                Create TDatePosting record
                Assign Field to field
                Assign Startdate to the start date
                Assign Enddate to the end date
                Insert TDatePosting in linked list
            end
            else begin
                Create TDatePosting record
                Assign Field to field
                Assign Startdate to the start date
                Assign Enddate to the end date
                Store TDatePosting record
                Insert record number into DI
                    pointing to TDatePosting record
            end
        end
    end
    Repeat above process with person's mother, father,
        spouses and children using appropriate indexes
end
```

The above algorithm will create the term and date search indexes as described in sections 7.3.2 and 7.3.3.

Appendix I

Retrieval and Ranking algorithm

The following algorithm is used to retrieve and rank records in response to query Q.

```
For each term, ki, in Q begin
    if ki is a word and not a period
    begin
        Locate ki in STI
        Read TTermRecord
        Replace ki by EquivalentCode
    end
end For each conjunction of terms, QCT begin
    For each Code in QCT
    begin
        Locate Code in TI
        Retrieve TPosting linked list, calculate
            weight for each term and store in memory
            in a new linked list
    end


    Perform union merge on linked list and evaluate

      equation 4.1 for each person record

    For each period in QCT
    begin
        For each record number in linked list
        begin
            Locate record number in DI
            Traverse TDateposting linked list and
                calculate weight for date term and use

                equation 4.1 to determine a
```

```
                new ranking for each record
                in the linked list
        end
    end
end For the disjunction of terms, QDT begin
    For each Code in QDT
    begin
        Locate Code in TI
        Retrieve TPosting linked list, calculate
            weight for each term and store in memory
            in a new linked list
    end

    Perform union merge on linked list and evaluate

      equation 4.2 for each person record

    For each period in QCT
    begin
        For each record number in linked list
        begin
            Locate record number in DI
            Traverse TDateposting linked list and
                calculate weight for date term and use

                equation 4.2 to determine a

                new ranking for each record
                in the linked list
        end
    end
end

Perform union merge on linked lists from conjuctions

  and disjunctions and evaluate equation 4.2

  for each person record

Retrieve 10 highest ranking records and display to the user
```

Bibliography

[BP98] Sergei Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30:107–117, April 1998.

[Bur92] Forbes J. Burkowski. Retrieval activities in a database consisting of heterogeneous collections of structured text. In *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 112–125. ACM Press, 1992.

[BYN96] Ricardo Baeza-Yates and Gonzalo Navarro. Integrating contents and structure in text retrieval. *ACM SIGMOD Record*, 25(1):67–79, 1996.

[Chi96] C. Chinner. Transformation algorithms to determine the similarity between names. Master's dissertation not completed, Department of Computer Science and Information Systems, U.P.E., Port Elizabeth, 1996. Supervisor : G. de V. de Kock.

[CKR01] J. Choi, M. Kim, and V. V. Raghavan. Adaptive feedback methods in a extended boolean model. *ACM SIGIR Workshop on Mathematical/Formal Methods in Information Retrieval*, 7 - 12 September 2001.

[DK88] G. de V. De Kock. Measuring the success of name matching algorithms in a genealogical database (*die meting van sukses van naampassingsalgoritmes in 'n genealogiese databasis). Q.I.*, 6(3):119 – 122, 1988.

[DK02] G. de V. De Kock. *Proceedings SAICSIT 2002 : Enablement through Technology, Annual Research Conference of SAICSIT : Searching on Full Name*

Providing for Spelling Variations, chapter 2, page 255. ACM International Conference Proceedings. SAICSIT (SA Institute of Computer Scientists and Information Technologists), Boardwalk, Port Elizabeth, 16 to 18 September 2002. Abstract.

[DK04] G. de V. De Kock. A search algorithm for an archive database using person and place names. In *Proceedings of the 2nd International Conference on Cumputer Science and its Applications*, pages 216–223. US Education Service, LCC, 2004.

[DKDP93] G. de V. De Kock and Charmaine Du Plessis. The empiracal evaluation of some word matching algorithms to determine eqivalent surnames in a genealocial database (*die empiriese evaluering van enkele variasies van 'n woordpassingalgoritme vir die bepaling van ekwivalente vanne in 'n genealogiese databasis*). *S.A. Computer Journal*, 10:48 – 53, September 1993.

[dT03] S.F. du Toit. A search module based on the similarity name database for the wingis system. Honours project, Department of Computer Science and Information Systems, U.P.E., Port Elizabeth, 2003. Supervisors : M.C. du Plessis, G. de V. de Kock.

[Dur94] J. Durkin. *Expert Systems, Design and Development.* Macmillan Publishing Company, 1st edition, 1994.

[FMC98] R. Veroff F. M. Carrano, P. Helman. *Data Abstraction and Problem Solving with C++.* Addison-Wesley, 2nd edition, 1998.

[Fra92] W. B. Frakes. Stemming algorithms. *Information retrieval: data structures and algorithms*, pages 131–160, 1992.

[Goo04] Google. Google search engine. *www.google.com*, 2004.

[GT02] M. T. Goodrich and R. Tammasia. *Algorithm Design: Foundations, Analysis, and Internet Examples.* John Wiley and Sons, 1st edition, 2002.

[Hal80] P. A. V. Hall. Approximate string matching. *Computing Surveys*, 12(4):381 – 402, 1980.

[HK01] F. M. Ham and I. Kostanic. *Principles of Neurocomputing for Science and Engineering*. McGraw-Hill, international edition, 2001.

[KM93] Pekka Kilpelinen and Heikki Mannila. Retrieval from hierarchical texts by partial patterns. In *Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 214–222. ACM Press, 1993.

[Knu73] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[LW75] R. L. Lowrance and R. A. Wagner. An extension of the string-to-string correcting problem. *J. ACM*, 22(2):177 – 183, 1975.

[MJM83] G. Salton M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1st edition, 1983.

[NBY95] Gonzalo Navarro and Ricardo Baeza-Yates. A language for queries on structure and contents of textual databases. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 93–101. ACM Press, 1995.

[NBY97] Gonzalo Navarro and Ricardo Baeza-Yates. Proximal nodes: a model to query document databases by content and structure. *ACM Transactions on Information Systems (TOIS)*, 15(4):400–435, 1997.

[Nic90] W. Keith Nicholson. *Linear Algebra with Applications*. PWS Publishing Company, third edition, 1990.

[Ore02] Nir Oren. Reexamining tf.idf based information retrieval with genetic programming. In *Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 224–234. South African Institute for Computer Scientists and Information Technologists, 2002.

[Ple91] Charmain Du Plessis. *Woordpassingsalgoritmes vir die bepaling van gelyksoortige vanne in 'n Genealogiese Inligtingstelsel*. Masters dissertation, University of Port Elizabeth, Port Elizabeth, Febuary 1991.

[Ple01] M.C. Du Plessis. *Genealogical Information System in Windows*. Honours project, University of Port Elizabeth, Port Elizabeth, December 2001. Supervisor : G. de V. de Kock.

[Por97] M.F. Porter. An algorithm for suffix stripping. *Readings in Information Retrieval*, pages 313–316, 1997.

[RBJ89] V. V. Raghavan, P. Bollmann, and G. S. Jung. Retrieval system evaluation using recall and precision: problems and answers. In *Proceedings of the 12th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 59–68. ACM Press, 1989.

[RBY99] B. Ribeiro-Neto R. Baetza-Yates. *Modern Information Retrieval*. Addison-Wesley, 1st edition, 1999.

[RJ88] Stephen E. Robertson and Karen Sparck Jones. Relevance weighting of search terms. *Document retrieval systems*, pages 143–160, 1988.

[SB88] G Salton and C Buckley. Term-weighting approaches in automatic text retrieval. *Information processing and management*, 24(5):513 – 523, 1988.

[SFW83] Gerard Salton, Edward A. Fox, and Harry Wu. Extended boolean information retrieval. *Communications of the ACM*, 26(11):1022–1036, 1983.

[SL68] G. Salton and M. E. Lesk. Computer evaluation of indexing and text processing. *Journal of the ACM (JACM)*, 15(1):8–36, 1968.

[TC90] H. Turtle and W. B. Croft. Inference networks for document retrieval. In *Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 1–24. ACM Press, 1990.

[TCB99] I. H. Witten T. C. Bell, A. Moffat. *Managing Gigabytes, Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Inc., 2nd edition, 1999.

[vH04] Marena van Hemert. *Families Registers, Old Swellendam Families*, volume 2. M. van Hemert and R.L. Aspeling, 1st edition, 2004.

[VR89] P. Bollman V.V. Raghavan, G.S. Jung. A critical investigation of recall and precision as measures of retrieval system performance. *ACM Transactions on Information Systems (TOIS)*, 7(3):205–229, July 1989.

[WF74] R. A. Wagner and M. J. Fischer. The string-to-string correcting problem. *J. ACM*, 21(1):168 – 173, 1974.

[WH91] Ross Wilkinson and Philip Hingston. Using the cosine measure in a neural network for document retrieval. In *Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 202–210. ACM Press, 1991.

[XC98] Jinxi Xu and W. Bruce Croft. Corpus-based stemming using cooccurrence of word variants. *ACM Trans. Inf. Syst.*, 16(1):61–81, 1998.

[YA94] Tak W. Yan and Jurgen Annevenlink. Integrating a structured-text retrieval system with an object-orientated database system. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 740–749, 1994.

[Yah04] Yahoo. Yahoo search engine. *www.yahoo.com*, 2004.