A Word Game Support Tool Case Study

T Botha, D G Kourie, B W Watson

Fastar / Espresso Research Group, Department of Computer Science, University of Pretoria, Pretoria, South Africa 0001

ABSTRACT

This article reports on the approach taken, experience gathered, and results found in building a tool to support the derivation of solutions to a particular kind of word game. This required that techniques had to be derived for simple yet acceptably quick access to a dictionary of natural language words (in the present case, Afrikaans). The main challenge was to access a large corpus of natural language words via a partial match retrieval technique. Other challenges included discovering how to represent such a dictionary in a "semi-compressed" format, thus arriving at a balance that favours search speed but nevertheless derives a savings on storage requirements. In addition, a query language had to be developed that would effectively exploit this access method. The system is designed to support a more intelligent query capability in the future. Acceptable response times were achieved even though an interpretive scripting language, ObjectREXX, was used.

KEYWORDS: support tool, case study, natural language dictionary access, word puzzle language, inverted file approach, query language, search techniques, dictionary corpus, Afrikaans language

1 INTRODUCTION

"... you shall seek all day ere you find them; and, when you have found them, they are not worth the search." [Merchant of Venice]

One is often surprised at how complex and challenging certain word puzzles / games are that are regularly—daily or weekly—being solved by thousands of readers of magazines and newspapers. The compilers of these puzzles make a host of assumptions that are presumed to be "common knowledge" based on their own thorough background knowledge of the relevant natural language. These assumptions relate to matters such as language knowledge, word construction, word usage, synonyms, general current affairs knowledge, etc.

Along with these assumptions, the typical word puzzle involves a large body of possible candidate solutions that can fit the particular sub-entry of the puzzle. The overall solution is thus to solve a pattern matching problem of natural language words that fit the puzzle in a particular combination. It is therefore not surprising that a variety of support tools are available for puzzle solver enthusiasts. Most of these support tools are available in book format such as typical dictionaries or ordered word lists based on criteria that are unique to the relevant game or word problem. Examples include:

- Alphabetic ordering according to word length
- Word types used for clustering and ordering—such as clusters of geographic or person names, Greek alphabet naming, etc.
- Limited synonyms and word explanations added to the relevant word
- Various forms of thesauri adapted to the peculiarities of the type of word puzzle or challenge (See for example [1])

The support tools are generally employed by the users to augment their own knowledge. Here the classical combination in information retrieval comes to mind: the user's involvement along with the support tool's capabilities form a powerful team in solving a complex search problem. (Note that this is a classical example of user relevance feedback in information retrieval [2, 3].)

This research is inspired by the fact that one of the authors, being a willing and regular victim of one such challenge, decided to seek more powerful support in solving a particular word puzzle challenge. This has resulted in an elegant solution that has been embedded into a software support tool.

The remainder of this article is laid out as follows. In section 2, the background to this problem is discussed in terms of the problem statement and a brief consideration of the range of potentially applicable search types. Section 3 explains the word list database, as well as the use and storage of letter groups. It also introduces a novel second-level indexing technique that was used. Sections 4 is devoted to an overview explanation of query language matters: the language's syntactical design; techniques used in building the query language engine; the steps followed in deriving a list of candidate answers; and the need for filtering these answers. In section 5, the focus is on the so-called word puzzle language, which is derived

Email: T Botha `acbotha@iafrica.com`, D G Kourie `dkourie@cs.up.ac.za`, B W Watson `watson@cs.up.ac.za`

from the natural language used in order to facilitate queries. The conclusion of the article, in section 6, includes an assessment of future work that is needed on the support tool.

2 BACKGROUND

2.1 Problem Statement

The following is a short description of "letterspeletjie"—a word game that appears in Saturday editions of the Johannesburg-based Afrikaans newspaper, Beeld, and that is the target of the support tool that has been built. The player is presented with a number of cells. Each cell in the puzzle contains a number (between 1 and 26) that represents a letter. The only clue given, is that a number of cells (usually 2) are pre-filled with letters. The challenge is to establish which letter is represented by which number. Fortunately, the answer is always a one-to-one mapping. Some imbedded "clues" are present such as the particular word length allowed for an accompanying word. Short words—2 or 3 characters long— are the best candidates with which to start and to use for an initial trial-and-error approach.

The challenge to be met is to build a tool that supports the quest for a solution to such a problem—a problem which may be described roughly as a kind of constraint satisfaction problem. More specifically, a solution to this problem involves, at a minimum, a partial match search and retrieval. Because of its complexity, a number of prerequisites need to be met by a tool that handles this type of problem. The main ones are listed below.

The first practical challenge to overcome relates to dealing with the large volume of words that are potential candidates for the "knowledge base" of the tool. In the example case that was used to implement the support tool, an Afrikaans language dictionary of more than 132000 words was constructed and used. This is in line with various sources that suggest that more than 130.000 and even as much as 180.000 words can be involved in Afrikaans word game puzzles[1]. In order to set up the dictionary, the following questions needed to be answered:

- How can the dictionary be represented to support acceptably fast searching?
- How can the storage volume of the dictionary be limited to acceptable levels?
- What search techniques that encompass strict partial match search / retrieval should be used?

In addition, a relative simple but powerful query language was needed to deliver solutions for the search / retrieval goals. The language had to allow for direct interactive personal usage, as well as for programmatic access. Also needed was a set of support and build routines to maintain the words and knowledge base. Finally, a rule recogniser was needed to augment the dictionary's word knowledge with relevant domain knowledge. In this study this was limited to (embryonic) rules defined for the "letterspeletjie" word puzzle.

As a practical demonstration of the techniques used and proposed, a support tool was built that significantly supports the user in overcoming many of the challenges in a game that is as complex as the named "letterspeletjie" word puzzle. It does so by exploiting the user's "common knowledge" so that the mutual effort of user and system lead to suggested solutions.

2.2 Types of Search

One can use various ways to approach the retrieval of information from a set of information elements. Essentially these approaches are differentiated by their generality. The following categorisation of these so-called "intersection searches" [4] are widely quoted in the literature. (See for example [5, 6, 7, 8, 9, 10].) In summary these types of searches are as follows:

1. *Exact matching queries:* This type of search assumes that a single answer exists for the query. The answer may be a record, a document, or general knowledge-rich answer. This type of query simply asks whether the answer is present in the relevant database or document base or knowledge base.
2. *Single key queries:* Here it is expected that all answers match a particular value of a single attribute in the query. For example, a search will yield all records that match a particular field such as, say a given date.
3. *Partial match queries:* Partial match queries retrieve answers where only some of the attributes are known and the rest are accepted as generally true. This type of query is relevant for this study. For example, a word puzzle challenge may be to search for words matching the query: "B?T??R" where the "?" character represents a single character "wild card". Answers may include words such as "BATTER", "BETTER", "BITTER", "BUTLER", and "BUTTER".
4. *Range searching:* These search types are similar to partial match retrieval except that the request allows the answer to have a range of values for the specified attributes.
5. *Best match with restricted distance:* In this case, one of the many different distance functions that have been researched, is used to measure whether the answer qualifies or not.
6. *Boolean queries:* In this type of search, Boolean functions are defined and applied to the attributes.

While studying the particular challenge in question, it became obvious that the problem is best addressed by generally regarding it as a partial match retrieval problem, but by also taking into account that some of its characteristics correspond to the other

[1]The corresponding author was privileged to be a team member while working on the Afrikaans Dictionary for the IBM DisplayWrite family of word processing products in the mid-1980s. This electronic dictionary was published consisting of 186.000 words (and fits on a 1.45MB stiffy!).

types of searches mentioned above. Therefore, in some cases a hybrid approach (combining types of search) is taken by the proposed solution.

3 PROPOSED WORD REPRESENTATION

In deriving the proposed solution, the first consideration was how to represent the thousands of natural language (Afrikaans) words to allow for quick access and partial match retrieval. The challenge at hand is primarily character string based. Various studies have been conducted to tackle the partial match retrieval problem. Complicated techniques—mostly involving trees, tries, and hash indexing—have been proposed. (See for example: [11, 5, 4, 12].) Most of these techniques expose strengths in some areas and weaknesses in others. This study was aimed at a simple yet powerful approach to resolve the problem at hand. The proposed solution is based on a simple yet powerful inverted file technique. As mentioned, when combined with other retrieval types this approach offers an elegant and acceptably quick solution.

3.1 Building the Word List Database

The decision was made to use the ObjectREXX[2] scripting language for developing the example implementation. ObjectREXX (like C++) offers both traditional and object-oriented features. ObjectREXX is particularly strong in its string handling features, has powerful tracing abilities, and offers elegant streamed file handling which was very beneficial for development of the example implementation[3]. Despite its interpretive nature, ObjectREXX proved surprisingly adequate for present purposes. However, in future extensions of the tool, a compilable base language may be considered to retain acceptable performance levels.

As a starting point, the words were represented as character strings in a streamed file. This can be considered as a word list. However, some words were slightly modified, as discussed in section 5 below, yielding a list of words in what may be called the “word puzzle language”. It was decided to employ different types of word databases: a set of fixed-length databases for shorter, frequently-used words; and a set of mixed-length databases. The former set thus contained a separate database for words of length 2; another, for words of length 3; $\cdots$ and finally, another for words of length 10. These allow for focussed retrieval techniques that capitalise on the known word length. The latter set consists of a single database whose word list contains a mixture of words ranging in length from 1 through 35. Having the two types of databases offers a number of benefits for the design and implementation of the query language, as is discussed later.

Each database consists not merely of the word list file. There are two additional files: an inverted list file; and a secondary index file that references the inverted list file. The inverted list file is discussed next. In subsection 3.3, the secondary index file is briefly discussed.

[2]ObjectREXX—Object-Oriented REstructured eXtended eXecutor programming language

[3]The corresponding author also had access to an installed version of ObjectREXX for Windows

		arm	bedaar	daar	darem	warm
1	aa	.	1	1	.	.
2	ar	1	1	1	1	1
3	be	.	1	.	.	.
4	da	.	1	1	1	.
5	ed	.	1	.	.	.
6	em	.	.	.	1	.
7	re	.	.	.	1	.
8	rm	1	.	.	.	1
9	wa	.	.	.	.	1

Table 1: Matrix-based solution to a query

3.2 Letter Group Approach

A letter group (or n-gram) is simply a sequence of alphabetic characters. The following example illustrates how such letter groups can be used to facilitate the partial match queries. While the example is based on letter groups of length 2 only, it was clear at the outset of the study that the proposed application would benefit from the use of letter groups of length 3 and 4 as well.

3.2.1 Example: Matrix Representation

Consider the following list of Afrikaans words: (`arm, bedaar, daar, darem, warm`). Collectively, they incorporate the following 9 letter groups of length 2 (i.e. 2-grams): (`aa, ar, be, da, ed, em, re, rm, wa`). Suppose that each entry in the word list that stored these words, also incorporated information of the word’s 2-grams. Additionally, suppose that there was another file storing 2-gram information: for each 2-gram, it indicated the words in which the 2-gram occurred. These two files, the word list file and the letter group file, collectively constitute a database whose collective contents can be summarized in the matrix shown in table 1.

Suppose that, as part of a partial match query, a search for words containing the substring `arm` is to be made. The first step (not considered in detail here) is to determine that the query is comprised of the letter groups `ar` and `rm`. Thereafter, the matrix representation may be used. The search steps are as follows: extract the contents of the `ar` and `rm` records in the letter group file; do a logical-AND operation on these records; determine which word list entries are set as a result; extract and return these word list entries as the answer to the partial match query. These steps may be summarised as follows:

$$\begin{aligned} V &= \texttt{AND}\ \{\texttt{ar and rm}\} \\ &= \{(1,1,1,1,1)\ \texttt{AND}\ (1,0,0,0,1)\} \\ &= \{(1,0,0,0,1)\} \\ &= \{\texttt{arm,warm}\} \end{aligned}$$

Word length	2	3	4	#letter groups	#characters
2	1			1	2
3	2	1		3	7
4	3	2	1	6	16
5	4	3	2	9	25
6	5	4	3	12	34
7	6	5	4	15	43
8	7	6	5	18	53

Table 2: Number of letter groups for words of given length

3.2.2 Infeasibility of Direct Matrix Representation

However, while the logic of the above approach is simple enough, it is not feasible to implement the approach directly. Even from the small set of words in the example, it is evident that one cannot efficiently store the matrix rows and columns as records in the letter group and word list files respectively. The matrix is largely empty—a sparse matrix—and even a bit representation of each possible letter group, for example, requires excessively large storage. The extent of the problem is explained next. Recall that the full application was to rely on letter groups of length 2, 3 and 4.

$$\begin{aligned} \text{There are } 26^2 &= 676 \text{ letter pairs} \\ 26^3 &= 17576 \text{ letter groups of length 3} \\ 26^4 &= 456976 \text{ letter groups of length 4} \\ \text{Total} &= 475228 \text{ possible letter groups} \end{aligned}$$

Representing each occurrence by one bit implies 475228/8 = 59459 bytes. Thus, about 59K bytes per word is required to cater for each possible substring of length 2, 3, and 4—i.e. the column information in the above matrix would require about 59K bytes for each word.

In contrast to this, the table 2 shows the number of possible letter groups of length 2, 3 and 4 that could be derived from words of lengths 2 to 8. It also shows the total number of letter groups as well as the total number of characters that would be needed to represent each possible letter group separately. From this table it is evident that to store letter-group information for, say an 8-letter word, a maximum of 18 letter groups could be present, i.e. a maximum of only 18 bits in the 59K byte column would have to be set.

Even from early on in the study, it was clear that special attention would have to be given to techniques for storing this sparse matrix—even though it remains desirable that the searching and querying is based on a technique that is essentially the above-described matrix matching type of approach. This “matrix approach” allows elegant and recursive matching of the query’s search strings to the existing search strings of the “matrix”. Therefore the proposed access method acts in theory as a matrix matching engine but in practice, one seeks for compromises on various fronts.

3.2.3 Using an Inverted List

Each record in the word list constitutes, logically, a so-called word list group (wlg), that can be represented as:

$$wlg \;=\; (index,\ word,\ letter\ groups)$$

Thus, a wlg-entry for a given word in a word list, logically embeds a list of the letter groups that occur in the word. Note, however, that in the final word list file, only the words themselves are stored. The index and lett-groups are implied.

Related to each word list, is an inverted list[4]. Each entry in this inverted list (file) is a pair, called a letter group index (lgi). The first element of an lgi pair is a letter group of length 2, 3 or 4 characters. The second element of the pair is a list of indices into the word list, indicating words in which these letter groups occur. This latter component is referred to as an occurrence list. The inverted list file’s entries are thus represented as:

$$lgi = (letter\ group,\ occurrence\ list)$$

Recall that there are several databases, each with its own word list. The inverted list file is derived for each word list, by identifying the letter groups in each word of the word list. Each database therefore includes of a word list and associated inverted file list. (A secondary index file—discussed later—is also used, primarily for efficiency reasons.) The inverted list offers a unique short path to the word occurrences that contain relevant letter groups. Answering a query is done by matching the letter groups that are derived from the query to these word substrings.

3.2.4 Inverted List Example

Consider the previous example. The word list (indicated by {wlg}) and the inverted list (indicated by {lgi}) are now as indicated below. (Note, however, {wlg} reflects the logical contents, whereas the physical contents is limited to the word itself.)

```
{wlg} =  { (1, arm,    (ar, rm)),
           (2, bedaar, (be, ed, da, aa, ar)),
           (3, daar,   (da, aa, ar)),
           (4, darem,  (da, ar, re, em)),
           (5, warm,   (wa, ar, rm)) }

{lgi} =   { (aa, (2,3)),
            (ar, (1,2,3,4,5)),
            (be, (2)),
            (da, (2,3,4)),
            (ed, (2)),
            (em, (4)),
            (re, (4)),
            (rm, (1,5)),
            (wa, (5)) }
```

The search for words that contain the letter groups `ar` and `rm` now proceeds as follows. Extract the records matching `ar` and `rm` from the inverted list file. (The

[4]Technique inspired by an approach of [13].

secondary index file will be used for this purpose, as explained below.) These records represent sets of indices to records in the word list file. Find the intersection of these sets. Look up and return from the word list file, the words corresponding to all indices in this intersection. The general steps in the process to be followed may be represented as:

$$\begin{aligned} V &= \texttt{AND \{ar and rm\}} \\ &= \text{words}(1,2,3,4,5)\ \text{AND words}(1,5) \\ &= \text{words}(1,5) \\ &= \{\texttt{arm,warm}\} \end{aligned}$$

3.2.5 Inverted List in Practice

Approaching the matter in this way, each (existing) letter group contains only the relevant occurrence list. Each of the occurrence list items consists of a string of digits (with no leading zeros) in a comma-delimited format. In fact, in the latest version of the support tool, we have implemented a special 2-byte format storage technique for representing the occurrence list items. Each occurrence item now requires only 2 bytes, with occasional special delimiters imbedded, instead of the earlier versions that stored occurrence items directly as strings of digits.

In practice, this resulted—in the real case of building the total word database—in a collective database of size 1,5MB for the word lists and 8MB for the index lists. This is consistent with reports in the literature that the size of the inverted list usually exceeds the size of the word list. In this particular application, size is no longer a major concern, since the above storage strategies have rendered storage needs well within reach of modern storage costs.

From this example it is clear that the access method produces a list of word candidates. This is merely an interim step in dealing with the query. This list of candidates then needs to be filtered, possibly referencing more detail from the original query, in order to produce the answer set of words. The filtering is dependent on requirements set by the query language—such as the order of the substrings and limit(s) on word length. For example, it was pointed out in the small example discussed above, that the first step was to split the actual string required, `arm`, into two substrings `ar` and `rm`. It is merely co-incidental in this example that the answer set of words happen to comply with the earlier requirement so that no additional filtering was needed.

Over and above the 2-byte storage format of occurrence items in the inverted list to lower storage requirements, the current implementation also employs another technique to enhance the execution speed. This is next described.

3.3 Building the second-level index and reverse index

Because of longer than expected worst-case performance was experienced when accessing the inverted list file, special attention was given to the construction of the indices for the inverted lists. A novel 2-level indexing approach was implemented that involved fast string scanning and limited reading of the inverted list files. This, in combination with the 2-byte storage format, introduced dramatically faster results when retrieving frequently occurring letter groups, compared to the previous implementation technique used. (The latter were, however, more elegant in ObjectREXX code.)

The 2-level hierarchical index was built as follows. The first level contained, for each first character of the letter groups, a pointer to the second level index. Each entry in the second level index contains a second character, as well as a tuple for each letter group that has these two first letters as a prefix. The first element in the tuple is a pointer to the relevant inverted letter group in the inverted list file; the second tuple element is the number of elements in the occurrence list of that inverted letter group entry. By sorting this count information according to the number of elements, it was possible to use this 2-level index to dramatically reduce access time when searching for word patterns[5]

Note that access to the database takes place in read-only mode, as the focus is on fast retrieval of the results. This is acceptable because of the short time needed to rebuild a particular database. A complete rebuild of a database is done in batch mode when required.

4 THE QUERY LANGUAGE AND ENGINE

4.1 The Query Language Design

There are several search strategies, as mentioned above. The query language was required to specifically cater for ease of interactive use, and to be flexible enough to address the specific requirements of the particular example implementation. In later versions a rule-based engine will be considered. The current implementation includes imbedded "rules" for the challenge at hand.

In essence, the requirement is to cater for the possible search patterns as listed in table 3.
Per definition, a query consists of a recursive combination of these search patterns—with some restrictions to keep matters practical. In practice a user will combine stems and wild cards to construct a query. For example: `beu?lb*` will search for all words having substrings `beu` and `lb` in positions 1 and 5 while the stem is `beu?lb` [6].

4.2 The Query Engine Techniques

While constructing the example support tool a number of programming techniques were required. Listed

[5]It was also found that by analysing "hotspots", certain heuristics could be deployed to achieve yet better performance. However, these hotspots strongly related to the frequency of prefixes and suffixes commonly occurring in Afrikaans. To take the reader into a detailed account of Afrikaans structure is beyond the scope of this article.

[6]The answer turns out to be the words: `beuelblaser` and `beuelblasers`

Type of Pattern	Search Pattern	Comment
Exact match	$a_1a_2\cdots a_n$	Unique string = word
Stem words, OR-ed	$a_1a_2\cdots a_m*$	Back end truncation
Words ending, OR-ed	$*a_1a_2\cdots a_m$	Words ending with substring $a_1a_2\cdots a_m$
Word stem and ending known	$a_1a_2\cdots a_m * a_p\cdots a_n$	Words with stems and endings
Word substring	$*a_1a_2\cdots a_n*$	
Stem words+1, OR-ed	$a_1a_2\cdots a_m?$	Words with single letter ending
Words ending+1, OR-ed	$?a_1a_2\cdots a_m$	Single letter word stem
Word stem and ending known Restricted word length	$a_1a_2\cdots a_m?a_p\cdots a_n$	Single letter between stem and ending

Table 3: Types of search patterns

here is a summary of the most important of these techniques and approaches taken.

— Fast string searching on short strings—pos() function of ObjectREXX
— Excellent parsing features on short string—parse function of ObjectREXX
— Special substring technique for long string parsing
— Multi-level indexing employed on dictionary
— Comparing and AND-ing occurrences as strings
— Pre-reading candidate strings (named "kstringe") to obtain occurrence frequencies
— Ordered AND-ing of the candidate occurrence strings to enhance performance
— Specifying a limit on the number of words to retrieve
— Filtering techniques to produce the answer set of words starting from the candidate set of words.
— Combining different search techniques dependent on the word length in a hybrid approach

4.3 The Query Steps

The macro steps for the query engine can be listed as follows:

[query]
⇒ analyse query for type and pattern
⇒ recognise {query letter groups} aka {search parts}
⇒ get inverted letter group (ilg) frequencies
⇒ optimize search order: order {search parts} from small to large; this forms a crude "effort function"
⇒ search {search parts} for candidate words list
⇒ AND-ing of candidate strings
⇒ read candidate word list (wl)
⇒ filter {candidate word list} according to query's pattern
⇒ obtain {set of answer words}

4.4 Filtering the Candidate Word List

The letter group technique, which is primarily used for partial match retrieval, indexes only subparts of words. Therefore the preliminary AND-ing of the candidate occurrence strings delivers only candidate words for the query not the final answer set. The following two examples illustrate this:

Example 1: A query against database db8 (having words of length 8 only) for `speld` and using three-letter groups starts by identifying the letter groups `spe` and `eld` in the query. (Note that we do not handle letter group `pel` which we realize will be unnecessary while simultaneously reduce the number of occurrence list comparisons drastically.) This results in the following 9 words in the candidate word list: `aanspeld`, `dasspeld`, `kopspeld`, `losspeld`, `speeldae`, `speeldag`, `toespeld`, `uitspeld`, and `vasspeld`.

The two words `speeldae` and `speeldag` are candidates but not answers, and will be filtered out of the final answer set. They were retrieved as candidates because they each contain the string `speeld` which is the concatenation of the two three-letter letter groups identified from the query.

Example 2: The query `babel` against database db6 (that contains only words of length 6) results in the candidate word: `babbel` which is filtered out to result in an empty answer set; because this database does not contain `babel` which is only of length 5 (the city name that appears in the Afrikaans Bible).

5 THE WORD PUZZLE LANGUAGE

Although word puzzles are challenges (one may say 'games') of matching natural language words in a specific combination, it is convenient to regard the specific language used in a word puzzle as a language with specific peculiarities. Such a 'word puzzle language' consists of a subset of the natural language words, but also includes some 'non-words'. This is specifically true for languages (such as Afrikaans) that employ various special characters (such as hyphens and quotes) as well as diacritic symbols (such as characters using circumflex and diaeresis). The puzzle language may well require that such special symbols are ignored or treated in a particular way. Even in English word games, one may find that the puzzle language combines the words in a phrase into a single "puzzle word". The puzzle language then appears to be less general and more restricted in terms of corpus size. In the present study,

*^-	hyphen (koppelteken) removed from word
*^'	accent removed from word; ex. BUROS instead of buro's.
*^b	blank removed from phrase; ex. ADHOC instead of ad hoc
*ë	word contains ë
*ê	word contains ê
*ï	word contains ï
*_=	word stem; ex. elektro= and hiper=
*_-	word stem with hyphen; ex. antropo- and mikro-

Table 4: Additional information stored in puzzle language dictionary

the puzzle language was built according to the rules below.

5.1 Syntactical Rules

Words consist only of capital letters A,B,···,Z. Special characters and diacritics are translated to these capital letters according to the following rules:

- Hyphenated words are written as single words. For example, AANAAN is used instead of aan-aan and ABCBOEK instead of ABC-boek.
- Diacritic symbols are normalised. For example aasvoëls becomes AASVOELS; Australië becomes AUSTRALIE; argaïes becomes ARGAIES; arbitrêr becomes ARBITRER

These special translations were done for the support tool dictionary. However, references were included in the dictionary to signal information about the translated word, as summarised in the table 4.
Care was taken to exclude duplicate words that result from these translations, for example: voel and voël both translate to VOEL.

Afrikaans employs a number of diacritic symbols for correct pronunciation and semantics. Eleven different diacritic symbols were found in the original word corpus. Table 5 shows the frequency distribution of the special symbols in the original word corpus before the translations were done for the “puzzle language”. (The corpus size was greater than 131000 words at the time of analysis.)

5.2 Additional Dictionary Features

Word length restrictions posed another language constraint on the construction of the dictionary. In other language processing contexts, morphological information is often stored to allow the user to derive word extensions such as plurals, etc. However, in the present case, it was decided to record word extensions as separate words.

In order to imbed more knowledge of the corpus words in the approach the analysis of the words was done by considering that words “contain” two additional characters—both being ‘_’ (i.e. underscore)—indicating the first word stem character and the last word ending character. For example: the word iets is considered to be _iets_ before the substring analysis starts. This allows the query language to easily include stem word searches (handling prefixes) and word ending searches (handling suffixes). Having this format then allows queries such as $a_1a_2\cdots a_m$* to be equivalent to the exact string query $\_a_1a_2\cdots a_m$ (and consequently, to find words with letter groups $\_a_1a_2\cdots a_m$) and, vice versa, the query $*a_1a_2\cdots a_m$ is equivalent to the exact string query $a_1a_2\cdots a_m\_$. This obviously adds a host of additional letter groups to the dictionary database but greatly enhances the dictionary access method's ability to accommodate more sophisticated queries.

	Char	Hex	Dec	Frequency
1	ë	89x	137	2437
2	-	2Dx	45	662
3	’	27x	39	209
4	ê	88x	136	644
5	ï	8Bx	139	562
6	é	82x	130	74
7	ä	84x	132	12
8	ô	93x	147	58
9	è	8Ax	138	19
10	ö	94x	148	85
11	ü	81x	129	17

Table 5: Frequency counts

5.3 Properties of the Puzzle Language

The foregoing rules gave rise to some oddities in the Afrikaans puzzle language. For example the puzzle language contains words such as teeeet (from teë-eet), toeoe (from toe-oë), and seeeend (from see-eend). However, these occurrences affect neither the overall search strategies employed in the tool, nor its storage requirements.

Having constructed the dictionary in this format allows one to analyse the special cases for the specific language at hand. One can, for example, analyse the existence or not of certain letter group combinations; the occurrence of vowels following or preceding different letter group combinations; occurrence of letters as first letter of word stems or last letter of word endings; common word stems; common word endings; etc. These findings will lead to special “puzzle language” heuristics which are expected to eventually assist significantly with the envisaged “rule based” extensions.

The experience gained while constructing the dictionary and the example query engine was most valuable in comprehending the complexity of the challenge at hand. The skewness of the letter group frequencies was expected, but it was nevertheless a surprise to experience the reality of the final corpus. The frequencies are listed in the appendix of the different lengths of words found in the corpus. The appendix also contains a table that lists the 20 highest occurrences of letter groups found in the corpus as well as a list of letter frequencies for the Afrikaans corpus.

One should note that the construction of the letter groups was fully automatic. No distinction is made on the practical value of a letter group: a letter group was recognised merely on the grounds that it appeared somewhere in the corpus.

6 CONCLUSION

Possible extensions to the implemented example support tool are summarized in the following list.

1. Other development language—such as C for certain core routines to enhance worse case performance; or Java for deployment in a browser environment
2. Enhanced word puzzle specific user interface—made much more interactive and "click" oriented than current version
3. As part of an intended extension in a following version of the support tool the building of a "reverse index" is being considered—which is essentially an adapted B-tree approach.
4. Word list extended to include thesauri type entries for display and linking to other parts of the dictionary
5. Use the current constructs as a test bed for testing other approaches such as employing "automata" and fast string searching on the dictionary
6. Develop heuristics and rules and construct a Rule Search Engine
7. Introducing online updating capabilities (low priority)

It was worth the effort to experience in practice what a number of articles articulate in theory. The result of the study became a very handy support tool for resolving the challenge of the "letterspeletjie". Even relatively simple dictionary-based knowledge collections—one may dare to even call these databases "simple knowledge bases"—possess a surprisingly large complexity. Even simple approaches turned out to be very beneficial for resolving this type of challenge. Again it is confirmed that simple solutions are much more effective than they are usually given credit for. This type of approach offers relative ease of maintenance and expansion. Exciting extensions and alternate approaches can result from this study. A practical support tool for the specific example "word puzzle" was constructed and is in *regular* use by the primary author. It has reduced the time to solve a puzzle from several hours to approximately 30 minutes.

REFERENCES

[1] L. L. Pansegrouw. *Superblokraaiselwoordeboek.* J L van Schaik, 1995. ISBN 0-627-02024-8.

[2] A. C. Botha. *Dokumentherwinnig met behulp van Herwinhulpmiddele.* M.sc. dissertation, University of Pretoria, 1980.

[3] G. Salton. *Automatic Text Processing, The Transformation, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, 1989.

[4] R. L. Rivest. "Partial match retrieval algorithms". *SIAM Journal of Computing*, vol. 5, no. 1, pp. 11–49, March 1976.

[5] D. E. Knuth. *Searching and Sorting, The art of Computer Programming.* Addison-Wesley, 1976.

[6] W. B. Frakes and R. Baeza-Yates (editors). *Information Retrieval Data Structures and Algorithms.* P T R Prentice-Hall, Inc., 1992.

[7] R. S. Boyer and J. S. Moore. "A fast searching algorithm". *CACM*, vol. 20, no. 10, pp. 762–772, October 1977.

[8] D. E. Knuth, J. H. Morris and V. R. Pratt. "Fast pattern matching in strings". *SIAM Journal of Computing*, vol. 6, no. 2, pp. 323–350, June 1977.

[9] F. Murtagh. "A very fast exact nearest-neighbor algorithm for use in information retrieval". *Information Technology: Research and Development*, vol. 1, no. 4, pp. 275–283, October 1982.

[10] A. Drozdek. *Data Structures and Algorithms in C++.* Thomson Course Technology, third ed., 2005. ISBN 0-534-49182-0.

[11] W. A. Burkhart. "Hashing and trie algorithms for partial match retrieval". *ACM Transactions on Database Systems*, vol. 1, no. 2, 1976.

[12] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings; Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002. ISBN 0-521-81307-7.

[13] H.-J. Schek. "The reference string access method and partial match retrieval". *IBM TR*, vol. 5, no. 77.12.008, December 1977.

APPENDIX

Length	Frequency	Length	Frequency
1	2	18	1171
2	93	19	772
3	608	20	530
4	1810	21	296
5	4176	22	226
6	7661	23	140
7	12487	24	97
8	15733	25	61
9	18592	26	45
10	18038	27	23
11	14587	28	15
12	11394	29	13
13	8487	30	6
14	5955	31	6
15	4369	32	2
16	2625	33	2
17	1748	34	1

The single 34 letter word is "MEERVOUDIGEPROBLEEM-SIEKTETOESTANDE"

Table 6: Frequency count of words in database Afr (131771 words at time of analysis)

1	13900	ING
2	7753	NG_
3	6553	STE
4	6454	TER
5	6397	VER
6	5845	IES
7	5762	SIE
8	5131	TE_
9	4107	NGS
10	3731	SE_
11	3399	RS_
12	2954	UIT
13	2553	WER
14	2362	TIG
15	2095	TEE
16	1867	TJI
17	1707	TIN
18	1634	ROO
19	1583	TRO
20	1538	TRA

Table 7: The 20 highest frequencies of three letter word groups for database Afr

65	A	3	97	a	100518
66	B	0	98	b	27697
67	C	0	99	c	1133
68	D	3	100	d	53061
69	E	0	101	e	220267
70	F	0	102	f	16271
71	G	0	103	g	64771
72	H	0	104	h	16301
73	I	0	105	i	101815
74	J	5	106	j	5198
75	K	0	107	k	61156
76	L	0	108	l	67391
77	M	0	109	m	33302
78	N	0	110	n	82381
79	O	1	111	o	85710
80	P	0	112	p	33252
81	Q	0	113	q	49
82	R	1	114	r	106084
83	S	10	115	s	104145
84	T	0	116	t	72874
85	U	0	117	u	33915
86	V	13	118	v	18566
87	W	20	119	w	16480
88	X	3	120	x	76
89	Y	6	121	y	10315
90	Z	9	122	z	117

Table 8: Frequency count of characters for database Afr