Data Structures for Natural Language Processing
Dave Davies, 2005

Abstract
The implementation of Natural Language Processing systems poses some interesting problems for the Software Engineer. In addition to the algorithmic problems associated with parsing and inference in an open-ended language there is the challenge of providing a diverse range of data structures needed to deal with issues of complexity, efficiency and reliability.

In this paper we look at several data structures that have been designed to facilitate and enhance Natural Language Processing. First we look at the Link Matrix, a structure designed to provide efficient lexical association. We then look at a set of structures forming a Content Addressable Memory (CAM) based on the use of Hashtables. These structures can be viewed as providing a framework for the systematic generation of Hash Keys. The system also employs a range of Iterator structures that facilitate dealing with the complexity and ambiguity inherent in NLP.

Introduction
If computer programs can be viewed as consisting of algorithms plus data structures then much of the evolution of Computer Science can be viewed as the development of specialised data structures to simplify and improve the efficiency of algorithms. The complexity of NLP provides a particularly demanding domain for software development and the work described here can be viewed as developing, and extending structures to suit it.

The Software Engineering processes and techniques discussed here form part of a Natural Language Agent project. While the project is aimed at commercial applications it is hoped that it will also provide a useful tool for Natural Language research. This paper is the first of a series of three that tackle the problems of data structures, parsing, and inference in a Natural Language context [Davies 2005b, 2005c].

The system described here is a fine grained (word level) object-oriented design implemented in Java. It provides low level inter-process communication via the XML-RPC protocol that allows any instantiation of the system to act as a server for either dictionary (word level) or knowledgebase (sentence level) data. The layered system architecture is based on the model-view-controller pattern. Here we consider only the model layer.

Text indexing
Many techniques for text indexing or file inversion have been discussed in the literature over the years, particularly in relation to searching large text databases or the Web. Reviews of indexing and search techniques can be found in [Heinz and Zobel 2003] and [de Moura et.al. 2000] respectively. The problem domain addressed in this report differs significantly from those usually targeted in text search engines. Here we are interested in providing an infrastructure for mechanical inferencing in text collections (knowledgebases) of up to a gigaword rather than terabytes typically indexed in search engines. We want all the text to be in memory, all words indexed, and the sentence structure to be accessable through the index.

History of the project

The principal data structure, the Link Matrix (LM), was initially developed for application to abstract symbol matching in a program for playing the board game Go. Its application to NLP [Davies 1983] had the primary objectives of text compression and indexing. The LM formed the basis of The Word Machine which was released commercially as an ideas processor in 1986. The Word Machine Inference Engine was not released commercially, primarily due to the restrictions posed by hardware limitations of the time. Contemporary desktop hardware provides ample memory and speed for large scale text processing and inference on large rule sets.

Design

Best Practice in the Software Engineering design phase demands the prior specification of a set of requirements that adequately define the scope of the task. Functional and usability requirements for the full system are discussed in associated papers [Davies, 2005b, 2005c]. Here we consider a requirements focussed design for the data structures used.

System requirements from he parser and inference engine:

1. Size: The system should allow the processing of millions of statements but be capable of restricting algorithmic complexity to just those statements that are relevant to the task.
2. Performance: The system should be capable of parsing in excess of one thousand statements per second and performing over one thousand inference steps per second.
3. Expansion: The system should allow for the runtime expansion of the language definition and inference policy.
4. Lexicon: The system should be capable of efficiently handling an extensive English word set including variations in meaning and Part-of-Speech (POS). It should be runtime extensible.

Resulting Software Engineering Requirements:

1. Process: The development should follow industry standards appropriate for a large scale software package.
2. Modularity: System design should use modular package and layered structures at a high level and employ effective Object Oriented techniques at lower levels.
3. Flexibility: The system should provide a ‘plug-in’ structure that allows rapid and safe code modification, upgrade and runtime flexibility. Data structures used in the lexical processing and grammar should be capable of runtime definition.
4. Memory usage and performance: Current memory availability allows scope for trading memory usage for performance.
5. Quality: Industry Best Practice standards should be followed in Design, Construction, Testing and Documentation.
6. Connectivity: The system should allow access to operating system level functions and network connectivity. Consideration must be given to fast data communication between concurrently running instances of the system, with other NL systems and with problem domain applications

7. Cross-Platform Capability: The system should be available on Windows, Linux, OSX and Solaris platforms.

In following sections we outline the constraints placed on the system by these requirements and the principal design solutions: the Link Matrix, Hash Key Grammars and the use of nested Iterators.

The Link Matrix

In this section we deal with the problem of text indexing and lexical association. Alternatively we can view the process as the construction of a semantic network. The task is to efficiently associate all statements in a knowledge base (KB) that contain instances of a particular word. In doing so we can reduce the complexity of inference steps from an order related to the size of the complete KB when a full search is required, to one related to the number of statements that may have relevance due to direct or indirect lexical or semantic associations. To clearly distinguish this task from conventional text indexing we derive the structure in a systematic manner starting from a simple tokenised KB representation and progressing to the Link Matrix..

Symbols used in the analysis are:

W	Word object: stores the ASCII form of the word and other information
D	Dictionary object: stores word objects and instance counts
T	Token object: pointer to a word in the dictionary
S	Sentence object: an array of words, tokens or links
L	Link object: a pointer to another word instance in sentence
I	Index object: pointer to the first occurrence of word in hte KB

We start with a simple four word sentence:

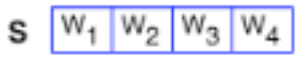

Fig. 1. Simple sentence

Next (Figure 2) we create a dictionary and tokenise the text in the conventional manner replacing each word in the KB with a pointer into the dictionary. By tokenising text, moving from a character based level to the word level, we can increase temporal efficiency and achieve a small gain in spatial efficiency. However, we still have to search the complete data set to investigate the associations between different instances of a word.

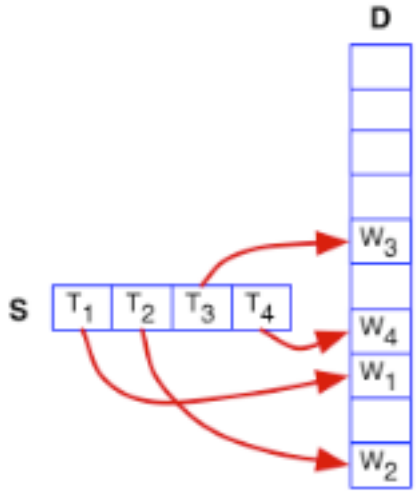

Fig. 2. Tokenised text

The simplest approach to lexical association is an index. For our purposes here the index references every instance of a particular word in the KB from an instance array (e.g. L3... in Figure 3). An index can be effective from the point of view of speed, but we have introduced an inefficiency in memory usage. In Figure 3 we can see that by having an index entry for each word in the data set we have doubled the memory used for each word instance.

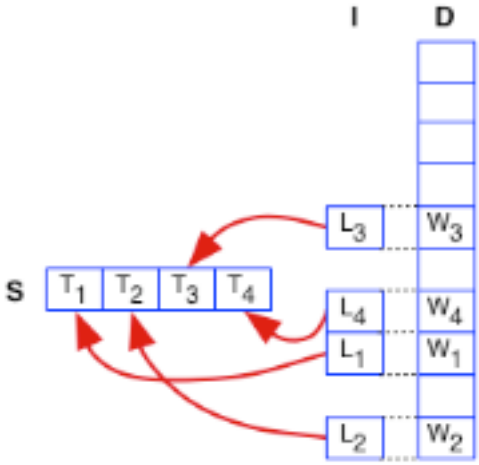

Fig. 3. Indexed Sentence

To progress from this point it is helpful to analyse the nature of the information that we are storing. One requirement is that we retain the structure of the sentence. In Figure 3 this is done in the sentence array S. We can absorb this information into the index by replacing links to the sentence array with links to the corresponding items in the index.

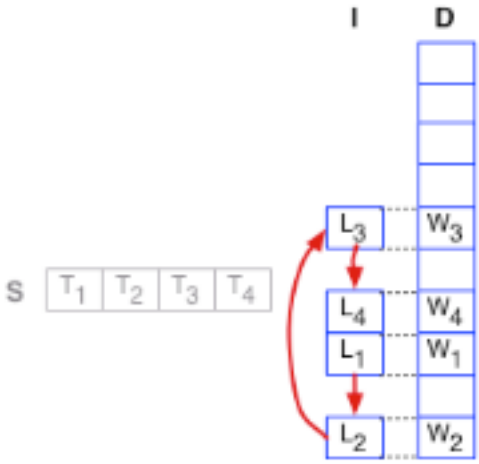

Fig. 4. Linked Index

This structure is illustrated in Figure 4. The original sentence S is now represented by the link sequence [L1, L2, L3, L4] with the ends of the list joined to form a loop allowing access to the complete list starting from any element. The original sentence is now redundant so we have returned to the memory usage of the tokenised sentence but have retained the lexical association information in an efficient form.

Pushing the analysis of the information content of the data structure further we can view the new Linked Index form of the sentence as a two dimensional hyperplane consisting of the index array in one dimension and a linked list in the other. In each of these dimensions there is information embedded in the order of the elements. The order in the linked list is used to represent the word order of the sentence. The order of the index array is arbitrary and unused.

One way that we might be able to utilise this otherwise unused information is to represent priorities for lexical associations. From a software engineering perspective, however, the structure is sub-optimal. Arrays are relatively inflexible in expansion and changes in order whereas Linked Lists are readily expanded and provide a flexible representation of set order. What we need is an array representing the sentence, and to represent word instances in a linked list. We need to invert the structure.

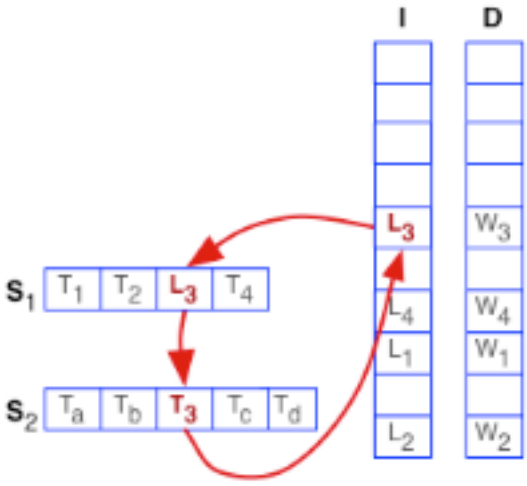

Fig. 5. Link Matrix (instances of only one word are linked)

Figure 5 shows how this can be achieved. To fully demonstrate the structure a second sentence has been included. The index structure has been inverted with the sentences now in arrays and the lexical associations represented in the linked lists. As with the Linked Index, each instance list is joined at the ends to form a loop so that the whole list can be accessed from any starting point within it.

Optimisation

We have seen that the LM, by using a linked list structure for the lexical associations, provides a means of prioritising the order in which instances are accessed in the list. In practice this can be achieved by placing an instance at the front of the list each time the statement containing it is used successfully in an inference. This has the effect of spreading the optimisation over all inference tasks. To reduce the impact of this activity on inference performance it can be allocated to a low priority background task.

The lexical associations discussed so far apply to individual words. Accessing the KB for a target pattern consisting of several words can be optimised with the aid of accumulated instance counts for each word. If the word with the lowest instance count is used as the start word of the pattern matching process, the number of sentences tested is minimised.

Emergent structures

So far we have discussed only word associations in the LM. At minimal additional cost we can also link punctuation which has some interesting consequences.

If punctuation is linked, and we control the scope of these links, we can create a variety of structures such as tables and tree structures within documents. Any sentence with components delimited by tabs, commas or semicolons can be viewed as a row in a table. Multiple rows with comparable structure can be associated through the links of these delimiters. The table can be distributed arbitrarily through a document or even across documents. Colons can delimit row headings or a table heading that is followed immediately by column headings linked to their respective cells within associated rows. Linked together, such

tables can be viewed as records of a distributed database structure.

The outliner format commonly provided by word processors provides a tree structured hierarchy for document contents. Multiple consecutive tabs or other punctuation characters leading a sentence or paragraph can provide this structure naturally within the LM. The Nth tab, if linked only to other Nth tabs within a predefined scope, defines the Nth sublevel for the tree structure and can recursively form the head of a substructure.

The structure of documents can also be represented using XML. The tokeniser used here parses HTML and XML tags as a single word. Identical tags are are linked to provide a content-type superstructure for document sets. For a Natural Language Inference Engine these embedded structures can provide an efficient method for providing structure and semantics for grammar rules and the KB.

Hash Key Grammars

Hashtables can be viewed as providing a form of associative or content addressable memory implemented in software. Any information that can be represented by a unique integer can be combined in arbitrarily complex ways to form a unique identifier for the combination and used as a key to access information stored in the table.

In practice it is desirable to assert some structure to the formation of keys. It is this structure that we refer to as Key Grammars. Having formalised patterns for key structures provides a level of generalisation in the use of tables that enables a system to create new data structures at run-time based on novel inputs or discovered patterns. By providing a uniform structure that is used consistently across the system we can provide an answer to the question: 'If some other process has already solved this task and stored information, how was it likely to have been stored?'

Inbuilt support

The simplest form of compound Hash Key used in this work is is a binary form:

Key = {<set label> <value label>}

which represents a set with name <set label> containing elements referenced by <value label>.

In the current implementation some basic functionality is inbuilt with an abstract class HashSet sub-classed as HashArray and HashList which take numeric and text data labels respectively. The HashArray can be compact (no missing values) or sparse. In its sparse usage it is not enumerable. The HashList stores the keys used so is always enumerable. An iterator class is provided for the enumerable forms.

The HashSet forms share common Add (extend), Set (overwrite), Remove and Iterator functions. They can be used recursively (each form can contain the other) to create arbitrarily complex tree structures.

Implementation

A primary example of the use of HashSets is dealing with lexical ambiguity. Most words

exhibit ambiguity in the written form. Here we represent words as a HashSet consisting of a compact HashArray, the elements of which represents alternative meanings for the word (Figure 6). A second dimension to the structure formed using HashLists is used to store the possible Part of Speech alternatives associated with each meaning.

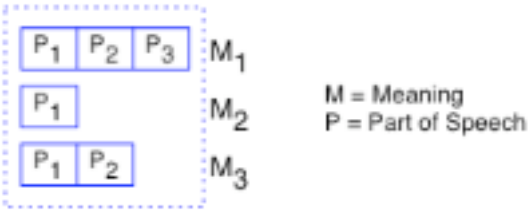

Figure 6. Typical word variant structure for a single word

Using this set structure the Word class is sub-classed as a MultiWord form that uses the HashSet and its Iterators to allow two dimensional scanning of meaning and POS in any context that a word instance appears. The Iterators can be locked to a particular value.

Consequent to this treatment of lexical ambiguity, sentences containing MultiWords can take multiple forms. A VariantSentence class is provided to allow iteration through the possible sentence forms. An algorithm using grey-scale coding was developed to ensure the uniqueness of sentence representations generated by multiple iterators and to ensure that the most common meanings were presented first for each word in a sentence (assuming that the meanings have previously been prioritised in the dictionary).

To facilitate runtime tracing and debugging, an inherently messy problem when using hash tables, labelled and unlabelled forms are provided in the HashKey class. The labelled form maintains a list of the objects used to create the key and associated display methods that can be used for tracing and debugging. The more compact unlabelled form can be used when tracing is not required.

The current implementation of the LinkMatrix uses a HashArray structure to store linked sentences and a HashList to store document level information. Further examples of the use of structured hash keys will be given in subsequent papers in this series.

Summary

Bibliography

Winograd, Terry **1979**, Beyond Programming Languages, Communications of the ACM, 22(7).

Mizoguchi, Fumio and Kondo, Shozo **1982,** A Software Environment for Developing a Natural Language Understanding System, COLING'82, North-Holland.

Davies, David R.L., **1983**, A data structure for text compression, Australian Microcomputer Software Conference, ACS, Canberra

de Moura, Edleno Silva; Navarro, Gonzalo; Ziviani, Nivio; Baeza-Yates, Ricardo **2000**, Fast and Flexible Word Searching on Compressed Text, ACM Transactions on Information

Systems, Vol. 18, No. 2, April 2000, Pages 113–139.
Heinz, S. and **Zobel**, J. **2003**, Efficient single-pass index construction for text databases, Journal of the American Society for Information Science and Technology, 54(8):713-729, 2003.
Davies, David R.L., **2005b**, A Table Based Parser for Natural Language Processing, Paper 2 in this seriesPaper 3 in this series.
Davies, David R.L., **2005c**, A Table Based Inference Engine for Natural Language Processing, Paper 3 in this series.