Building Efficient Natural Semantic Nets

David R. L. Davies
School of Information Sciences and Engineering
University of Canberra
Canberra, Australia, 2601
dave.davies@canberra.edu.au

Abstract
Semantic Nets form an integral part of Natural Language inference but they can greatly increase computational complexity in Natural Language inferencing systems. A general approach to reducing computational complexity is to trade algorithmic complexity for data complexity by building data structures tailored to the requirements of efficient algorithms. This paper outlines the application of specialised data structures to the problem of dealing with complexity and usability in Natural Language Processing.

1. Introduction
Construction of large Semantic Nets such as Cyc (Lenat 1990, 1995) and WordNet™ (Miller 1995) initially relied heavily on human involvement. With the rise of the Web and the availability of other ‘found data’ in electronic form there has been increasing interest in generating nets from the data. ConceptNet (Miller 1995) uses NL data but converts it into a computationally oriented internal form. The Word Machine (TWM) described here uses a compact representation of the original NL data internally which can improve both reliability and usability as the internal processing is human readable.

Current software IDEs provide extensive libraries of data structures. These range from implementations of simple arrays and lists to to highly adorned set structures that include the ability to convert, at runtime, to a form that suits the task at hand. These library structures form a powerful foundation for application specific structures. The structures described here for NLP have been implemented on top of the java.util library.

The Word Machine is a Natural Language Inference Engine. NLP presents the software engineer with multiple sources of complexity. The first, which is not dealt with in detail here, is the problem of tokenisation - converting a string of characters into a stream of word level tokens. Technical solutions to this problem have been developed and refined in the domain of computer language compilation but Natural Language provides diverse context dependent demands.

Found data can include HTML and XML tags along with a wide variety of formats for lists of text and numeric data. The Finite State Machine used in TWM is run-time adaptable allowing the tokenisation policy to be adapted to the domain.

Once text is tokenised we have to deal with the problem of lexical ambiguity. Many English words have multiple meanings or senses and within each meaning the part-of-speech (POS) assignment may vary. Solutions to this problem are addressed in Section 2.

The third source of complexity, the Semantic Net constructed from lexical associations embedded in rules of the knowledge-base, is addressed in Section 3. Section 4 deals with a layer of abstraction of the Semantic Net generated from partial and completed inference results. It also describes techniques for systematically specifying context for the construction of domain specific ontologies. Section 5 looks at the application of the semantic net to the process of semantic inference.

2. Lexical ambiguity and Iterators
The primary requirements for dealing with lexical ambiguity in the parser and inference engine are, firstly, that at all stages the complete set of alternates, both meaning and POS, be available for scanning and, secondly, that lexical associations relevant to each alternate be independently accessible. The third requirement is that the representation used be capable of run-time update and modification.

The first of these requirements is readily met by a combination of conventional Object Oriented Design Patterns. The Word class is sub-classed to a MultiWord form that can present itself as any of the alternates in turn using an embedded Iterator class. A Next method scans all alternates while NextDown and NextAcross methods allow the scanning of meaning and POS separately. At any stage the Iterator can be locked to fix a particular meaning and POS.

The direct independent access specified by the second requirement is provided by the Associative
Memory (AM) structures discussed in Section 4. The Word iterators scan a two dimensional array embedded in a hash table. Individual hash keys can be generated for each of the items in the array. The third requirement of dynamic update is also provided by the AM.

Additionally, the parser emits a MultiSentence form for sentences that have more than one parse as a consequence of lexical ambiguity. As with the MultiWord class, the MultiSentence has an embedded Iterator that allows scanning of alternate parses. The combinatorial problem created at this stage is not yet controlled, as it could be, by the enforcement of semantic constraints across the sentence. In extreme cases thousands of alternates may be generated.

3. Semantic Nets and the Link Matrix

![Figure 1: Illustration of a word link chain in the Link Matrix](image)

Central to the computational efficiency of Natural Language Processing is the creation of efficient semantic nets. In the Link Matrix (LM) (Davies 2005a), each sentence in the KB is represented by an array of links with one link per word. Each link points to a previous instance of the word in the KB in a looped linked list. An example of a link chain for the word ‘cat’ is illustrated in Figure 1. Each distinct sense of a word forms a separate loop with the loops connected via the MultiWord Iterator.

The LM forms a cyclic net which, in general, would be difficult to navigate. However, the simple nature of the loops gives this structure a complexity comparable to an acyclic net. Since the order of the nodes in each loop is initially arbitrary, the order can be dynamically updated to provide a prioritisation of word instances within a loop.

In practice, a process can scan a loop looking for instances of the word in a particular syntactic context - e.g. ‘the subject of a class definition’ as in the second sentence of Figure 2. Each word instance along the loop has a compact representation of its context within the sentence parse enabling fast context checking. The full sentence parse is also available from an Associative Memory structure described in Section 4. The parse information allows other components of the sentence to be efficiently accessed (e.g. ‘the object of the class definition’ represented by ‘a mammal’ in the previous example). The parse representation and access is discussed in (Davies 2005a) and (Davies 2005b). Using the parse information all lexical associations are automatically categorised by the verb phrase, adjuncts or modifiers used in the defining statement.

The LM also links punctuation and other non-lexical tokens such as HTML and XML tags. This provides document-level structures for defining context. Higher level structures classifying document sets could also be included in a similar manner.

4. Hash tables and hash-key grammars

Underlying the Iterators discussed in Section 2 and overlaying the LM are structures embedded in hash-tables. These tables form an Associative Memory accessed by structured, compound hash keys. Computationally, any Object (e.g. word, sentence, parse tree) can be represented by a unique integer that can be used as a key into a hash table. Keys for sets of Objects can be generated by combining their individual keys in an order-dependent manner for lists or otherwise for unordered sets.

For any object, different properties, or attributes, can be stored and accessed using keys that include a label for the attribute. For example, the parse structure for a sentence can use the key set {“parse”, sentence_key} where the sentence_key has been constructed from the individual words of the sentence. The same parse structure can also be stored more generally using the key set {“parse”, pos_sentence_key} where the POS form of the sentence, shared by many sentences, is used in the key.

The use of a specific label in a key (e.g. “parse”) defines a set of keys that constitutes the domain for that label but if the position of the label within the key is unconstrained then mapping that domain is complex. We can reduce this complexity by providing constraints on the structure of keys using standard templates or patterns. Using generalised rules for creating these patterns leads to the concept of hash-key grammars.

With structured keys, the key {“parse”, sentence_key} defines an element in the set of parses while the key {sentence_key, “parse”} defines an element in the set of information for a particular sentence. We can now access information in an efficient manner but only if we
know what we are looking for. The set of all parses or all information on a particular sentence are still unavailable.

To overcome this problem two virtual set structures are defined within the hash-table. The first, a virtual array, is created automatically if two objects are stored with the same key. The second, a labelled set, stores a label for each member of the set. Both sets share Iterator structures that allow scanning of the sets. Either set can contain an instance of the other so generalised virtual trees are possible.

These provide the structures described in Section 2 for storing words (key = {"word"}), word sense variants (key = {"word", word_key}) and the possible POS assignments for each word (key = {"word", word_key, pos_label}). They also provide storage for parse information and the partial and full results of inference (Davies 2005c). The inference results can be viewed as a superstructure sitting above the LM and providing short-cuts for the traversal of semantic structures.

5. Discussion

A primary motivation for good Software Engineering is usability. The data structures described here address three usability issues: performance, reliability and transparency.

To perform useful tasks over a wide range of domains, NLP requires large amounts of data to enable a comprehensive world-view to be established. The system described here is designed to handle gigabytes of data. To achieve this efficiently it is necessary to constrain search in the inference process to just those statements that are directly relevant to the task. The LM achieves this goal. We also need to ensure that expensive processing is not unnecessarily repeated. The use of Associative Memory makes this possible.

The reliability of a complex software system depends critically on structured design. Object Oriented techniques, a layered Design Pattern (tokenisation, LM and AM) along with the structure imposed on the use of hash keys all provide support for reliable development.

Transparency, the ability to readily see what is going on in the inference process, is important both to the developer and end-user. The use of NL in defining semantic structures and the ability to retrieve all steps in the inference process in a human readable form allows the user to interact with the system and guide it where necessary.

An example of the output of the system in operation is shown in Figure 2. The initial query “Does Tom eat meat?” is transformed into a simple statement to be verified: “Tom does eat meat”. The subject phrase, verb phrase and object phrase of the query statement are incrementally expanded across the semantic net. With each addition from the net a new formulation of the query statement is constructed and if its hash key accesses a statement in the KB a solution is found.

1 Verify: Does tom eat meat?
2 Found: A cat is a carnivore.
3 Found: Tom is a cat.
4 Found: Devours means eats.
5 Found: Does eat means eats.
6 Found: Mice are mammals.
7 Found: Mammals are animals.
8 Found: Animals are meat.
9 Found: A carnivore devours mice
10 So: Tom does eat meat.

Figure 2: Semantic inference

The class definitions and associated property inheritance of statements and sets in lines 2 and 3 in Figure 2 extend ‘Tom’ to ‘cat’ and ‘carnivore’. Equivalence statements in lines 4 and 5 associate ‘does eat’ with ‘devours’ and in lines 6 to 8 a relationship between ‘mice’ and ‘meat’ is established. The statement in line 9, accessed from the KB with a key constructed from ‘a carnivore’, ‘devours’ and ‘mice’, can now be used to support the original proposition. Note that the rule in line 5 substitutes for syntactic and morphological processing that is not yet fully implemented.

Performance of the inference engine is not easy to quantify in any standardised form but the inference chain developed in Figure 2 took less than 10 milliseconds on a 400 MHz PC.

6. Conclusion

The application of NLP to practical IT problems has languished for half a century as a backwater in Information Systems. In this paper I have attempted to show that the application of advanced Software Engineering techniques can increase the usability of NLP to the point where it can become a ubiquitous tool, enabling users with no understanding of conventional programming languages to write programs or build Knowledge-Bases in the Natural Language of their choice.

References


David R. L. Davies 2005b. *A Table Based Parser for Natural Language Processing*, awaiting publication

David R. L. Davies 2005c. *A Table Based Inference Engine for Natural Language Processing*, awaiting publication.