How to Make NL "Understanding" Systems Run Orders of Magnitude Faster

by Steve Richfield, IEEE 41344714
CTO of *our thing*
5498-124th Avenue East; Edgewood WA 98372
Steve.Richfield@gmail.com 505-934-5200

Abstract - *A new method of greatly speeding up natural language (NL) "understanding" is presented here. This speedup is achieved by nearly eliminating the overhead of failed tests during parsing and subsequent processing. Failed tests account for more than 99% of time taken by present natural language understanding systems. This speedup is independent of the particular NL processing algorithms used.*

This method triggers queuing of rules based on the appearance of their least frequently used elements, so rules whose least frequently element is absent incur no overhead. Lower-level rules are placed in higher-priority queues. High-level rules create output.

This new capability should revolutionize the Internet by making practical the real time analysis of postings for particular semantic content to trigger individually crafted advertisements or AI replies. This method will also facilitate developing a "drop in" natural language understanding module suitable for many disparate applications.

Keywords: Natural language processing, Software performance, Detection algorithms, Dynamic compiler.

1 Introduction

A subtle phenomenon has doomed natural language (NL) projects for the last 40 years. Current literature is almost devoid of recognition of this problem, let alone providing any discussion of potential solutions. Researchers have built countless NL systems only to abandon them without public explanation. I will explain this phenomenon and present a practical solution.

2 NLP's Dirty Little Secret

Natural language processing (NLP) has long been concerned with NL parsing, disambiguation, and semantic representation, together collectively referred to as "understanding". However, 40 years of effort have failed to produce a good method for understanding NL. Examination of past NL projects, including my own DrEliza.com, has uncovered a hidden problem that doomed these projects right from their start – as the rules and relationships grow in numbers, complexity, and depth, the number of tests that fail undergo a combinatorial explosion that quickly limits the number of rules that can be honored within an acceptable processing time. This continues to happen despite the orders of magnitude improvement in processing speed that has become available in recent years. To illustrate, to achieve a 3-second response time to short open-ended NL queries falling far short of "commercially viable production performance" IBM's WATSON utilizes 2,880 POWER7 processors, making it the 94th fastest supercomputer in the world.

NL projects seem to go through a common development cycle. Researchers implement a demonstration, then the program slows nearly to a stop as they start adding rules on the way to making it useful enough to sell. Soon it becomes apparent that adding more rules is counterproductive because the program is already running too slowly to demonstrate in real time. This occurs before entering thousands of idioms, adding automatic spelling correction, or including other important pieces of a commercial quality system. Then, careful analysis of the programming usually uncovers clever ways of making the program run an order of magnitude faster, e.g. by moving language rules from a database into arrays. However, by this time it has become clear that an order of magnitude is not nearly enough additional speed to achieve the desired performance, so yet another NL project is "shelved".

Researchers have attempted to address this situation by making their rules smarter (and in some cases dumber; to do less, but faster), but this doesn't address the fundamental issue that failed tests produce no output, yet consume >99% of the processing time. Some projects have produced fast parsing by separating the semantic analysis into a separate module. However, the semantic analysis that follows is subjected not only to the same combinatorial explosion, but the explosion is made worse by having to

reassemble components of semantic units separated during parsing.

A parallel problem existed in the early days of computers, when programs had to wait for their own I/O. Buffered I/O eliminated this problem, and programs ran much faster.

3 The Concept

This article explores a new method which nearly eliminates the time costs of failed tests, thereby speeding up the parsing of NL by several orders of magnitude. This is accomplished by only performing rules whose least likely elements are present, and doing this in a way that incurs zero overhead for rules whose least likely elements are absent. This presents a new problem – of coordinating the complex process of parsing once most of the "structure" has been selectively eliminated. This problem is overcome by creating a new structure to queue the evaluation of surviving rules having least likely elements present, and to coordinate the order or evaluation, linking of rules, scope issues, etc.

4 Compared to Other Methods

Since I first proposed this method, various people have come forward with statements like "I thought that _____ (fill in the blank with their favorite parsing method) was fastest." Methods that involve parsing character strings, rather than first converting words to ordinals, have a built-in opportunity to run more than an order of magnitude faster, by simply converting words to ordinals and performing the same analysis using integer operations on ordinals that represent entire words. Some methods, like recursive ascent-descent parsing, can be directly emulated on this platform at speeds that are orders of magnitude faster. Some methods, like those using left associative grammars, discard semantic unit information to run fast. Semantic units must be subsequently reconstructed before parsing can be useful. This reconstruction requires performing the same tests as required by other methods, >99% of which can be eliminated using the method described below.

5 The Method

During initialization the first few thousand most commonly used words are processed into the lexicon, so that later, when rules are being compiled, it will be easy to compare words in rules to identify which are least frequently used.

All input is first broken into tokens representing a word, number, or punctuation. The word tokens are then hashed as part of being converted to ordinals representing their frequency of use, e.g. the most common word in English is "the", which would be represented as 1.

Stored in the lexicon is the hash for the words (used to resolve collisions), the word strings (used to create output), and a list of pointers to the rules for which that word is the least frequently used (LFU) word in the rule.

Rules contain a compiled representation of their operation, and information regarding their depth (to place them into the correct queue) and scope (to restrict analysis to the appropriate syntactical unit, e.g. a sentence).

During execution, words are converted to ordinals. In the process, pointers to the rules for which those words were those rules' LFU words are placed into appropriately prioritized queues that also hold the locations of the associated words. Pointers to lower-level rules are placed in higher-priority queues, to perform the lower-level rules first, before performing higher-level rules that were placed in lower-priority queues.

String operations, e.g. as used to parse common German run-on words, are performed on the strings in the lexicon during initialization to add new rules to existing lexicon entries to process the substrings. String operations can be performed during execution, which can be useful for handling invented words. However, substrings cannot be used as LFU triggers, although they can be used in rules that have been triggered by other words or rules.

The vast majority of low-level rules will never be performed in any given passage because their LFU words will not be present. Only the higher-level rules that were referenced by successfully-performing lower-level rules will be performed. Those few higher-level rules that are performed will often reference lower-level rules that haven't been performed. Rules that have not been performed can safely be presumed to be FALSE, because they must lack their least likely elements to not have been performed.

It is hard to guesstimate the speedup that this will provide, partly because there are numerous other methods with which to compare it, and partly because no one has seriously attempted to enter the rules to fully understand any language. However, the range seems to be somewhere around 3-4 orders of magnitude improvement in speed.

6 Path Toward Universal Acceptance

The next step is to propose a broadly usable and easily extensible representation standard for parsing, disambiguating, and other rules, which will be needed to construct a software platform on which to build many products that involve understanding NL.

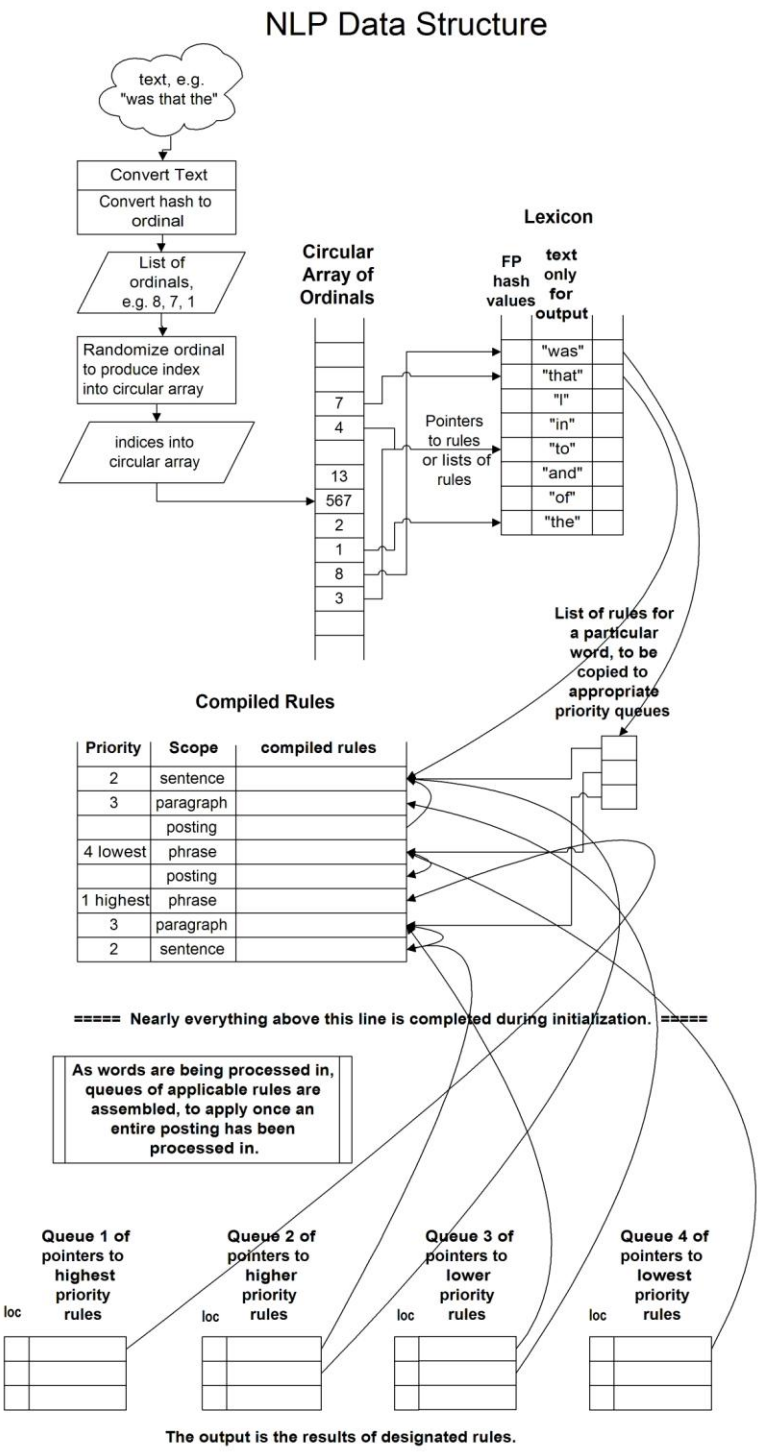
NLP Data Structure
text, e.g. "was that the"
Convert Text
Convert hash to ordinal
List of ordinals, e.g. 8, 7, 1
Randomize ordinal to produce index into circular array
indices into circular array
Circular Array of Ordinals
7
4
13
567
2
1
8
3
Lexicon
FP hash values
text only for output
"was"
"that"
"I"
"in"
"to"
"and"
"of"
"the"
Pointers to rules or lists of rules
List of rules for a particular word, to be copied to appropriate priority queues
Compiled Rules
Priority
Scope
compiled rules
2 sentence
3 paragraph
posting
4 lowest phrase
posting
1 highest phrase
3 paragraph
2 sentence
===== Nearly everything above this line is completed during initialization. =====
As words are being processed in, queues of applicable rules are assembled, to apply once an entire posting has been processed in.
Queue 1 of pointers to highest priority rules
Queue 2 of pointers to higher priority rules
Queue 3 of pointers to lower priority rules
Queue 4 of pointers to lowest priority rules
loc
The output is the results of designated rules.

Figure 1

However, by far the most labor intensive step lies beyond; to invest the large amount of linguistic work needed to precisely define the parsing of various world languages as they are written and spoken "in the wild".

6.1 Representation beyond BNF

Most of the thousands of rules would be written by linguists, not programmers. It would seem that a dual form of representation should be supported, where rules could be entered in either of two forms.

1. Grammatically correct sentences in any of several supported natural languages, inspired by the NL statement syntax used in COBOL.
2. Algebraic representation, which would be a major extension of Backus-Naur Form (BNF).

Syntax specification has previously been all about input, but also needed is a canonical way of specifying output to disparate applications, perhaps akin to that used in some meta-compilers.

6.2 Need for Coordination

There are several VERY different uses for NL parsing, each with its own special problems. For example DrEliza.com (which I wrote) has no provision for being able to output any of its own input, as there is no need for this in its application.

I have had experience writing commercial production compilers, source language optimizers, a linker, and worked in teams on computer-language related products, including the vectorizer and optimizer for CDC's supercomputer compiler. However, methods used to process computer languages are broadly inapplicable to efficiently processing NL.

The goal of automated language translation programs is to output (with suitable translation) EVERYTHING that is input, which is somewhat analogous to a compiler that targets a computer having a hyper-complex instruction set, but lacks operations that perform the precise functions of the operators in its input syntax.

Computers faced a similar challenge during the 1970s and 1980s. Thousands of computer programs in various computer languages running on 36-bit and longer CISC architectures, like IBM-7094 and Burroughs 6000 processors, were being converted to different computer languages running on simpler 32-bit architectures like IBM-360 and Intel processors that cost far less because they were made with far fewer transistors. Now, transistors are essentially free, so long word length CISC architectures should be revisited. That is another story for another paper. A number of automated translation tools were developed that operated similarly to tools Microsoft later offered to assist in converting programs to their simplified .NET platform. These translation programs left a lot of residual problems, not unlike those left by modern day automated language translation programs. Computers are MUCH less flexible in interpreting their programs than people are in reading text, so a higher state of perfection was needed in translating computer programs compared to translating NL.

To succeed, experienced automated language translation personnel would of necessity be participating in any effort to design a universal NL platform.

A working group is needed to define the syntax and operation of parsing rules for both input and output that will work for everyone, populated by people with sufficiently disparate backgrounds. The goal would be to produce a "drop in" module that will work for any NL application, ranging from problem solving and question answering, to automated language translation.

This module would support application-specific rules, backed up by a vast library of rules to take care of the myriad details of parsing NL, e.g. how to handle thousands of idioms, each with their own peculiarities. Using the method described above, including multiple languages in the library would have a negligible affect on performance, e.g. only where there are multiple identical words in different languages that have different meanings but which may be used together. Language-related disambiguation would be necessary only in these rare cases. This is so rare that I know of no such example to cite here, so the cost in time would be negligible.

6.3 Placement/Payload Theory

It is my theory that computerized speech and written understanding has eluded developers for the past ~40 years in part because of a lack of a fundamental understanding of the task, which turns out to be very similar to patent classification.

When classifying a patent, successive layers of sub-classification are established, until only unique details distinguish one patent from another in the bottom-level subclass. When reviewing the sub-classifications that a particular patent is filed within, combined with the patent's title, the subject matter usually becomes apparent to anyone skilled in the art.

However, when a patent is filed into a different patent filing system, e.g. filed in a different country where the sub-classifications may be quite different, it may be possible that the claims overlap the claims of other patents; and/or unclaimed disclosure would be patentable in a different country.

Similarly, when you speak or write, in your own mind most of your words are there to place a particular "payload" of information into its proper context, much as patent disclosures place claims into the state of an art. However, your listeners or readers may have a very different context in which to file your words. They must pick and choose from your words in an effort to place some of your words into their own context. What they see as the "payload" may not even be the payload you intended, but may be words you only meant for placement. Where no placement seems possible, they might simply ignore your words and file **you** as being ignorant or deranged.

The expressed relationship between various placement and payload words carries the author's point of view. For some applications, like automated language translation, this may be important to extract and preserve, while it is best ignored when solving problems, except when containing statements of ignorance. For example, take the statement "***I have a headache because I got drunk last night.***" The facts are "***I have a headache***" and "***I got drunk last night***". There is also a suspicious "***because***" relationship between those two facts. Most headaches are secondary to dehydration, especially those related to prior alcohol consumption. Hence, the primary reason the author of this statement has a headache is probably because he has not consumed enough water, and **not** because of the lesser contributing factor of having gotten drunk last night. In this context, "***because***" becomes a statement of ignorance and hence potential payload.

Many teachers have recorded a classroom presentation and transcribed the recording, only to be surprised to learn that what they actually said was sometimes the opposite of what they meant to say. Somehow the class understood what they meant to say, even though their statement was quite flawed. When you look at these situations, the placement words were adequate, though imperfect, but the payload was okay. Indeed, if another person's world model is nearly identical to yours, very few placement words are needed, and so these words are often omitted in casual speech which complicates translation.

These omitted words fracture the structure of about half of all sentences "in the wild", often rendering computerized parsing impossible. If a computer program first identifies prospective payloads, and then looks for nearby placement information while ignoring things it can't deal with, then fractured sentences only cause difficulty when the fractures are critically located.

As people speak or write to a computer, the computer must necessarily have a **very** different point of view to even be useful. The computer must be able to address issues that you cannot successfully address yourself, so its knowledge must necessarily exceed your own in its subject domain. This leads to some curious conclusions:

1. Some of your placement words will probably be interpreted as "statements of ignorance" by the computer, and so be processed as valuable payload, to trigger an appropriate response to teach you something you clearly do not know.
2. Some of your placement words will probably refer to things outside of the computer's domain knowledge, and so must be ignored, other than being recognized as non-understandable restrictions on the payload, that may itself be impossible to utilize.
3. Some of your intended "payload" words will serve as placement.

DrEliza's application seeks to intercept words written to someone who presumably has substantial common domain knowledge. Further, the computer seeks to compose human-appearing responses, despite its necessarily different point of view and lack of original domain knowledge. While this is simply not possible for the vast majority of writings, DrEliza can simply ignore everything that it is unable to usefully respond to.

If you speak a foreign language, especially if you don't speak it well, you will immediately recognize this situation as being all too common when listening to others with greater language skills than your own speaking among themselves. The best you can do is to quietly listen until some point in the conversation when you understand enough of what is being said and you are able to add something useful to the conversation.

Note the similarity to advertising within present (2013) Google Mail, where advertisements are selected based upon the content of email. If Google's computers were to perform a deeper analysis they could probably eliminate ~99% of the ads as not relating to users' needs and greatly improve users' experience, and customize the remaining 1% of the ads to precisely target users' interests.

That is very much the goal in my application. The computer knows about certain products and solutions to common

problems, etc., and scans the vastness of the Internet to find people whose words have stated or implied a need for things in the computer's knowledge base, and have done so in terms that the computer can "understand".

6.4 Representation Implications

When advanced compilers, especially optimizing and vectorizing compilers for supercomputers, compile computer programs to executable code, they make no attempt to translate program statements one-at-a-time. Instead, they build a diagram of the entire program, simplify and otherwise improve the diagram, and then translate the diagram to executable code. The same could be done with human speech and writing, in which case the output order would often be rearranged from the input order, in ways where statements would be clear and direct. Outputting in the same language, such a program might make a good automated editing program.

Several complex NL understanding project proposals have incorporated some sort of an a priori world model, onto which they plan to hang information gained from their input. Often the world model is simply built into the ontological information about individual words. It is unclear whether such a priori structures are a help or a hindrance, especially if there is some efficient way to find related statements without committing to file facts into a particular knowledge structure.

Foregoing an a priori world model requires a simple representation in which to state and store all of human knowledge. Presuming the placement/payload theory is correct, statements would be represented as groups of information fragments, some placement, and some payload, originally depending on what the author already knows. The computer would have the job of processing and/or outputting these fragments of information, without knowing which was placement and which was payload.

Most applications, from query to translation, require some way of accessing statements relating to particular collections of information fragments. One can envision contorted table structures and recursive descent searching algorithms to find the statements that best address particular subjects, but it would be preferable to simply have an efficient database tool like SQL do the job for us.

6.5 An Issue with SQL

SQL is my favorite AI programming language, because I can do really complex information accessing with single statements. However, it is powerless to access records that have particular contents, where those contents could appear in any field of a record.

Present SQL products are unable to "wildcard" subfields in an index, both in the records themselves and in the **`SELECT`** statements that access records. Complex kludged workarounds, e.g. using string operators in **`SELECT`** statements, indirectly force the SQL engines to perform sequential searching. These kludges slow things down so much that they are only practical for small demos. Other kludges in effect overlay wildcard accessing over conventional fully-specified keyed accessing, resulting in slow programs that are cluttered with the code needed to make this work.

Some SQL products have sophisticated string search capabilities, but are slowed down by their string operation, when AI programs work better with ordinals instead of strings.

Some future advanced NL understanding projects may have to wait for an SQL-like product that supports some sort of new capability to index through groups of information fragments.

This facility will never appear in SQL until someone specifies a particular new SQL capability that would facilitate these applications. Then, someone can add the new capability to one of the shareware SQL products, so that NL understanding can proceed without this challenge hanging over it.

7 Conclusion

Now that you understand the "logic" that has misguided so many NL understanding projects onto the scrap heap, you can quickly recognize it when you see it again in the future, and explain the pitfall that awaits such efforts. The answers to simple questions like "How are you going to represent words?" and "What triggers the evaluation of a rule?" will usually tell you if the projects are on the wrong track.

This new method promises several orders of magnitude improvement in the speed of NL understanding, regardless of which model of language is being used. This method has its own characteristic strengths and weaknesses around which a robust rules compiler could compile rules suitable for just about any imaginable approach to NL understanding.

However, it is one thing to "understand" NL, and quite another to usefully manipulate it, e.g. mine it for knowledge or translate it to a foreign language. Some sort of new database capability appears to be needed to replace

present ad hoc methods with high-level query-driven database solutions.

If this technology and associated descriptions of common languages are to be shared then some standards are necessary. If you would like to participate in developing a robust representation and interface to support your own needs for NL understanding, or if you just want some tables (like the 10,000 most commonly used English words in order of frequency of use), then please contact me at:

Steve.Richfield@gmail.com

8 The Future

It is hoped that this methodology will affect the world in three important ways:

1. People will stop writing NL understanding code that has no real possibility of ever scaling up to a useful level of functionality.
2. The Internet in general, and Google in particular, will shed its dependence on isolated word and n-gram recognition for web searching and advertisement triggering, and shift to looking for statements having specific meanings.
3. Future advertising engines that will watch the Internet for problem statements to trigger advertisements; will also be able to watch the Internet for problems statements relating to health, maintenance, politics, and other interesting domains. Then, precisely targeted responses can be produced to convey key knowledge, to finally achieve the goal of having an ***Intelligent Internet***.

9 References

[1] U.S. Patent Application 13/836,678.