A Time-Efficient Token Representation for Parsers

Sébastien Paumier

A time-efficient token representation for parsers

Sébastien Paumier
IGM, University of Marne-la-Vallée,
5 boulevard Descartes,
77454, Champs-sur-Marne, France
paumier@univ-mlv.fr

Abstract

One of the most important functions of linguistic tools is to apply grammars to texts in order to find matching sequences. These grammars are often represented by finite-state automata. The expressions described in these grammars are usually lexical units, but some systems offer the possibility of dealing with references to sets of words which require the use of electronic dictionaries (ex. <N> for nouns). We present here a method for applying such grammars rapidly, which is based on the tokenization of both grammars and texts and on a representation of these tokens by integers.

Introduction

Linguistic parsers can be divided into two categories: those which use approximate descriptions, like statistical parsers (Collins 1999, Plaehn 1999), and those that use exact descriptions. Exact parsers often process texts that have been tagged with Part-Of-Speech taggers (Luk 2001, Megyesi 2002). The tag sets are usually small. However, the behaviour of parsers is not well known in the context of large grammars, because few people have actually tried to use such grammars. Grammars inevitably grow considerably when we want to refine large-coverage descriptions. In fact, the complexity of the grammar grows with the size of the vocabulary, because of local constraints that can only be dealt with by systematic descriptions. These descriptions require the use of finer tags than those used by POS taggers; tags such as *human nouns*, *collective nouns*, etc. The

experiments that have been carried out (Carvalho 2001, Constant 2000 & 2002, Dister 2000, Domingues 2001, Fairon 2000, Gross 1997, 1998) show that these grammars must refer both to lexical units (single words) and to sets of words, that are usually handled by dictionary lookup. If grammars are fine enough, the difference between advanced pattern matching and syntactic parsing becomes superficial: the fact that the grammars describe phrases or sentences is a matter of scale.

Linguistic pattern matching is a particular case of pattern matching, because the base unit is the word rather than the character. It is common to consider texts as sequences of such tokens, for example in indexation processes, but most matching algorithms do not tokenize their input in a previous, separate pass. We propose here an original way of parsing, which is based on the coded representation of tokenized corpora.

The idea is to represent lexical units by integers, which are easier to manipulate than strings. This representation of tokens by integers has been used in computer science for a long time, but it has not been fully exploited in the field of linguistic parsers. This technique is well-known in compilation: the lexical parser analyses sequences of characters and produces sequences of symbols that are used as input to the syntactic parser (Aho 1986). However, methods differ, because the grammars and vocabularies are much smaller with compilation. In addition, users of linguistic grammars often modify them for their experiments.

We will show that the integerised technique can

yet be applied very successfully in the domain of linguistic pattern matching.

Our parsing method has 3 stages:
- tokenizing the text;
- optimizing the grammar for the text;
- applying the grammar.

First, we will present our tokenization rules. Then we will describe the transformations to be made on the grammar, taking into account the possible use of lexical resources. Finally, we will discuss operational complexities, focusing on some linguistic aspects in order to show that this method is well adapted to linguistic pattern matching.

1 Tokenizing texts

During the first stage, we tokenize the text into lexical units. To do this, we must take a formal definition of a lexical unit[1]. For languages with well delimited words, we can consider that a lexical unit can be:
- a sequence of letters[2];
- a non-alphabetic character.

If the tokenization is case sensitive, we can then associate a unique integer with each token. If we replace every token by its number, the text can be represented by the token list and the sequence of integers. Note that this process need only be executed once, even if we want to apply several grammars to the text. The process is proportional to the length of the text.

Results show that the two files produced occupy approximately the same disk space as the original

---

text file. As our goal is to speed up the parsing, we always give preference to time optimization over space complexity as long as there is no explosion.

2 Optimizing the grammar

The second stage consists in optimizing the grammar according to the text. We will assume that the grammar is represented by a finite-state automaton, whose transition tags are lexical units that conform to the tokenization rule used during the first stage.

2.1 Replacing lexical units by integers

First, we replace every transition tagged by a lexical unit by the exhaustive list of the tokens that it can match.

One lexical unit can match more than one token in the text because of case variation. In fact, it is very natural to look by default for all the case variants of a word when you search it. For example, if you look for the word *doctor*, you may want to match both sequences *doctor* and *Doctor*. For some applications, the variations allowed can be more permissive: for example, web search engines usually do not take accents into account. Thus, we can assume that some variation will typically be allowed.

A particular lexical unit in the grammar may not be matched in the text. By optimizing the grammar according to the text, we can remove such transitions. In fact, if we construct for each lexical unit in the grammar the exhaustive list of the lexical units of the text that can match, we can find out which lexical units can match nothing by checking if the list is empty. In that case, we can remove from the grammar every transition tagged by this lexical unit.

Finally, for all remaining transitions, we replace the original lexical unit by the list of the tokens it can match. The tokens are represented by their associated integers that have been computed during the text tokenization.

2.2 References to word classes

2.2.1 Word classes

Some linguistic tools (not only parsers) allow the use of lexical resources (Silberztein 1993, Bernard 2001, Jassem 1997). In general, the word classes in these resources are referenced in grammars through patterns. These patterns can be more or less precise :

- any word that is in the dictionary;
- any word of a particular grammatical category (any verb, any noun, ...);
- any inflected form of a word.

This list is not exhaustive, and we can imagine many kinds of patterns for many kinds of applications and dictionaries. The important point is that these patterns always define word classes.

These classes can be more or less important, depending on the pattern precision. The pattern "any word in the dictionary" recognizes many more words than "any noun in the feminine singular", which itself recognizes many more words than "any inflected form of the verb *to eat*".

Checking on the fly if a word in the text verifies a pattern or not has two disadvantages. First, it involves accessing the dictionary, which means reloading it when needed or keeping it in memory. The other point is that it makes the grammar non deterministic, because several patterns could recognize the same word.

It would also cost considerable space to replace systematically the pattern of a transition by the list of all the matching tokens in the case of a low-precision pattern like "any word in the dictionary". We have thus devised the following intermediate solution.

2.2.2 A class-size based solution

We propose to test the pattern precision by counting the matching tokens. If this number is lower than a certain limit, we can replace the pattern by the token list. Otherwise, we give a number to this pattern and we record the fact that every token in the list is recognized by this pattern. This can be done very efficiently by using a bit array for each token, setting to 1 the nth bit if the token verifies the nth pattern. As tokens are represented by integers, checking if a token matches a pattern can be done by simple access to a two-dimensional bit array, indexed by the token number and the pattern number.

As such an array is often very sparse, it can be greatly compressed, for example in terms of chained lists.
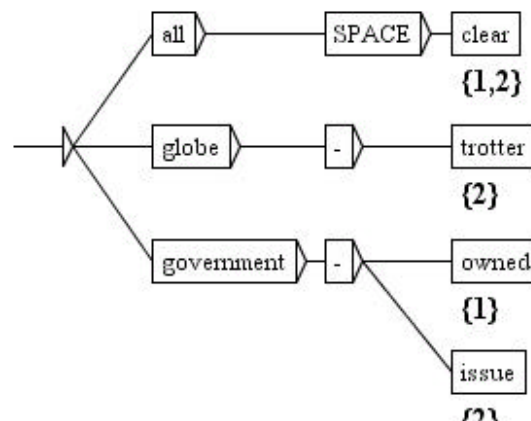
2.2.3 Compound words

This solution concerns only single tokens, but we can extend it to handle compound word resources. First, we check for each compound word all the patterns that can match it. Then we construct the list of all the compound words that are matched by one pattern at least; for example:

- *all clear*: matched as adjective and noun
- *globe-trotter*: matched as noun
- *government-owned*: matched as adjective
- *governmen-issue*: matched as noun

From this list, we can build a tree of which the nodes are the tokens that compose the compound words. On each terminal node of the tree, we store the numbers of the patterns that match the corresponding compound word. The tree constructed from the previous example list is shown on figure 1 (the numbers 1 and 2 that appear inside round brackets represent respectively the patterns "adjective" and "noun").

We add to this tree all the case variants of each token. Then we replace the tokens by their number and we arrive at a data structure ready for speedy comparison with the text.

Figure 1: example of token tree

Now we can test whether the pattern that has the number i matches one or more compound words in the text by exploring simultaneously the token tree and the sequence of integers in the text representation. If we reach a final node in the tree, we just have to check if i is in the list of the patterns that match the corresponding compound word.

2.3   Sorting integer lists

Once we have made all these replacements, there are two kinds of transitions : integer lists representing tokens, and patterns representing word classes. To explore a state of the grammar automaton, we must test both pattern transitions and token transitions.

Because of the non-determinism that they introduce, we must test every pattern transition and so, to do this, we make a list of them for each state.

On the other hand, token transitions can be merged into one integer list. The fact that each integer represents a unique token implies that, at the most, only one integer of the list can match with the one in the representation of the text. Checking if a token transition can match the text token is equivalent to testing if an integer appears in an integer list. To speed up this test, we turn the integer list into a sorted integer array, so that we can achieve very fast lookups by dichotomy.

3   Gain

3.1   Dictionary accesses

If a grammar contains word classes, we have shown that these references can be tested with a simple array access in the case of simple words.

In the case of compound words, we obtain a complexity of $O(n)$, where n is the number of tokens that compose the compound word to be matched [3] . If we test references through a dictionary lookup, this operation will have a minimal complexity of $O(m)$, where m is the number of letters in the word to be matched. This minimal complexity is always greater than $O(n)$, so we can say that our method tests references to word classes faster.

3.2   Token comparisons

We will demonstrate the importance of representing token transitions by sorted integer arrays in comparison with string lists and character automata. Let us consider the simplest case of a grammar composed of a single word list, such the grammar in Figure 2.
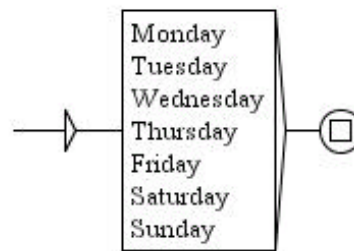

Figure 2: a simple grammar

To simplify comparisons, we will assume that these words have no case variants. Variants actually found in texts are mainly words that begin with an upper-case letter. As that phenomenon only concerns few words, we can

---

[3] We can note that, in our whole method, this n is the only value that has something to do with the length of words.

consider that ignoring variants has little effect on the length of the list.

Following our method, we obtain the complexity of a dichotomy research in a sorted array, which is $1+log(n)$, where n is the array size (here, n is the number of words in the list). The constant 1 is due to the fact that the word can be absent from the array.

Consider now the representation by a string list. If this list is not sorted, we obtain a complexity of $m*n$. If the list has been sorted, we can have the $log(n)$ complexity of the dichotomy search. However, the comparison function has a non-constant complexity because it compares strings, and so it is less efficient than an integer comparison. We can conclude that this representation is not as efficient as ours.

Let us consider now a representation by a deterministic character automaton, in which transitions are sorted so that they can be explored by dichotomy. The presence of a word with a length of m in such a deterministic automaton can be tested with a complexity of $m*(1+log(t))$, where t is the average number of transitions per state. m, n and t are independent values. To compare the values $1+log(n)$ and $m*(1+log(t))$, we must look at linguistic facts.

3.3   Linguistic aspect

If the number n of words in the list is lower than a certain limit which depends on the length of these n words, typically 50 words, the relation $log(n) < O(m*log(t))$ is verified because $log(n)$ is lower than m (the number of letters of the word we want to test), which means that for small word lists, our method is more efficient.

When n grows, the value $O(m*log(t))$ is quite impossible to calculate, because the number of transitions per state depends on the words that are in the list. However, we can make the following estimate.

The $log(n)$ value can be limited by the logarithm of the number of words of the language. If we took one million as a rough approximation, we would obtain $log(1000000) \sim 20$. In fact, the number of words in a list is much lower than that for a linguistic reason: words do not appear randomly in the same list. The aim of a linguistic grammar is to describe a consistent set of rules, and so if some words appear in the same list, that means that they are linked (same grammatical category, by synonymy, etc). This phenomenon is illustrated by the grammar of Figure 2, in which "day" names constitute an evident word class. If the members of a class are too numerous (for example all the nouns), they might not appear as a list in the grammar, but as entries in a dictionary. In practice, most of these lists contain fewer than 10 items. The biggest lists we have found in grammars are made of proper nouns, like country names. Even these lists do not contain more than 500 items. In this way, we can consider that $log(n)$ is usually smaller than 7, and that 10 is a reasonable upper limit for $1+log(n)$.

As the m value considered in $m*(1+log(t))$ is the length of the word in the text to be matched, we can consider that this value is close to the average length of words in a text. As an example, we have computed that the average length of a French word in a text is 5.7 letters. This value has been obtained through the examination of one year of the daily French newspaper *Le monde* for the year 1994[4].

If we consider that n is great enough, we can assume that the value $1+log(t)$ will be greater than 1. In that case, the value $m*(1+log(t))$ will surely be greater than 10, which is a reasonable maximum value for $1+log(n)$.

While this calculation is just an estimate, it gives an idea of what could happen for word lists that are large enough. We have empirically verified that our method is faster than parsing algorithms that use character automata. We have applied several grammars containing word lists of various

---

[4] This represents 120 Megabytes of raw text.

sizes. Tests have been made on both English and French data:

- numeric expressions in French (Constant 2000);
- verbal sequences in English (Gross 1998);
- noun phrases in the economic domain in French.

The experiments show that our method is always faster. The acceleration factor depends very much on the grammar structure, but in the most favourable cases, our method was 600 faster than using character automaton and consulting dictionaries on the fly. Empirical tests thus confirm the intuition arrived at in the previous estimate (Paumier 2000).

3.4 Cost of pre-processing

The experiments show that the parsing is faster when it is based on a representation of lexical units by integers. However, we must take into account the complexity of the pre-processing operations, which are tokenizing the text and optimizing the grammar.

The text tokenization has a linear complexity depending on the text length. In practice, the cost of this operation can be ignored when grammars are non-trivial, because the time that is required to tokenize the text is negligible in relation to the time necessary to parse it. As linguistic grammars are almost always non-trivial, we can consider that the complexity of this operation does not affect the global performance of the method.

The optimization of the grammar requires the replacement of every token in the grammar by its equivalent integer list. The complexity of this step is $O(N)$, where N is the size of the grammar alphabet. This operation is negligible compared with the parsing complexity. If the grammar contains references to dictionaries, we must access these dictionaries, which is an operation with a complexity that depends on the dictionary representation. If this complexity is important, this stage in the grammar optimization may be

critical.

If the grammar is fully explored during the parsing, then all the references to dictionaries must be computed. In that case, there is no difference between our method and an on the fly dictionary exploration: whatever complexity this operation has, it is unavoidable.

On the other hand, if the grammar is not explored at all, no dictionary access will have been done with an on the fly algorithm, while these operations are done anyway with by method. The critical case is thus when we apply a grammar that has very few chances of matching in the text. From a linguistic point of view, this case is quite rare, because there is no sense in applying a grammar to a text that has nothing to do with the expressions described by this grammar. No one would apply a grammar of French noun phrases to an English corpus, or an electronic term grammar to a poetic text.

3.5 Global performances

As very few parsers allow dictionary lookups, our experiments were done comparing our own program with Intex software. After discussion in 2000 with Max Silberztein, the author of this software, we have concluded that the Intex parsing algorithm was at that time roughly equivalent to character automaton parsing.

In practice, we have measured the sum of pre-processing time and parsing time and established that it is lower than the parsing time of Intex when grammars are non trivial. Table 1 presents some results obtained on a Pentium 500 MHz with a RAM of 128 Mb. The numeric expression grammar contains very few dictionary references, whereas there are many in the verbal sequences and about 20 in the noun phrases.

Our experiments show that this method is well adapted to linguistic pattern matching. However, the gain is very important only when grammars are large enough. Our method is not very efficient when applied to small scale data.

	Numeric expressions	Verbal sequences	Economic noun phrases
States	107 000	250 000	587
Transitions	703 000	2 000 000	3596
Corpus size	0.4 Mb	16 Mb	120 Mb
Time with Intex	20 min	Parsing failure	9 min
Time with our program[5]	22 sec	3 min 37 s	1 min 11 s

Table 1: experiment results

Conclusion

We have proposed a way of optimizing text parsers that use lexical resources such as electronic dictionaries. This method is based on word tokenization, and on a representation whereby tokens are coded by integers.

We have established that the gain obtained grows with the size of grammars. Our method is thus a specific optimization for linguistic parsers that use large grammars. The current development of grammars and electronic resources in many fields of computational linguistics make us think that such optimizations will be very useful.

References

Aho Alfred V., Ravi Sethi and Jeffrey D. Ullman (1986) *Compilers. Principles, Techniques and Tools,* Addison-Wesley

Balvet Antonio (2000) *Approches catégoriques et non catégoriques en linguistique des corpus spécialisés, application à un système de filtrage d'information*. Thèse de doctorat. Université Paris 10

Bernard Pascale et al. (2001) *Ressources linguistiques de l'ATILF.* Actes du 8e colloque TALN, Tours

Carvalho Paula (2001) *Grammaires de levée d'ambiguïtés entre noms et adjectifs.* Lingvisticæ Investigationes, 24:1, pp. 127—145.

Collins Michael et al. (1999) *A statistical parser for Czech*. In ACL 99 Proceedings

Constant Matthieu (2000) *Description d'expressions numériques en français*. *In Anne Dister (ed.), Actes des 3e Journées Intex,* Revue Informatique et Statistique dans les Sciences Humaines, n° 1 à 4

Constant Matthieu (2002) *On the Analysis of Locative Phrases with Graphs and Lexicon-Grammar: The Classifier/Proper Noun Pairing. In proceedings of* PorTAL 2002: 33-42

Dister Anne, ed. (2000) Actes des 3èmes Journées INTEX. Revue Informatique et Statistique dans les Sciences Humaines, 36ème année, n° 1 à 4

Domingues Catherine (2001) *Étude d'outils informatiques et linguistiques pour l'aide à la recherche automatique d'information dans un corpus documentaire*. Thèse de doctorat. Université de Marne-la-Vallée

Fairon Cédrick (2000) *Structures non-connexes. Grammaires des incises en français : description linguistique et outils informatiques*. Thèse de doctorat. Université Paris 7

Gross Maurice (1997) *The Construction of Local Grammars.* In E. Roche and Y. Schabes (eds.), Finite-state language processing. MIT Press, pp. 329—354.

Gross Maurice (1998) *Lemmatization of compound tenses in English.* Lingvisticæ Investigationes, 22, pp. 71—122.

Jassem Krzysztof (1997) *A Polish-to-English Text-to-text Translation System Based on an Electronic Dictionary.* In proceedings of ACL Workshop 1997

---

[5] An implementation of this method can be found in the GPL licensed software Unitex, available at `http://www-igm.univ-mlv.fr/~unitex`

Luk Po Chui, Weng Fuliang and Meng Helen (2001) *Automatic Grammar Partitioning for Syntactic Parsing*. Proceedings of the International Workshop on Parsing Technologies, Beijing

Megyesi Beata (2002) Shallow Parsing with PoS Taggers and Linguistic Features. Journal of Machine Learning Research, 2 :639-668

Paumier Sébastien (2000) Reconnaissnce d'expressions dans de grands corpus: le système AGLAE. *Mémoire de DEA*, Université de Marne-la-Vallée

Plaehn Oliver (1999) *Probabilistic Parsing with Discontinuous Phrase Structure Grammars.* PhD Thesis, Saarland University

Silberztein Max (1993) *Dictionnaires électroniques et analyse automatique des textes. Le système INTEX*, Paris, Masson