A Probabilistic Chart Parser implemented with an Evolutionary Algorithm *

L. Araujo

Dpto. Sistemas Informáticos y Programación. Universidad Complutense de Madrid. Spain. lurdes@sip.ucm.es

Abstract. Classic parsing methods use complete search techniques to find the different interpretations of a sentence. However, the size of the search space increases exponentially with the length of the sentence or text to be parsed and the size of the grammar, so that exhaustive search methods can fail to reach a solution in a reasonable time. Nevertheless, large problems can be solved approximately by some kind of stochastic techniques, which do not guarantee the optimum value, but allow adjusting the probability of error by increasing the number of points explored. Evolutionary Algorithms are among such techniques. This paper presents a stochastic chart parser based on an evolutionary algorithm which works with a population of partial parsings. The paper describes the relationships between the elements of a classic chart parser and those of the evolutionary algorithm. The model has been implemented, and the results obtained for texts extracted from the Susanne corpus are presented.

keywords: Evolutionary programming, Partial Parsing, Probabilistic Grammars

1 Introduction

Parsing a sentence can be sought as a procedure that searches for different ways of combining grammatical rules to find a combination which could be the structure of the sentence. A bottom-up parser starts with the sequence of lexical classes of the words and its basic operation is to take a sequence of symbols to match it to the right-hand side of the rules. Thus, this parser can be implemented simply as a search procedure for this matching process. However, such implementation would be extremely expensive because the parser would try the same matches again and again. This problem is avoided in the chart parsing algorithms by introducing a data structure called *chart* [1]. This structure stores the partial results of the matchings already done.

Classical parsing methods are based on complete search techniques to find the different interpretations of a sentence. However, experiments on human parsing suggest that people do not perform a complete search of the grammar while

* Supported by projects TIC2003-09481-C04 and 07T/0030/2003.

parsing. On the contrary, human parsing seems to be closer to a heuristic process with some random component. This suggests exploring alternative search methods. Another central point when parsing is the need of selecting the "most" correct parsing from the multitude of possible parsings consistent with the grammar. In such a situation, some kind of disambiguation is required. Statistical parsing provides a way of dealing with disambiguation. Stochastic grammars [4], obtained by supplementing the elements of algebraic grammars with probabilities, represent an important part of the statistical methods in computational linguistics and have allowed important advances in areas such as disambiguation and error correction. A probabilistic context free grammar, PCFG, is defined as a CFG along with a set of probabilities on the rules such that

$$\sum_j P(N^i \rightarrow \eta^j) = 1$$

for all i, where N^i is a nonterminal and η^j is a sequence of terminals and nonterminals.

In order to improve the efficiency of a parser based on a PCFG, we can develop algorithms that attempt to explore the high-probability components first. These are called *best-first parsing* algorithms. The goal is to find the best parse quickly and thus to avoid exploring much of the search space. Chart parsing algorithms can be easily modified to consider the most likely components first.

Another alternative to search the parses are evolutionary algorithms (EAs). EAs have already been applied to some issues of natural language processing [6], such as query translation [8], inference of context-free grammars [11, 10, 7, 5], tagging [3] and parsing [2]. In the system described in [2] parse trees are randomly generated and combined. The evolutionary process is in charge of giving low rates of probability of surviving to those trees which do not match the grammar rules properly. This system has been tested on a set of simple sentences, but the size of the population required to parse real sentences with real grammars, as those extracted from a linguistic corpus, is too large for the system to work properly.

This paper presents the implementation of a stochastic bottom-up chart parser based on an evolutionary algorithm which works with a population of partial parsings. The algorithm produces successive generations of individuals, computing their quality or "fitness" at each step and selecting the best of them for the next generation. The purpose of most EAs is to find a good solution and not necessarily the best solution, and this is enough for most natural language statistical processes. EAs provide at the same time a reasonable accuracy as well as a unified scheme of algorithm applicable to different problems.

The rest of the paper proceeds as follows: Section 2 describes the evolutionary parser, presenting the main elements of the evolutionary algorithm; section 3 presents and discusses the experimental results, and section 4 draws the main conclusions of this work.

```
function Arc-Extension(chart, agenda, grammar, C, p1, p2)
    Insert(chart, C, p1, p2);
    for each active arc X → X1, ···, ∘C, Xn from p0 to p1 do{
        AddNewActiveArc(chart, X → X1, ···, C ∘ Xn, p0, p2);
    }
    for each active arc X → X1, ···, Xn ∘ C from p0 to p1 do{
        AddNewComponent(agenda, X, p0, p2);
    }
end
```

Fig. 1. Arc extension algorithm to add a component from position p1 to position p2

2 Evolutionary Algorithm for Chart Parsing

The algorithm can be view as a probabilistic implementation of a bottom-up chart parser. In a chart parser, the *chart* structure stores the partial results of the matchings already done. Matches are always attempted from one component, called *key*. To find rules that match a string involving the key, the algorithm looks for rules which start with the key, or for rules which have already been started by early keys and require the present key either to extend or to complete the rule. The chart records all components derived from the sentence so far in the parse. It also maintains the record of rules that have partially matched but are incomplete. These are called *active arcs*. The basic operation of a chart parser consists in combining an active arc with a completed component. The result is either a new completed component or a new active arc that is an extension of the original active arc. Completed components are stored in a list called *agenda* until being added to the chart. This process is called *arc extension algorithm*, of which Figure 1 shows an scheme. To add a component C into the chart from position p_1 to position p_2, C is inserted into the chart between those positions. Then, for any active arc of the form $X \rightarrow X_1, \cdots, \circ C, X_n$ (where $\circ$ denotes the key position) from p_0 to p_1, a new active arc $X \rightarrow X_1, \cdots, C \circ X_n$ is added from position p_0 to p_2. Finally, for each active arc $X \rightarrow X_1, \cdots, X_n \circ C$ from position p_0 to p_1, which only requires C to be completed, a new component of type X is added to the agenda from position p_0 to p_1. Figure 2 shows a scheme of the chart parsing algorithm. It consists in a loop repeated until there is no input left. At each iteration, if the agenda is empty, the lexical categories for the next word of the sentence are added to the agenda. Then a component C is selected from the agenda. Let us assume it goes from position p_1 to p_2. For each grammar rule of the form $X \rightarrow CX_1, \cdots, X_n$, a new active arc $X \rightarrow \circ CX_1, \cdots, X_n$ from p_1 to p_2 is added from position p_1 to p_2. Finally, C is added to the chart by means of the arc extension algorithm.

Probabilistic Chart parsing algorithms consider the most likely components first. The main idea is to implement the agenda as a *priority queue* — where the highest-rate elements are always first in the queue. Accordingly, the parser always removes the highest-ranked component from the agenda and adds it to the chart.

```
function ChartParser(sentence, chart)
    while cont < Length(sentence) do{
    if Empty(agenda) {
        AddInterpretation(sentence[cont]);
        cont++;
    }
    SelectComponent(agenda, C, p1, p2);
    for each grammar rule X → CX1 ··· Xn from p1 to p2do{
        AddNewActiveArc(chart, X → ∘CX1 ··· Xn, p1, p2);
    }
    ArcExtension(chart, agenda, grammar, C, p1, p2);
end
```

Fig. 2. Bottom-up chart parsing algorithm

In the evolutionary parser (EP), the population is composed of partial parses of different sequences of words of the sentence, that are randomly combined to produce new parses for longer sequences of words. Thus, the *agenda* of a chart parser, which stores completed components, is represented by the population of the EA. The algorithm of arc extension is represented by the crossover operator, which combines partial parses until completing the categories requires by the right-hand side of a rule, so as to produce a new completed component of the classes given by the left-hand side of the rule. At this point, the EP differs of the chart parsing because the arc extension algorithm is continuously applied until completing a component. Thus, in the EA active arcs only exist during the crossover operation, because when this finishes every arc explored has been completed. Another difference comes from the way of selecting the rules to be applied. A *best-first parsing* algorithm always selects the most likely rule, while the EA, can select any rule, though those of higher probability have more chances to be selected.

Finally, the input data for the EA are those of the classic chart parser: the sentence to be parsed, the dictionary from which the lexical tags of the words can be obtained along with their frequencies, and the PCFG, together with the genetic parameters (population size, crossover and mutation rates, etc.).

Let us now consider each element of the algorithm separately.

2.1 Chart Representation: The EA Population

The chart data structure stores all intermediate results of the parsing, that is, any valid parse of the substrings of the input. This intermediate results are the *edges* of the chart. Thus, each edge stores three things: the grammar rule, the parse subtree and the corresponding locations in the sentence.

Individuals in our system can be view as edges, directly represented by subtrees. They are parses of segments of the sentence, that is, they are trees obtained by applying the probabilistic CFG to a sequence of words of the sentence. Each

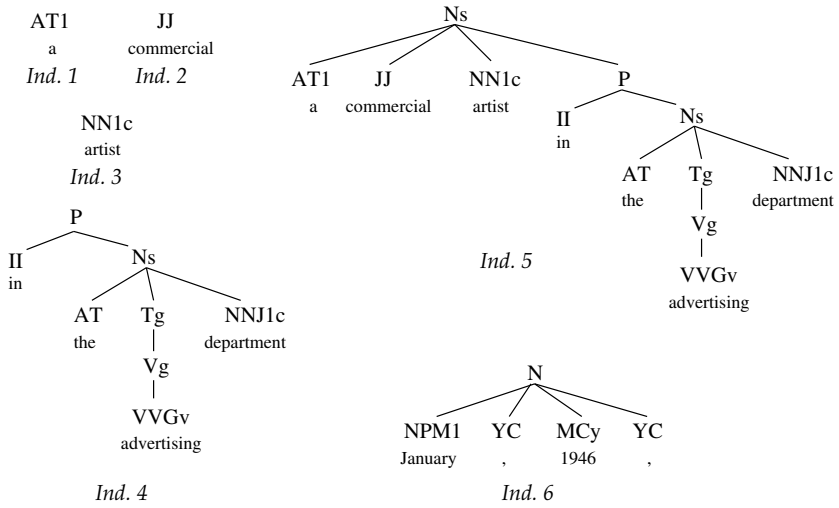

Fig. 3. Examples of individuals for the sentence *The new promotion manager has been employed by the company since January +, 1946 +, as a commercial artist in the advertising department +.*

individual is assigned a syntactic category: the left-hand side of the top-level rule in the parse. The probability of this rule is also registered. The first word of the sequence parsed by the tree, the number of words of that sequence and the number of nodes of the tree are also registered. Each tree is composed of a number of subtrees, each of them corresponding to the required syntactic category of the right-hand side of the rule. Figure 3 shows some individuals for the sentence *The new promotion manager has been employed by the company since January +, 1946 +, as a commercial artist in the advertising department +.*, used as a running example, which has been extracted from the Susanne corpus. We can see that there are individuals composed of a single word, such as *1*, while others, such as *5*, are a parse tree obtained by applying different grammar rules. For the former, the category is the chosen lexical category of the word (a word can belong to more than one lexical class); e.g., the category of *1* is *AT1*. For the latter, the category is the left hand-side of the top level rule; e.g., the category of *5* is *Ns*.

Chart Initialization: The Initial Population The first step in parsing a sentence with a bottom-up chart parser is to initialize the chart with word edges. For each lexical rule which expands to a given word, a corresponding edge is created. This edge stores a leaf tree and a location equal to the location of the word in the sentence.

Lexical rules can be obtained by finding the possible lexical tags of each word. For example, for the word *as*, which has the tags *CSA*, *IIa* and *RGa* in the

Susanne corpus, we will include in the grammar the rules $CSA \longrightarrow as$, $IIa \longrightarrow as$ and $RGa \longrightarrow as$. The lexical tags are obtained, along with their frequencies, from a dictionary.

The initialization of our system amounts to creating the initial population of the evolutionary algorithm which, like for the chart parser, is composed of individuals that are leave trees formed only by a lexical category of the word. The system generates a different individual for each lexical category of the word.

In order to improve the performance, the initial population also includes individuals obtained by applying a grammar rule provided that all the categories of the right-hand side of the rule are lexical. The individual *6* of Figure 3 is one such example.

2.2 Arc-Extension Algorithm: the Genetic Operators

The edge set represents the state of a chart parser during processing, and new edges can be inserted into the set at any time. Once entered, an edge cannot be modified or removed. New edges can correspond to complete components or to active arcs, pending of further extension to be completed.

In our system, active arcs only have a temporary existence and they do not appear in the population, which is only composed of complete components. The process of extending an arc is not partitioned in a number of steps, but is done as a whole in the crossover operation. New individuals are created by means of two genetic operators: *crossover* and *cut*. The crossover operator combines a parse with other parses present in the population to satisfy a grammar rule; cut creates a new parse by randomly selecting a subtree from an individual of the population. The rates of crossover and cut operations performed at each step are input parameters, to which the algorithm is very sensitive.

At each generation genetic operators produce new individuals which are added to the previous population that in this way is enlarged. The selection process is in charge of reducing the population size down to the size specified as an input parameter. Selection is performed with respect to the relative fitness of the individuals, but it also takes into account other factors to ensure the presence in the population of parses containing words that can be needed in later generations.

Reproduction The crossover operator produces a new individual by combining an individual selected from the population with an arbitrary number of other ones. Notice that the crossover in this case does not necessarily occurs in pairs. The operator repeatedly applies the arc-extension algorithm until completed components are obtained. The individuals to be crossed are randomly selected. This selection does not consider the fitness of the individuals because some grammar rules may require, to be completed, individuals of some particular syntactic category for which there are not higher fitness representatives.

Crossover begins by selecting an individual from the population. For example, let us assume that the individual *1* of Figure 3 is selected. Then the process continues as follows:

- Identify the syntactic category of the root of the tree to be combined. For *Ind1* it is *AT1*.
- Select among the grammar rules those whose right-hand side begins with this syntactic category. For the grammar used in this work, some examples of these rules for the *AT1* category are:

  Ns → AT1 JJ NN1c P
  Ns → AT1 JJ NN1n P
  Ns → AT1 JJ Tg NN1c P
  Ns → AT1 NN1c Po
  Ns → AT1 NN1c YC Nns YC MCn
  Ns → AT1 NN1n Po

  Let us assume that the first rule is the selected one.
- For each category of the right hand side of the rule after the first one, search in the population for an individual whose syntactic category matches the considered category, and whose sequence of words is the continuation of the words of the previous individual. In the example, we look for an individual of category *JJ*, other of category *NN1c* and other of category *P*. The sequence of words of the individual of category *JJ* must begin with the word *commercial*, the word which follows those of the individual *1*. Accordingly, the individual *2* of Figure 3 is a valid one (likewise, individuals *3* and *4* are also chosen for the crossover).
- Construct a new individual which has in its root the syntactic category of the rule (Ns) and is composed of the subtrees selected in the previous step, what produces the individual *5* of Figure 3.
- Add the new individual to the population.

With this scheme, the crossover of one individual may produce no descendant at all, or may produce more than one descendant. In this latter case all descendants are added to the population. The process of selection is in charge of reducing the population down to the specified size. Figure 4 shows a scheme of the operator. For each individual in the population the operator is applied according to the crossover rate. If a individual is selected to be crossed, the function *SearchGrammarRules* returns the set of grammar rules whose right-hand side begins with the category of the individual. Then, for each of these rules, the function *SearchIndividuals* searches the population for individuals to complete the right-side of the rule. If they are found, new individuals are created for each possible combination (*CreateTree*) and added to the population. If crossover is applied alone, the mean size of the individuals increases at each generation. Though this is advantageous because at the end we are interested in providing as solutions individuals which cover the whole sentence, it may also induce some problems. If the selection process removes small individuals which can only can be combined in later generations, the parses of these combinations will be never produced. This situation is prevented by applying some constraints in the selection process, as well as by means of the *cut* operator.

```
function Crossover(Population, per_crossover)
    for each individual in Population do{
        prob = random(100);
        if (prob < per_crossover) {
            Rules = SearchGrammarRules(individual);
            for each rule in Rules do{
                Trees = SearchIndividuals(Population, rule, individual);
                for each selection in Trees do{
                    new_individual = CreateTree(rule, individual, selection);
                    population.add(new_individual);
                }
            }
    }
end
```

Fig. 4. Scheme of the crossover operator

The Cut operator Because our chart representation does not ensures the presence of any edge previously produced, we include the *cut* operator, which allows extracting a part of a parse from an individual. The rate of application of the cut operator increases with the length of the individuals, while crossover is applied at a fixed rate. Thus, in the beginning of the evolution process, crossover is applied almost alone, and so the length of the individuals increases. Later on, when the length of the individuals is long enough, cut and crossover are applied together. Accordingly, the application of the cut operator depends on two parameters, *per_cut* and *threshold_cut*. *Per_cut* is the percentage of application of cut, while *threshold_cut* is the minimum number of words of the individual required to allow the application of *cut*. It is given as a percentage of the length of the sentence being parsed. Figure 5 shows a scheme of this operator. For each individual in the population, the conditions to apply cut are checked. If they are fulfilled, a subtree of the parse tree of the individual is randomly selected and added to the population.

```
function Cut(Population, per_cut, threshold_cut)
    for each individual in Population do{
        if number_words(individual > threshold_cut){
            prob = random(100);
            if (prob < per_cut) {
                new_individual = choose_random_subtree(individual);
                population.add(new_individual);
            }
    }
end
```

Fig. 5. Scheme of the cut operator

2.3 Selection of Components: Fitness Evaluation

A classic parser produces any valid parse of the substrings of the input. This is not guaranteed in an evolutionary algorithm, which tends to generate highly probable individuals, though individuals with low probability also have some opportunities to be produced and survive. Thus, we need a measure of the individual quality or fitness.

The fitness function is basically a measure of the probability of the parse. It is computed as the average probability of the grammar rules used to construct the parse:

$$fitness = \frac{\sum_{\forall s_i \in T} prob(s_i)}{nn(T)}$$

where T is the tree to evaluate, s_i each of its nodes and $nn(T)$ is the number of nodes. For the lexical category, the probability is the relative frequency of the chosen tag.

2.4 The Selection Process

Selection usually replaces some individuals of the population (preferably those with lower fitness) by others generated by the genetic operators. However, there are two issues that make selection a bit different in our case. First at all, our genetic operators include every new individual in the population, that in this way grows arbitrarily and therefore needs to be reduced to a suitable size. And secondly, if fitness were the only criterion to select the individuals to be eliminated, individuals that are the only ones parsing a particular word of the sentence could disappear, thus making impossible to generate a complete parse of the sentence in later generations. Accordingly, our selection process reduces the size of the population by erasing individuals according to their fitness but always ensuring that each of their words are present in at least other individual. If the population size *popu_size* is not enough to allow this, the parsing process is halted. However, this situation is not to be expected because any population size larger than the length of the sentence is enough to guarantee this condition. Figure 6 presents an scheme of the selection process. First at all, in order to improve diversity, duplicated individuals are erased from the population (*Eliminate_duplicate*). The function *ChooseToErase* returns the sequence with the tentative order in which the individuals must be erased. This order is randomly determined with probability inversely proportional to the fitness. Then a loop which erases individuals is repeated until the population is sufficiently reduced. The function *SequencePresent* checks that every of word in the sequence parsed by the individual *ToErase[i]* is present in at least another individual.

3 Experimental Results

The algorithm has been implemented using C++ language and run on a Pentium III processor. In order to evaluate its performance we have considered the

```
function Selection(Population, popu_size)
    Eliminate_duplicate(Population);
    ChooseToErase(Population, ToErase);
    i = 0;
    while Size(Population) > popu_size do{
        if SequencePresent(Population, ToErase[i]){
            Erase(Population, ToErase[i]);
            i++;
            }
    }
end
```

Fig. 6. Scheme of the selection process

parsing of the sentences extracted from the Susanne corpus [9], a database of annotated sentences from the Brown Corpus of written American English manually annotated with syntactic information. The Susanne analytic scheme has been developed on the basis of samples of both British and American English. The corpus comprises 64 files of annotated text and a lexicon. Each file has a line for each word of the original text. In this corpus, punctuation marks and the apostrophe-s suffix are treated as separate words and assigned separate lines. Each line has six fields, which contain at least one character. Figure 7 shows some lines of one Susanne file.

```
A01:0120.21     -       AT      the     the     [Ns-.
A01:0120.24     -       NN1c    number  number  .
A01:0120.27     -       IO      of      of      [Po.
A01:0120.30     -       NN2     voters  voter   .Po]Ns-]
```

Fig. 7. Sequence of lines extracted from the file A01 of the Susanne corpus. The fields of each line are *reference*, *status*, *wordtag*, *word*, *lemma* and *parse*.

The grammar used herein has been read off the parsed sentences of the Susanne corpus. In order to simplify the process, those sentences which make reference to elements outside them (*trace* sentences) have not been used to extract the grammar. Each grammar rule is assigned a probability computed as its relative frequency with respect other rules with the same left-hand side [1].

[1] That is, if we are considering the rule r of the form $A \rightarrow \cdots$, the probability of r is computed as:

$$P(r) = \frac{\#r}{\sum_{r'=A\rightarrow\cdots} \#r'}$$

where $\#r$ is the number of occurrences of r

3.1 Recall, Precision and Accuracy

Recall, precision and accuracy are the most commonly measures used for parsing evaluation. *Precision* is given by the number of brackets in the parse to evaluate which match those in the correct tree; *recall* measures how many of the brackets in the correct tree are in the parse, and *accuracy* is the percentage of brackets from the parse which do not cross over the brackets in the correct parse.

One of the several reasons why a parser can produce a wrong parse for a sentence is that the necessary rules are not present in the grammar due to lack of statistics in the corpus. It is a serious problem when using the Susanne corpus because of its large tag sets. However, if we are mainly interested in evaluating a parser, this problem can be circumvented by applying the parser to sentences from the training corpus. Thus we have tested the parser on a set of 17 sentences from the training corpus (the average length of the sentences is 30 words). In order to compare the EA with a classic parser, we have implemented a classic *best-first chart parsing* (BFCP) algorithm. Table 1 shows the precision, recall, accuracy and tagging results obtained for grammars of different size (best results achieved in ten runs). We can observe that the results of the EP improve those of a classic chart parser for the first grammar. Though these results get a bit worse when the size of the grammar is enlarged, they can be improved again by modifying the parameters of the evolutionary algorithm (those employed are suitable for the grammar of 225 rules). Anyway, the Susanne corpus produces too large grammars, inappropriate for the EP, so we expect to improve the results by using more appropriate corpus.

What is most relevant in the obtained results is that the EP is able to reach a 100% in any of the three measures, while the probabilistic chart parsing does not reach this value simply because the correct parse of some sentences is not the most probable one. In this way the heuristic component of the evolutionary algorithm shows its usefulness for parsing.

	225 r.		446 r.		795 r.	
	BFCP	EP	BFCP	EP	BFCP	EP
Precision	99.23	100	99.23	99.01	99.23	97.48
Recall	99.23	100	99.23	99.01	99.23	94.86
Accuracy	98.20	100	98.20	99.01	98.20	97.42
Tag. accuracy	100	100	100	100	100	99.61

Table 1. Results obtained for different sizes of the grammar with a best-first chart parser (BFCP) and with the evolutionary parser (EP).

4 Conclusions

This paper presents an implementation of a probabilistic chart parser by means of an evolutionary algorithm. It works with a population of partial parses for a given input sentence and a given grammar. Evolutionary algorithms allow a statistical treatment of the parsing process, providing at the same time the typical generality of the evolutionary methods, which allows to use the same algorithm scheme to tackle different problems.

The grammar and sentences used to evaluate the system have been extracted from the Susanne corpus. Measures of precision, recall and accuracy have been provided, obtaining results which improve those of a classic chart parser, thus indicating that the evolutionary algorithms are a robust approach for natural language parsing. Moreover, the heuristic component of these algorithms seems to harmonize with the non deterministic nature of the natural language tasks.

Another conclusion of this work is the inappropriateness of the Susanne corpus for statistical processing. The sets of lexical and syntactic tags used in this corpus are too large to get significant statistics. We thus plan to test the system on other corpora.

References

1. J. Allen. *Natural Language Understanding.* The Benjamin/Cumming Publishing Company, Inc., 1994.
2. L. Araujo. A parallel evolutionary algorithm for stochastic natural language parsing. In *Proc. of the Int. Conf. Parallel Problem Solving from Nature (PPSNVII)*, 2002.
3. L. Araujo. Part-of-speech tagging with evolutionary algorithms. In *Proc. of the Int. Conf. on Intelligent Text Processing and Computational Linguistics (CICLing-2002), Lecture Notes in Computer Science 2276*, pages 230–239. Springer-Verlag, 2002.
4. E. Charniak. *Statistical Language Learning.* MIT press, 1993.
5. B. Keller and R. Lutz. Evolving stochastic context-free grammars from examples using a minimum description length principle. In *In Worksop on Automata Induction, Grammatical Inference and Language Acquisition. ICML097.* ICML, 1997.
6. Anne Kool. Literature survey, 2000.
7. Robert M. Losee. Learning syntactic rules and tags with genetic algorithms for information retrieval and filtering: an empirical basis for grammatical rules. *Information Processing & Management*, 32(2):185–197, 1996.
8. T. Dunning M. Davis. Query translation using evolutionary programming for multilingual information retrieval II. In *Proc. of the Fifth Annual Conf. on Evolutionary Programming.* Evolutionary Programming Society, 1996.
9. G. Sampson. *English for the Computer.* Clarendon Press, Oxford, 1995.
10. T.C. Smith and I.H. Witten. A genetic algorithm for the induction of natural language grammars. In *Proc. IJCAI-95 Workshop on New Approaches to Learning Natural Language*, pages 17–24, Montreal, Canada, 1995.
11. P. Wyard. Context free grammar induction using genetic algorithms. In *Proc. of the 4th Int. Conf. on Genetic Algorithms*, pages 514–518, 1991.