A Genetic Programming Experiment in Natural Language Grammar Engineering

Marcin Junczys-Dowmunt

Faculty of Mathematics and Computer Science
Adam Mickiewicz University
ul. Umultowska 87, 61-614 Poznań, Poland
junczys@amu.edu.pl

Abstract. This paper describes an experiment in grammar engineering for a shallow syntactic parser using Genetic Programming and a treebank. The goal of the experiment is to improve the *Parseval* score of a previously manually created seed grammar. We illustrate the adaptation of the Genetic Programming paradigm to the problem of grammar engineering. The used genetic operators are described. The performance of the evolved grammar after 1,000 generations on an unseen test set is improved by 2.7 points F-score (3.7 points on the training set). Despite the large number of generations no overfitting effect is observed.

Keywords: Shallow parsing, genetic programming, natural language grammar engineering, treebank.

1 Introduction

This paper describes an experiment in grammar engineering for a shallow syntactic parser using Genetic Programming and a treebank. The goal of the experiment is to improve the *Parseval* [1] score of a previously manually created seed grammar. The shallow parser cannot be easily trained on a treebank using classical grammar extraction methods like the creation of a probabilistic context-free grammar. Neither does the parser support weights nor is the grammar formalism context-free. Parsing rules are applied in the same order as they appear in the rule set, no search is carried out during parsing. This requires a human grammar engineer to tune the grammar very carefully.

The Genetic Programming [2,3] approach has been chosen because we regard it to be similar to the grammar engineering process as it is performed by a human grammar engineer, based on a trial-and-error search roughly guided by a treebank. We illustrate the adaptation of the Genetic Programming paradigm to the problem of grammar engineering by treating grammars as programs.

2 Related Work

Various flavors of genetic algorithms have been widely used for the induction of context free grammars (e.g. [4,5]), but only few (e.g. [6,7]) focus on natural language grammars.

P. Sojka et al. (Eds.): TSD 2012, LNCS 7499, pp. 336–344, 2012.

These works use Genetic Programming or other genetic algorithms for the unsupervised inference of context free grammars from unannotated corpora with rather unsatisfactory results.

For the training on annotated data like treebanks, direct grammar extraction approaches are dominant. Probabilistic context free grammars can be read out from the given tree structures, probabilities are based on frequency of the extracted structure occurring in the treebank, see for instance [8] for experiments on the Penn Treebank or [9] for results for the German TüBa D/Z treebank. We know of no work that uses treebanks to automatically improve manually created grammars.

3 The Seed Grammar

3.1 The Shallow Parser

The shallow parser used in our experiments is a component of the *PSI-toolkit* [10]. It is based on the *Spejd* [11] shallow parser and employs a similar rule formalism. It has been used as a parser for French, Spanish, and Italian in the syntax-based statistical machine translation application *Bonsai* [12] and in other applications.

The parsing process relies on a set of string matching rules constructed as regular expressions over single characters, words, part-of-speech tags, lemmas, grammatical categories etc. Apart from the matching portion of a rule, matching patterns for left and right contexts of the main match can be defined. The parse tree construction process is linear, matching rules are applied deterministically. The first match is chosen and a spanning edge is added to the result. No actual search is performed during parsing. The parsing process for a sentence is finished if during an iteration no rule can be applied. The parser cannot make use of weights or other disambiguation methods.

3.2 Manual Grammar Engineering

For this kind of parser the use of a treebank is limited, for instance, it is not possible to use a PCFG. Acceptable parsing quality is achieved by the careful manual construction of the parsing grammar. The order of the rules in a grammar determines the order of application to a sentence.

However, a treebank in combination with an evaluation metric like *Parseval* can be used to guide the grammar engineer through a trial-and-error process. A human grammar designer seeks to improve the grammar in such a way that the score is improved. If the amount of work that needs to be invested into the grammar is not reflected in further improvement, the construction can be regarded as finished.

Relying on *The French Treebank* [13], a basic French grammar has been created for the `PSI-toolkit` shallow parser which is also used for French-Polish translation in the mentioned *Bonsai* MT system. The manual work on that grammar was stopped at an F-score level of 76.55% on the development set (76.20% on the test set). This grammar serves as the starting point for the experiment.

4 Genetic Programming

Genetic programming (GP) [2,3] is a genetic algorithm flavor that evolves computer programs. A population of programs is optimized according to a fitness function that measures the performance of an individual on a training set. Programs that perform better than others are allowed to produce off-springs which are added to the next generation of computer programs. Repeating this process for a large number of iterations is expected to result in a population of programs that perform reasonably well at the task the fitness function was based on.

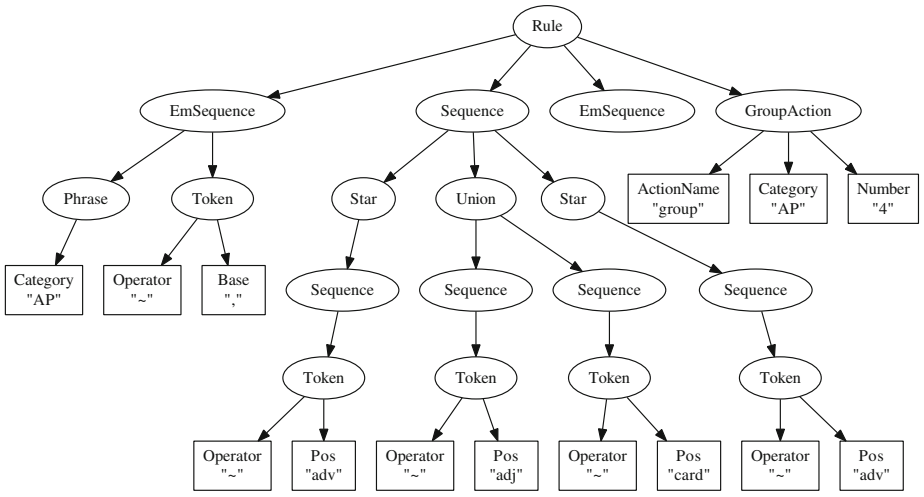


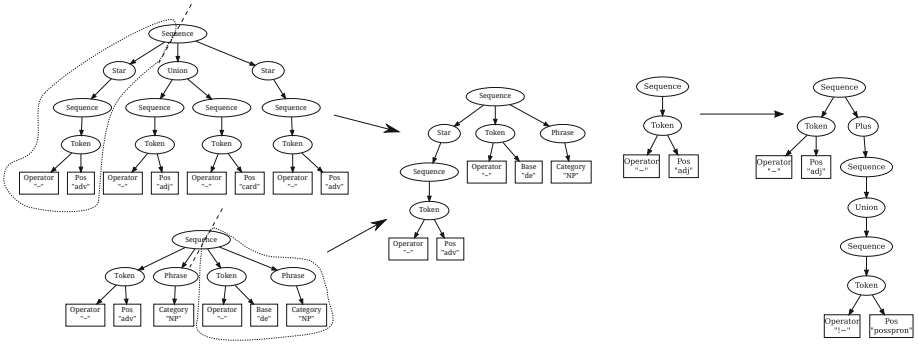
Fig. 1. Chromosome tree representation of rule `Example 1`

4.1 Chromosome Representation

Genetic programming evolves computer programs, which are most commonly represented as tree structures. Child nodes contain arguments to the functions represented by their respective parent nodes. Running a program is equivalent to a recursive evaluation of the tree. In the terminology of genetic algorithms such a tree representation is called a *chromosome*.

If the shallow parser is treated as a programming language interpreter, a grammar is nothing else but an interpretable program. This assumption makes it straightforward to apply GP directly to grammar engineering. The grammar only needs to be represented as a tree structure that can be modified by genetic operators. The original grammar is a text file consisting of rules similar to the following:

```
Rule "Example 1"
Left:  [type=AP] [base~","];
Match: ([pos~"adv"])* ([pos~"adj"]|[pos~"card"]) ([pos~"adv"])*;
Eval:  group(AP,4);
```



(a) Crossover (b) Mutation

Fig. 2. Variable-arity genetic operators

This rule creates an adjective phrase from an adjective or a cardinal number that can be preceded or followed by any number of adverbs (including none). The rule is only allowed to create the adjective phrase if its immediate left context is another adjective phrase followed by a comma. Fig. 1 gives the tree representation of this rule. A full grammar consists of many such trees, the roots of which are children of a single `RuleSet` node. We differentiate between fixed-arity nodes (fixed number of children) and variable-arity nodes (variable number of children) for which specialized genetic operators are implemented.

4.2 Genetic Operators

The main genetic operators used in GP — and other genetic algorithms — are crossover and mutation. In our case two variants exist for each operator depending on the arity of the nodes involved. The genetic operators are type-aware and guaranteed to generate a correct grammar.

Crossover. Crossover creates a new individual by combining genetic material of two individuals from the previous generation that have been selected for reproduction. A node of the first individual is randomly selected and based on its type a node of the same type is randomly selected from the second individual.

Fixed-arity Crossover. Fixed-arity crossover simply replaces all child nodes (and attached subtrees) of the first chosen node with the children of the second node.

Variable-arity Crossover. Variable-arity crossover is a variant of "cut-and-splice" crossover. The number of children for both chosen nodes may vary, therefore two random crossover points are chosen for each sequence of children. The new individual contains all children of the first node left of the first crossover point and all children of the second node right of the second crossover point. The example below is a possible

result of the crossover of rule `Example 1` with another rule. Fig. 2a illustrates the operation for fragments of both rules.

```
Rule "Example 1 variable-arity crossover"
Left:  [type=AP] [base~","];
Match: ([pos~"adv"])* [base~"de"] [type=NP];
Eval:  group(AP,4);
```

Mutation. Mutation forms a new individual from a single selected individual of the previous generation.

Fixed-arity Mutation. Fixed-arity mutation replaces a randomly chosen child of a fixed-arity node with a randomly generated tree. The root of this tree has to be of the same type as the replaced node.

Variable-arity Mutation. Variable-arity mutation simply adds a randomly generated branch to the list of children of the mutated variable-arity node. The rule below is the result of the mutation of a node of rule `Example 1` illustrated in Fig. 2b.

```
Rule "Example 1 - variable-arity mutation"
Left:  [type=AP] [base~","];
Match: ([pos~"adv"])* ([pos~"adj"]
       (([pos!~"posspron"]))+|[pos~"card"]) ([pos~"adv"])*;
Eval:  group(AP,4);
```

4.3 Fitness Function

The measure how well an individual is doing compared to other individuals in its generation is called *fitness*. The labeled version of the *Parseval* [1] metric is a natural candidate for a fitness function in the case of grammar engineering. Therefore, the *Parseval* F-score is chosen as the first component of our fitness function. Obviously, the greater the F-score the fitter is the evaluated individual.

Bloat — a rapid increase in the size of evolved trees — is a common phenomenon for genetic programming. Many strategies have been proposed to counteract the bloat effect [3, p. 78]. We choose to use a multi-objective fitness function: apart from optimizing F-score, tree complexity measured as the number of nodes is also optimized by applying Ockham's razor: if two individuals have the same F-score, the smaller tree is considered fitter than its larger competitor.

4.4 Algorithm

Algorithm 1 contains the pseudo-code of the main evolution procedure. The parameter `seedGramar` is the seed grammar the algorithm is supposed to improve. The first population consists of `populationSize` copies of that grammar. We set `populationSize` to 500 individuals per generation and limit the evolutionary process to 1,000 generations (`maxGenerations`). The parameters `eliteFraction` and `selectFraction` are

Algorithm 1. Main evolution procedure

```
Require: seedGrammar, trainingSet, populationSize, eliteFraction, selectFraction,
  crossProb, mutateProb, maxGenerations, maxComplexity, order

  seedScore ← EVAL(seedGrammar, trainingSet)
  seedNodes ← COMPLEXITY(seedGrammar)
  population ← {(seedGrammar, seedScore, seedNodes) | 1, 2, ..., populationSize}
  for i ← 1 to maxGenerations do
    nextPopulation ← BESTN(population, order, populationSize × eliteFraction)
    selection ← SELECT(population, order, populationSize × selectFraction)
    for k ← 1 to popsize do
      offspring ← GENOP(selection, crossProb, mutateProb)
      offsprComplexity ← COMPLEXITY(offspring)
      if offsprComplexity > maxComplexity then
        score ← 0
      else
        score ← EVAL(offspring, trainingSet)
      end if
      nextPopulation ← nextPopulation ∪ {(offspring, score, offsprComplexity)}
    end for
    population ← nextPopulation
  end for
  return BESTN(population, order, 1)
```

both set to 0.2. This means that the first 100 best individuals of a population are copied unaltered to the next population and are the only ones allowed to reproduce. The total number of nodes in a tree is limited by `maxComplexity` which is set to 25,000. The two parameters `crossProb` and `mutateProb` determine the probabilities of the respective genetic operators. For our experiment, a mutation probability of 0.2 and a crossover probability of 0.8 were chosen. The sort order in the population is determined by the parameter `order` which implements Ockham's razor for the fitness function as described in the previous section.

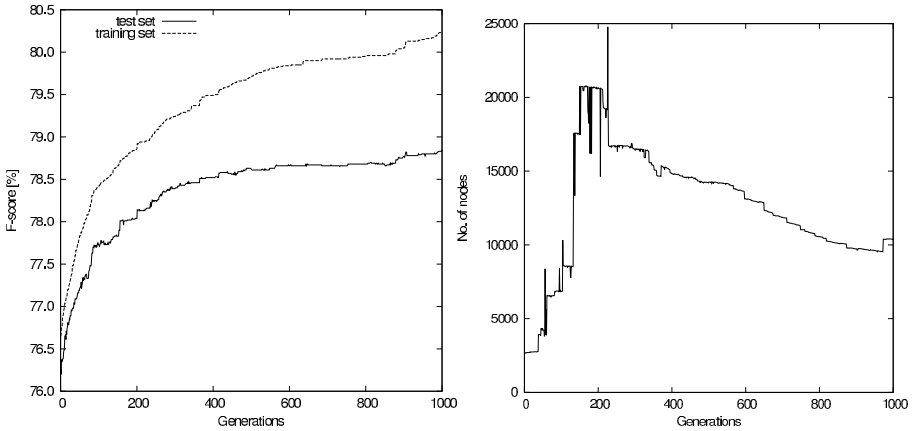
5 Evaluation

5.1 Training and Test Data

The French Treebank [13] comprises ca. 21,100 syntactically annotated sentences. Originally, the first 15,000 sentences were set apart as a training set, but for this experiment only the first 2,500 sentences are used in order to make calculations feasible. For 1,000 iterations of the algorithm with 400 new individuals per iteration the training set needs to be parsed 400,000 times. The performance of the best individual in each generation is evaluated on the last 6,000 sentences of the treebank which have not been seen during training.

Table 1. Performance for chosen generations

		Training data			Test data		
Generation	Complexity	Precision	Recall	F-score	Precision	Recall	F-score
0	2,672	77.29	75.83	76.55	76.90	75.52	76.20
50	4,235	78.80	76.87	77.83	78.13	76.31	77.21
100	6,834	79.46	77.40	78.42	78.70	76.78	77.73
500	14,256	81.08	78.38	79.71	80.02	77.26	78.61
1,000	10,395	81.49	79.03	80.24	79.99	77.74	78.85



(a) F-Score for training and test set (b) Size of the best-ranked rule set

Fig. 3. Progression of F-Score and Complexity

5.2 Results

Tab. 1 and Fig. 3 summarize the results for the best individual of each generation. The 0^{th} generation represents the seed grammar. As shown on Fig. 3a, the increase in quality progresses slower on the test set than on the training data. However, during the first 1,000 iterations no overfitting seems to occur. From the 400^{th} generation on, little improvement on the test data can be observed, but the curve follows the shape of the curve for the training data. The performance of the evolved grammar after 1,000 generations on the unseen test set is improved by 2.7 points F-score (3.7 points on the training set).

The changes in size of the best rule set in each generation are presented in Fig. 3b. At first, increasing the size of the rule set seems to be a good strategy to increase parse quality. A cut-off size of 25,000 nodes is applied and one might expect the grammars to reach this maximum size and to stay there for a longer time ([14]). Interestingly, apart form a single peak in the 226^{th} generation this does not happen. Starting from this generation, the second optimization objective, rule set complexity, plays a greater role.

Size decreases steadily and compared to the rather dynamic first phase of the process no significant jumps in size occur.

6 Conclusions and Future Work

We demonstrated that a Genetic Programming algorithm can improve our complex, manually created natural language grammar with the help of a treebank by around 2.7 points *Parseval* F-score on an unseen test set. The grammar is non-probabilistic and not context-free. It can be assumed that the presented method could be used to any type of grammar that can be represented as a tree structure. The process is however quite resource-intensive.

Future research needs to focus on larger training data sets and better parameter settings for the presented algorithm. Another important direction will include the full extraction of a grammar from treebanks without a seed grammar. In that case a more popular treebank shall be used to make results comparable to traditional approaches to treebank grammars.

Acknowledgements. This paper is based on research funded by the Polish Ministry of Science and Higher Education (Grant No. N N516 480540).

References

1. Abney, S., Flickenger, S., Gdaniec, C., Grishman, C., Harrison, P., Hindle, D., Ingria, R., Jelinek, F., Klavans, J., Liberman, M., Marcus, M., Roukos, S., Santorini, B., Strzalkowski, T.: A Procedure for Quantitatively Comparing the Syntactic Coverage of English Grammars. In: Proceedings of a Workshop on Speech and Natural Language, San Francisco, pp. 306–311 (1991)
2. Koza, J.R.: The Genetic Programming Paradigm. In: Dynamic, Genetic, and Chaotic Programming, New York, pp. 203–321 (1992)
3. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming (2008), `http://www.gp-field-guide.org.uk`
4. Dunay, B.D., Petry, F.E., Buckles, W.P.: Regular Language Induction with Genetic Programming. In: Proc. of the 1994 IEEE World Congress on Computational Intelligence, Orlando, pp. 396–400. IEEE Press (1994)
5. Keller, B., Lutz, R.: Learning Stochastic Context-Free Grammars from Corpora Using a Genetic Algorithm. University of Sussex (1997)
6. Smith, T.C., Witten, I.H.: A Genetic Algorithm for the Induction of Natural Language Grammars. In: Proc IJCAI 1995 Workshop on New Approaches to Learning for Natural Language Processing, pp. 17–24 (1995)
7. Korkmaz, E.E., Ucoluk, G.: Genetic Programming for Grammar Induction. In: 2001 Genetic and Evolutionary Computation Conference, San Francisco (2001)
8. Klein, D., Manning, C.D.: Accurate Unlexicalized Parsing. In: Proc. of the 41st Annual Meeting of the Association for Computational Linguistics, pp. 423–430 (2003)
9. Kübler, S., Hinrichs, E.W., Maier, W.: Is it really that difficult to parse German. In: Proc. of the Conference on Empirical Methods in Natural Language Processing, pp. 111–119 (2006)
10. Graliński, F., Jassem, K., Junczys-Dowmunt, M.: PSI-toolkit: A Natural Language Processing Pipeline. In: To appear in: Computational Linguistics — Applications. SCI. Springer

11. Przepiórkowski, A., Buczyński, A.: ♠: Shallow parsing and disambiguation engine. In: Proceedings of the 3rd Language & Technology Conference, Poznań (2007)
12. Junczys-Dowmunt, M.: It's all about the Trees — Towards a Hybrid Syntax-Based MT System. In: Proceedings of IMCSIT, pp. 219–226 (2009)
13. Abeillé, A., Clément, L., Toussenel, F.: Building a Treebank for French. In: Treebanks: Building and Using Parsed Corpora, pp. 165–188. Springer (2003)
14. Crane, E.F., McPhee, N.F.: The Effects of Size and Depth limits on Tree Based Genetic Programming. In: Genetic Programming Theory and Practice III, pp. 223–240. Springer (2005)