Maximum Likelihood Analysis of Heap Sort

Ulrich Laube, Markus E. Nebel

October 10, 2008

Abstract

We present a new approach for an average-cases analysis of algorithms that supports a non-uniform distribution of the inputs and is based on the maximum likelihood training of stochastic grammars. The approach is exemplified by an analysis of the average running time of heap sort. All but one step of our analysis can be automated on top of a computer-algebra system. Thus our new approach eases the effort required for an average case analysis exceptionally allowing for the consideration of realistic input distributions with unknown distribution functions at the same time.

1 Introduction

Finding a precise characterization of the behavior of an algorithm or a data structure, given a few assumptions about the probabilistic distribution of the inputs, is the aim of the analysis of algorithms. Typically the execution time or the amount of memory used shall by studied. As best- and worst-case often occur with a rather low probability the average-case is more informative, if the underlying probabilistic model is realistic. But many times a realistic model is not known or one has to keep the mathematics of the analysis manageable and thus falls back to a uniform distribution instead of a more realistic one. And even if more complicated distributions like i.e. a Bernoulli model can be handled mathematically we often have to admit that it still does not fit reality.

Our maximum likelihood approach presented in this paper avoids the two problems mentioned. We are able to evaluate the expected performance while using a realistic probabilistic model and at the same time keep the mathematics practicable. This is possible by supplementing the rules of a Chomsky-grammar with probabilities and training them according to the maximum likelihood principle on real and hopefully typical inputs of different sizes. The grammar itself is set up in such a way, that the words of it's language represent the flow of control through a program.

According to the ideas of Chomsky and Schützenberger [3] such a stochastic grammar can be translated into a probability generating function for the generated language. Then the expected value and higher moments of several parameters of the language are readily available by standard generating function methods. This makes it easy to analyze the expected running time of an algorithm as the underlying grammar will be right-linear implying rational generating functions only.

In this case all the methods which are necessary for this type of analysis can be performed automatically on a computer using computer-algebra, an automatic average-case analysis for inputs that have a non-uniform distribution is possible. We have nearly finished a prototype of such a system as a proof of concept. A software system that allows an automatic average-case analysis in the uniform case was presented in [5].

After collecting the results and definitions needed in section 2 we will demonstrate our approach by an analysis of heap sort in section 3.

2 Basic Definitions and Known Results

2.1 Stochastic Context-Free Grammars

We will use a generalization of context-free grammars by assigning probabilities to the rules. This leads to *stochastic context-free grammars* (SCFGs) as found in [8] which are commonly used in the area of natural language processing, see for example [12] and in connection with RNA secondary structure prediction in bioinformatics, see for example [4].

Definition 1 (stochastic context-free grammar)**.** *A* stochastic context-free grammar *is a five-tuple $G = (N, T, R, P, S)$ where $N = \{V_1 = S, V_2, \ldots, V_k\}$ is a set of variables or non-terminal symbols, $T = \{a_1, \ldots, a_m\}$ is a set of terminal symbols disjoint from N, $R = \{r_1, \ldots, r_n\}$ is a subset of $N \times (N \cup T)^{\star 1}$, its elements $r_j = (V_i, \omega_j)$ are called rules or productions. Additionally we denote by $R_i = \{r_j \mid r_j = (V_i, \omega_j) \in R\}$ the set of all rules with the same left-hand side V_i. P is a mapping from R into $]0, 1]$ that assigns each rule r_j its probability p_j. We write $V_i \to p_j : \omega_j$ for a rule $r_j = (V_i, \omega_j) \in R$ with $P(r_j) = p_j$. We require that*

$$\sum_{\{j | r_j \in R_i\}} p_j = 1 \quad i = 1, 2, \ldots k$$

holds, that is, we have probability distributions on the rules with the same left-hand side. The symbol $S = V_1 \in N$ is a distinguished element of N called the axiom or start symbol. All sets in this definition are finite.

The language $\mathcal{L}(G)$ generated by a SCFG G is the set of all terminal strings or words which can be generated by successively substituting all non-terminals according to the productions starting with the axiom S. Such a derivation is called *left-most* if always the left-most non-terminal is replaced next. The probability for such a derivation is given by the product of the probabilities of all productions used. The probability of a word generated by the grammar G is the sum of the probabilities of all its different left-most derivations. Every left-most derivation corresponds uniquely to a parse tree of a word. Clearly unambiguous languages are important here.

The probabilities on the rules thus induce probabilities on the words and parse trees but it is a priori unknown whether a probability distribution on the entire language is

[1] When X is a set of symbols, $X^\star$ denotes the set of all finite strings of symbols of X completed by the empty string ϵ.

induced by a SCFG or not. A SCFG is called *consistent* (sometimes *proper*) if it provides a probability distribution for the generated language, i.e.

$$\sum_{w \in \mathcal{L}(G)} p(w) = 1.$$

Example 1. *The ambiguous SCFG* $S \to \frac{2}{3} : SS, S \to \frac{1}{3} : a$ *is not consistent. As the first rule is more likely than the second, each of the* S*'s is likely to be replaced with two more* S*'s, thus the variable* S *multiplies without bound and the derivation* never terminates*. No word is produced in this case and the sum above is less than one due to the missing words. We will find a formal way to prove or disprove consistency later.*

2.2 Training and Consistency of SCFGs

What we call the training of a grammar employs the maximum likelihood principle. On a fixed sample from a larger population the maximum likelihood principle tunes the free parameters of the underlying probability model in such a way, that the sample has maximum likelihood, that is, other values for the parameters make the observation of the sample less likely. In our setup the free parameters are the probabilities of the grammar rules. Training the grammar then sets those probabilities in such a way that grammar generates words that closely match the sample set of words provided for the training with maximum likelihood.

The conditions for consistency of such a trained grammar has been investigated by a number of scientists. A conjecture by Wetherell [14] proven in [2] states that assigning relative frequencies found by counting the rules in the parse trees of a finite sample of words from the language results in a consistent SCFG. A simpler proof was found by Chi and Geman in [1]. The results cited above were obtained with more or less explicitly stated assumptions on the grammar, sometimes chain and epsilon rules are not allowed or the grammar must be unambiguous. In [13] these restriction are lifted. The authors prove that the relative frequency, the expectation maximization and a new cross-entropy minimization approach each yield a consistent grammar without restrictions on the grammar.

We will use the fact that estimating the probabilities of a stochastic context-free grammar by their relative frequencies yields a maximum likelihood estimate. Counting the relative frequencies is especially efficient for unambiguous grammars used here, as there is only one left-most derivation (parse tree) to consider.

2.3 Formal Power Series and SCFGs

Following [3] we can make use of the connection between formal languages and formal power series and translate an *unambiguous* context-free grammar into a corresponding structure generating function that counts the number of words of length n, see [6, 10].

Definition 2 (generating functions)**.** *The* ordinary generating function *of a sequence* $(a_n)_{n \geq 0}$ *is the formal power series*

$$A(z) := \sum_{i \geq 0} a_i \cdot z^i.$$

The probability generating function *of a discrete random variable X is*

$$P(z) := \sum_k \Pr(X = k) \cdot z^k.$$

In case of an unambiguous SCFGs the grammar can be translated into a probability generating function $P(z)$ where the coefficient $\Pr(X = k)$ is the probability that a word of length k is generated. By evaluating $P(1) = \sum_k \Pr(X = k)$ we can check whether the SCFG is consistent.

Example 2. *The unambiguous CFG* $G = \big(\{S\}, \{(,)\}, \{S \to (S)S, S \to \epsilon\}, S\big)$ *translates into the equation* $S(z) = z^2S(z)^2 + 1$*, which has the solution* $S(z) = \frac{1-\sqrt{1-4z^2}}{2z^2} = 1 + z^2 + 2z^4 + 5z^6 + 14z^8 + 42z^{10} + \mathcal{O}(z^{12})$*. It can easily be verified that the coefficients at* z^n *count the number of words of length* n *generated by* G*. The unambiguous SCFG* $G' = \big(\{S\}, \{(,)\}, \{S \to \frac{1}{3} : (S)S, S \to \frac{2}{3} : \epsilon\}, S\big)$ *translates into the equation* $S'(z) = \frac{1}{3}z^2S'(z)^2 + \frac{2}{3}$*, which has the solution* $S'(z) = \frac{3-\sqrt{9-8z^2}}{2z^2} = \frac{2}{3} + \frac{4}{27}z^2 + \frac{16}{243}z^4 + \frac{80}{2187}z^6 + \mathcal{O}(z^8)$*. Here the coefficient of* z^n *is the probability that* G' *produces a word of length* n *and verifying that* $S'(1) = 1$ *holds, confirms that* G' *is consistent.*

As seen in Example 2 the interesting information are the coefficients "hidden" in the generating function. Extracting information about the coefficients is possible by singularity analysis. However as the generating functions that emerge in the rest of the paper are rational functions, the `SeriesCoefficient` command of Mathematica 6 is able to provide exact expression for the coeffcients.

3 Maximum Likelihood Analysis of Heap Sort

The maximum likelihood analysis of heap sort performed in this section serves as a showcase, as other algorithms can be analysed too and a variation of this method works for data structures as well, see [11].

As the running time of an algorithm depends on the actual implementation and the machine model used, we opt for Knuth's MIX computer and the implementation from [9] as they are well documented and as results are available for later comparison. However our method is not connected to any specific machine model and works as general as any other technique for performing an average-case analysis.

First we have to describe how we derive the SCFG from the algorithm. The words of the language generated by the grammar will describe the possible flows of control. The algorithms in Knuth's books are written in MIXAL the MIX assembly language and their instructions are easily translated into the rules of a SCFG by the following three rules:

1. An unconditional JMP from line i to line j becomes: $L_i \to 1 : l_iL_j$.
2. A conditional JMP instruction in line i that may jump to line j becomes: $L_i \to p_i : l_iL_j \mid 1 - p_i : l_iL_{i+1}$ because the jump may be taken or not.
3. All other instructions yield: $L_i \to 1 : l_iL_{i+1}$.

When m is the number of the last line of the program we add one final rule $L_{m+1} \to \epsilon$ thereby allowing the grammar to produce terminal strings (which corresponds to termination of the program).

The words described by such a grammar are just sequences of the line numbers l_i of the instructions executed. As the grammars are right-linear we obtain regular languages. This is noteworthy as regular languages always have rational structure generating functions, see [10]. We process the SCFGs further to reduce their size, e.g. removing chain rules, but strictly speaking this is not necessary.

In the next step a homomorphism h is used to substitute the l_i's by $h(l_i) = y^{\mathrm{cost}(l_i)}$, as we do not care about the actual line numbers but about the cost (denoted by $\mathrm{cost}(l_i)$) it implies. In our setup the function cost just looks up the cost of the MIXAL instructions, as shown in the table on the inside of the back cover of Knuth's books. This substitution is however quite flexible, it allows us to introduce the parameter (here the running time) which we are interested in, into the grammar and thus into the generating function which we want to build.

As the coefficient at z^n in the generating function should "count" the parameter we are interested in, subject to the size of the input n, we have to make sure that the correct number of z's is inserted into the grammar to be able to keep track of the size of the input. In case of heap sort we "mark" the instruction(s) that moves the keys to their final positions with a z each. The code that performs this task is found at the lines 27 and 30 in Program H on page 146f in [9]. Thus when the sorting is finished we have n z's and thus the cost is correctly contributed to the coefficient at z^n. Note that the *placement* of the variables z, taking care of one or more elements of the input, is the only part of our analysis which has no obvious algorithmic automation, as it requires understanding of the algorithm under examination.

The rules of the grammar are now interpreted as equations. The equations are of course linear and after eliminating the variables we can find the solution, i.e. the generating function, which is a rational function in z and y as mentioned before. Turning Program H from page 146f in [9] into a grammar yields:

$$
\begin{aligned}
L_1 &\to 1 : \texttt{idpt}L_{21} & L_1 &= y^{12}L_{21} \\
L_{11} &\to p_{11} : \texttt{jolr}L_{16} \mid (1-p_{11}) : \texttt{jr}L_{16} & L_{11} &= p_{11}y^7L_{16} + (1-p_{11})y^3L_{16} \\
L_{16} &\to p_{16} : \texttt{jst}L_{21} \mid (1-p_{16}) : \texttt{jw}L_{24} & L_{16} &= p_{16}y^6L_{21} + (1-p_{16})y^3L_{24} \\
L_{21} &\to p_{21} : \texttt{j}L_{22} \mid (1-p_{21}) : \texttt{jcs}L_{11} & L_{21} &= p_{21}yL_{22} + (1-p_{21})y^5L_{11} \\
L_{22} &\to p_{22} : \texttt{jw}L_{24} \mid (1-p_{22}) : \texttt{jlr}L_{16} & L_{22} &= p_{22}y^3L_{24} + (1-p_{22})y^5L_{16} \\
L_{24} &\to p_{24} : \texttt{jg}zL_{29} \mid (1-p_{24}) : \texttt{jdpt}L_{21} & L_{24} &= p_{24}y^8zL_{29} + (1-p_{24})y^{11}L_{21} \\
L_{29} &\to p_{29} : \texttt{je}zL_{29} \mid (1-p_{29}) : \texttt{jpt}L_{21} & L_{29} &= p_{29}y^3z + (1-p_{29})y^8L_{21}
\end{aligned}
$$

Removing the chain rules gives a pair of rules for every conditional jump shown on the left with blocks of instructions that belong together, abbreviated with letters typeset with an upright font. The homomorphism h now replaces the non-italic letters in the rules in such a way that we track the running time in the exponent of symbol y in the

equations on the right. Back substitution yields the solution

$$H(z,y) = \frac{p_{24}p_{29}y^{27}z^2 \cdot h(y)}{1 - p_{16}p_{21}\bar{p}_{22}y^{12} - p_{16}y^{14} \cdot g(y) - \big(p_{24}\bar{p}_{29}zy^5 + \bar{p}_{24}\big)y^{15} \cdot h(y)} \tag{1}$$

with

$$g(y) = \bar{p}_{21}\bar{p}_{11} + p_{21}p_{11}y^4, \qquad \bar{p}_i = 1 - p_i,$$
$$h(y) = \bar{p}_{16}y^7 \cdot g(y) + p_{21}\bar{p}_{16}\bar{p}_{22}y^5 + p_{21}p_{22}.$$

As a consistent SCFG induces a probability distribution on the entire language we do not automatically have a probability distribution if we look on a subset of the language. To get a distribution on the subset we have to normalize the probabilities involved. This is achieved by dividing through the probability of the subset, in our case the probability that the flow of control string was recorded for an input of size n. Accordingly we have to determine the quotient of the weighted cost of all inputs of size n and the probability for such an input[2]:

$$\begin{aligned} H(n) &:= \frac{<z^n, \frac{\partial}{\partial y}H(z,y) \mid_{y=1}>}{[z^n]H(z,1)} \\ &= 2 + 5n + \frac{(n-1)\big(22 + 4p_{11}(1-p_{21}) - p_{21}\big(2 + 5p_{22}\big) - 8p_{16}(1 - p_{21}p_{22})\big)}{(1 - p_{16}\big(1 - p_{21}p_{22}\big)\big)p_{24}}. \end{aligned} \tag{2}$$

This solution still contains the probabilities p_i, and we use the maximum likelihood principle to train them on five generate sets of integer sequences. Each set is based on a different probability model (random permutation (rp), Gauss distributed (gd), nearly ordered ascending (noa) and descending (nod), many duplicates (md)) and consists of sequences with lengths from 10 to 860 in steps of 10. For each size 100 sequences are randomly generated and sorted with heap sort running on a MIX virtual machine from the GNU MIX Development Kit. For each input we record the flow of control as a string of line numbers and parse those words subsequently. The number of occurrences of each rule in the parses is counted for each size separately.

To extend the measurements to input sizes beyond 860 we perform a general linear least squares fitting. Here "general" refers to the fact, that the model function is a linear combination of basis functions and their products, which themselves can be nonlinear, but the model function's dependence on the parameters is linear. The set of basis functions is chosen according to the algorithm to be analyzed. E.g. if the algorithm contains nested loops, some of the numbers are likely to grow with $\Theta(n^2)$ or even $\Theta(n^3)$, thus we include n^2 or n^3 in the set of basis functions as well as a linear and constant term. Likewise if the algorithm splits the input repeatedly into halves or operates on a tree-like structures we would expect some numbers to grow with $\Theta\big(\ln(n)\big)$ or $\Theta\big(\sqrt{n}\big)$ and thus we include them in the set of basis functions.

This provides us with a function in n for every rule in the SCFG that estimates the number of times the rule appears in a parse of a "flow of control string" for an

[2]Taking coefficients of a generating function $A(z)$ is denoted by $[z^n]A(z)$ and returns a_n. As this notation may be ambiguous we use $<z^n, A(z)>$ synonymously.

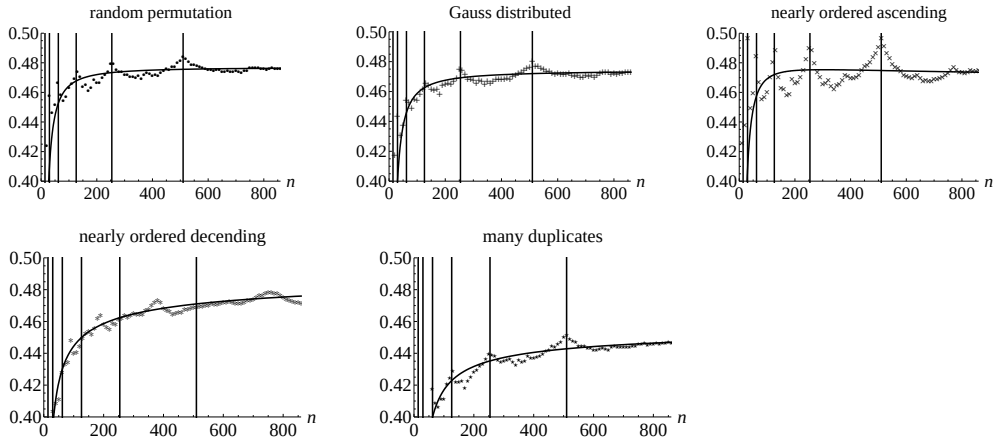

Figure 1: Probability p_{11} exhibits a periodic oscillation, in case of almost reversely ordered sequences the oscillation is shifted.

input of size n. The relative frequencies for each grammar rule are obtained by simply dividing each function by the sum of the functions of every rule with the same left-hand side, thereby ensuring that after the fitting the relative frequencies still sum up to one. Now the training of the SCFGs is complete as the relative frequencies are a maximum likelihood estimate for the unknown probabilities of the grammar rules. As already noted in [9] when comparing the probabilities for the different input distributions we observe that heap sort is quite insusceptible to differently distributed inputs.

The trained probability functions $p_{11}(n)$, $p_{16}(n)$, $p_{21}(n)$, $p_{22}(n)$, $p_{24}(n)$ are now inserted into (2) and dividing the leading coefficients of the nominator and denominator of (2) yields the following asymptotic results:

$$H_{\mathrm{rp}}(n) = 23.65 \cdot n \ln(n) + \mathcal{O}(n) \qquad H_{\mathrm{gd}}(n) = 23.63 \cdot n \ln(n) + \mathcal{O}(n)$$
$$H_{\mathrm{noa}}(n) = 23.72 \cdot n \ln(n) + \mathcal{O}(n) \qquad H_{\mathrm{nod}}(n) = 22.86 \cdot n \ln(n) + \mathcal{O}(n)$$
$$H_{\mathrm{md}}(n) = 22.79 \cdot n \ln(n) + \mathcal{O}(n)$$

In [11] the authors prove that the maxmimum likelihood analysis as performed above, finds the average case result, thus we can compare it to Knuth's empirical result from page 148 in [9]: $23.08 \cdot n \ln(n)$.

A closer look at probability p_{11} reveals, that the tree structure of the heap has a subtle influence, see Figure 1. Whenever the number of keys is near $\sum_{i=0}^{k} 2^i$ for a suitable k, indicated by the vertical lines in the plots, the probability peaks. $\sum_{i=0}^{k} 2^i$ is the number of nodes in a binary tree with height k, that is completely filled. Interestingly for almost reversely ordered inputs the periodic behavior is shifted. Such a periodic behavior is not captured by the functions used for the least squares fitting. We allowed experimentally "Fourier polynomials", that are truncated Fourier series, as basis functions in our fitting process, and we were able to capture the periodic behavior, but as the improvement in the results is less than the variance due to the randomness, this is not worth the effort.

3.1 Further results for other parameters

Changing the homomorphism h allows us to track the average number of key comparions in the exponent of symbol y, here $h(\mathtt{b}) = h(\mathtt{r}) = y$ and $h(\cdot) = \epsilon$ otherwise, we find:

$$C_{\mathrm{rp}}(n) = 2.97 \cdot n\ln(n) - 7.13n + o(n) \qquad C_{\mathrm{gd}}(n) = 2.97 \cdot n\ln(n) - 7.08n + o(n)$$
$$C_{\mathrm{noa}}(n) = 3.00 \cdot n\ln(n) - 6.41n + o(n) \qquad C_{\mathrm{nod}}(n) = 2.84 \cdot n\ln(n) - 7.29n + o(n)$$
$$C_{\mathrm{md}}(n) = 2.86 \cdot n\ln(n) - 7.09n + o(n)$$

The average number of interchanges is found by adjusting the homomorphism h once more: $h(\mathtt{s}) = h(\mathtt{w}) = h(\mathtt{g}) = y$ and $h(\cdot) = \epsilon$ otherwise:

$$I_{\mathrm{rp}}(n) = 1.49 \cdot n\ln(n) - 1.42n + o(n) \qquad I_{\mathrm{gd}}(n) = 1.49 \cdot n\ln(n) - 1.40n + o(n)$$
$$I_{\mathrm{noa}}(n) = 1.50 \cdot n\ln(n) - 0.83n + o(n) \qquad I_{\mathrm{nod}}(n) = 1.42 \cdot n\ln(n) - 1.78n + o(n)$$
$$I_{\mathrm{md}}(n) = 1.43 \cdot n\ln(n) - 1.57n + o(n)$$

For random inputs Knuth reports $2.885 \cdot n \cdot \ln(n) - 3.042 \cdot n - \ln(n)$ for the average number of key comparisons and $1.443 \cdot n \cdot \ln(n) - 0.87 \cdot n - 1$ for the average number of interchanges. In [7] on page 165 the results of a simulation of heapsort implemented in C are reported. The average number of key comparisons is given as $2.885 \cdot n \cdot \ln(n) - 3.0233 \cdot n$ and the average number of interchanges is given as $1.443 \cdot n \cdot \ln(n) - 0.8602 \cdot n$.

3.2 Calculating higher moments

Deriving results for the variance of the different parameters is technically possible, by using the well known formulars for generating functions, but does not yield accurate results. This is due to the fact, that the right-linear grammar introduced before is more general than necessary. In case of heapsort it's language contains infinitely many words that are valid paths in the corresponding automaton, but they would never occur as flows of control for real inputs. For example the loops in the program are finished after a certain number of iterations. However in the corresponding automaton the loops can be traversed arbitrarily often. The probability for this is small but still positive thus the generating function respects all the cases that would never occur for real inputs. In our experiments the calculated variance had the correct asymptotic growth but the coefficient was too large compared to actual variance of the sample inputs used for the training the grammars.

Even if the grammar only reproduces the valid flows-of-control strings there is another cause that hinders the calculation of the variance. Knuth's version of heapsort uses the siftup-subroutine in two phases, the heap creation and the selection phase, but we only assign one set of probabilities to the jumps in the siftup-subroutine, thus averaging the two phases and thereby giving up information. To clarify the before mentioned effects, imagine a source that produces strings of ones and zeros, where both symbols have probability $1/2$ and a second source that produces 0-1-strings of the form 0^k1^k. In both cases we would set the probabilities in a grammar trained on both sets of 0-1-strings to $1/2$, but the variance of the number of ones in the two sets of 0-1-strings for a fixed length is quite different, positive in the first case and zero in the second case. We are not

able to distinguish the two cases by means of right-linear grammarswith a finite number of non-terminals.

A slight improvement in the calculation of the variance was possible by changing the grammar to make a distinction whether the jump before the current one was taken or not. This allows one step dependencies to be taken into account and can be extended to allow for more steps. However the size of the grammar and the number of probabilities that have to be trained increase with every additional step and quickly become not feasible.

4 Conclusions

The probabilities of the rules are simply the probabilities for the various conditional jumps being taken or not. Instead of training them, they could be derived in an abstract manner by inspecting the profile of an algorithm. The profile is just the number of times each instruction is executed. Together with Kirchhoff's law one can deduce the number of times a jump is taken respective not taken. These numbers of course depend on various properties of the inputs, e.g. the number of left-to-right minima in case of permutations. Expressing these properties for non-uniform distributions of the input is the difficult part of a common average-case analysis.

Our new approach to describe the flow of control via stochastic context-free grammars avoids such tedious computations. Deriving exact results for the average performance is easily possible for different parameters and distributions.

Moreover, after a grammar has been trained on a set of inputs the resulting set of probabilities provides full information on the corresponding algorithm. Hence we are able to conserve the result of training by just storing the probabilites. If later we aim for studying a parameter we had not in mind during the training this is no problem at all; we just have to use a different homomorphism for introducing the parameter e.g. a different function $\mathrm{cost}(l_i)$. for weighting the instructions of the algorithm. This is impossible for a plain simulation as no statistics have been collected on unconsidered parameters beforehand. Only an expensive rerun of the entire simulation would allow other parameters to be analyzed in that case.

When based on right-linear grammars, all steps carried out during the maximum likelihood analysis can be performed automatically within a computer-algebra system, with the only exception of capturing the size of the input by introducing the symbol z as some understanding of the algorithm is required. Thus our approach eases the effort required for an average case analysis exceptionally allowing for the consideration of realistic input distributions with unknown distribution functions at the same time.

Further interesting questions are: Which probability distributions can be modeled by a SCFG? Is there any influence on the results obtained by changing the language and/or the grammar assuming that the same objects respective algorithms are encoded? Is the placment of the "marker" z tractable? Are there grammars that allow a good estimation of the variance? All this should be considered in further investigations.

References

[1] T. Chi and S. Geman: Estimation of Probabilistic Context-Free Grammars, Computational Linguistics **24(2)**, 299–305, 1998.

[2] R. Chaudhuri, S. Pham and O. N. Garcia: Solution to an Open Problem on Probabilistic Grammars, IEEE Trans. on Computers **C-32(8)**, 748–750, 1983.

[3] N. Chomsky and M.-P. Schützenberger: The Algebraic Theory of Context-Free Languages, In Computer Programming and Formal Languages, P. Braffort and D. Hirschberg (eds.), North Holland, 118–161, 1963.

[4] R. Durbin, S. Eddy, A. Krogh and G. Mitchison: Biological sequence analysis, Probabilistic models of proteins and nucleic acids, Cambridge Universiy Press, 1998.

[5] P. Flajolet, B. Salvy and P. Zimmermann: Automatic average-case analysis of algorithms, Theor. Comput. Sci. **79(1)**, 37-109, 1991.

[6] P. Flajolet and R. Sedgewick: Analytic Combinatorics, web edition 2007, to be published in 2008 by Cambridge University Press.

[7] G. H. Gonnet and R.Baeza-Yates: Handbook of Algorithms and Data Structures, Addison-Wesley 1991, ISBN: 0-201-41607-7.

[8] T. Huang and K. S. Fu: On Stochastic Context-Free Languages, Information Sciences **(3)**, 201–224, 1971.

[9] Donald E. Knuth: The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition, Addison Wesley, 1998.

[10] W. Kuich: Semirings and Formal Power Series: Their Relevance to Formal Languages and Automata, Chapter 9 in Handbook of Formal Languages, Vol. 1: Word, Language, Grammar, G. Rozenberg and A. Salomaa (eds.), 609–677, 1997, ISBN-10: 3-540-60420-0.

[11] Ulrich Laube and Markus E. Nebel: Maximum Likelihood Analysis of Algorithms and Data Structures, submitted to Theor. Comput. Sci.

[12] Christopher D. Manning and Hinrich Schütze: Foundations of Statistical Natural Language Processing, MIT Press June 1999, ISBN-10: 0-262-13360-1, ISBN-13: 978-0-262-13360-9.

[13] M.-J. Nederhof and G. Satta: Estimation of Consistent Probabilistic Context-free Grammars, In Proc. of the HLT-NAACL, New York, USA, 343–350, 2006.

[14] C. S. Wetherell: Probabilistic Languages: A Review and some Open Questions, Computing Surveys **12(4)**, 361–379, 1980.