Theoretical Computer Science 411 (2010) 188–212

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Maximum likelihood analysis of algorithms and data structures

Ulrich Laube [*], Markus E. Nebel

Fachbereich Informatik, Technische Universität Kaiserslautern, Gottlieb-Daimler-Straße, 67663 Kaiserslautern, Germany

ARTICLE INFO

Article history:
Received 22 August 2008
Received in revised form 26 August 2009
Accepted 15 September 2009
Communicated by H. Prodinger

Keywords:
Analysis of algorithms
Stochastic context free grammars
Generating functions
Maximum likelihood method
Tries
Sorting

ABSTRACT

We present a new approach for an average-case analysis of algorithms and data structures 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 expected size of binary tries as well as by three sorting algorithms and it is compared to the known results that were obtained by traditional techniques. Investigating traditional settings like the random permutation model, we rediscover well-known results formerly derived by pure analytic methods; changing to biased data yields original results. All but one step of our analysis can be automated on top of a computer-algebra system. Thus our new approach can reduce the effort required for an average-case analysis, allowing for the consideration of realistic input distributions with unknown distribution functions at the same time. As a by-product, our approach yields an easy way to generate random combinatorial objects according to various probability distributions.

1. Introduction

The aim of the analysis of algorithms is to derive a precise characterization of the behavior of an algorithm or a data structure. Typically, the execution time or the amount of memory used, given a few assumptions about the probabilistic distribution of the inputs, will be studied. As a worst-case analysis often focuses on pathological configurations, an average-case analysis tends to be more informative, provided that 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. Hence a uniform distribution is often assumed on the data or the resulting combinatorial structures like graphs or trees instead of a more realistic one. Additionally, even if more complicated distributions, such as a Bernoulli model, can be handled mathematically we often have to admit that they fall short of ones expectations.

Our new approach 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. To achieve this we use a suitable formal language that allows us to represent

- the resulting configuration of the data structures (e.g. trees) after processing an input, or
- the flow of control of a program processing the inputs (e.g. sequence of line numbers)

as words over an adequate alphabet. The rules of a grammar that generates this formal language are supplemented with probabilities and this stochastic grammar is trained according to the maximum likelihood principle on the actual inputs/structures/flows from the real application with (hopefully) typical inputs of different sizes. Thereby the stochastic grammar captures the probability distribution present in the sample set of inputs.

According to the ideas of Chomsky and Schützenberger [5], 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

[*] Corresponding author. Tel.: +49 631 205 2651; fax: +49 631 205 2573.
E-mail addresses: laube@informatik.uni-kl.de (U. Laube), nebel@informatik.uni-kl.de (M.E. Nebel).

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.09.025

language are readily available by standard generating function methods. This makes it easy to analyze parameters like the expected size of data structures or the expected running time of an algorithm.

As 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 [11].

We will demonstrate our approach by an analysis of the expected size of random tries in the Bernoulli model (which resembles sequences of random integer keys to be stored in the trie) and compare it with well-known results which were derived by traditional (i.e. purely) analytic methods.

Furthermore, we will apply our method to three sorting algorithms and different probability models. Straight insertion and bubble sort are short, easy to understand and, more important, in the case of a uniform distribution of permutations precise results are available in [19] for comparison. In the case of heapsort there is no complete analysis available, but its insensitivity to changes in the distribution of the inputs should by noticeable in our results.

The rest of the paper is organized as follows. We collect the results and definitions needed in Section 2 and present our work on tries and of the three sorting algorithms in Section 3. In Section 4 we present a proof that an average-case analysis performed with our new approach, equipped with probabilities derived from Knuth's analysis in [19], produces the same results.

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 [16].

Definition 1 (*Stochastic Context-Free Grammar*). A *stochastic context-free grammar* is a quintuple $G = (N, T, R, P, S)$, where $N = \{V_1 = S, V_2, \dots, V_k\}$ is a set of variables or non-terminal symbols, $T = \{a_1, \dots, a_m\}$ is a set of terminal symbols disjoint from N, $R = \{r_1, \dots, r_l\}$ 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

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

holds; that is, the probabilities of every set of rules with the same left-hand side sum up to 1 and thus 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. We write $\alpha \Longrightarrow^{\star} \beta$ if there is a possibly empty sequence of rules $r_{i_1}, \dots, r_{i_k}$ that leads from α to β; that is, $\alpha \overset{r_{i_1}}{\Longrightarrow} \xi_1 \overset{r_{i_2}}{\Longrightarrow} \dots \overset{r_{i_{k-1}}}{\Longrightarrow} \xi_{k-1} \overset{r_{i_k}}{\Longrightarrow} \beta$. Such a derivation is called *left-most* if the left-most non-terminal is always replaced next. The probability p for such a derivation is given by the product of the probabilities of all productions used; that is, $p = p_{i_1} \cdot p_{i_2} \cdot \dots \cdot p_{i_k}$. The probability p_w of a word w 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. Languages and their grammars that only have one left-most derivation per word are called unambiguous. They are important, as the training of the grammars introduced in the next section is especially efficient. In addition, such grammars can be translated into generating functions, as explained in Section 2.3.

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 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.

For some applications normal forms of a grammar are needed. For example, a CKY-Parser only works for grammars in Chomsky normal form. Hung and Fu have demonstrated in [16] that Chomsky and Greibach normal forms can be defined and constructed similarly for SCFGs; they even induce the same probability distribution on the words generated as the original version of the grammar.

[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.

SCFGs are commonly used in the area of natural language processing, see [22,29], and in connection with RNA secondary structure prediction in bioinformatics, see for example [9].

2.2. Training and consistency of stochastic context-free grammars

What we call the training of a grammar is based on the maximum likelihood method invented by R.A. Fisher between 1912 and 1922 [1]. On a fixed sample from a larger population, the maximum likelihood method 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 fits those probabilities in such a way that words generated by the grammar closely match the sample set of words provided for the training.

The conditions for such a trained grammar to be consistent have been investigated by a number of scientists. Booth and Thompson [2] use results from branching processes and matrix theory to show that a SCFG is consistent when the largest eigenvalue of a certain matrix, constructed from the grammar, is less than one. As the exact computation of eigenvalues for large matrices is challenging, the applicability is limited. A conjecture by Wetherell [32] proven in [4] 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 is found in [3]. Sánchez and Benedí [28] extend the results by showing that training a grammar with the classical inside–outside algorithm on a sample yields a consistent SCFG. Chi and Geman [3] show, by a simple counting argument, that training the grammar by Expectation–Maximization yields a consistent SCFG too. 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 [6,24,25] these restrictions 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. The results have been extended in [7] along an information theory approach.

We will use the fact that estimating the probabilities of a stochastic context-free grammar by their relative frequencies yields a maximum likelihood estimate [3]. Counting the relative frequencies is especially efficient for the unambiguous grammars used here, as there is only one left-most derivation (parse tree) to consider.

2.3. Formal power series and SCFGs

In [5], a set of equations is associated with a context-free grammar, by giving a polynomial expression for every non-terminal, derived from its rules. The solution of this set of equations is a formal power series, whose support is the language generated by the grammar. This exhibits a connection between formal languages and formal power series. This transition from a grammar to a set of equations is particular useful with *unambiguous* context-free grammars as it leads to the corresponding structure generating function that counts the number of words of length n; see [12,21].

The original translation from [5] associates a monomial with every right-hand side of a rule, and the monomials of every rule with the same left-hand side are summed up to produce the polynomial for every non-terminal symbol. We extend this translation by including the probabilities of the rules of the SCFG in every monomial. In this case a probability generating function $P(z)$ is generated where the coefficient at z^k is the probability that a word of length k is generated.

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

$$P(z) := \sum_{k} \Pr[X = k] \cdot z^k.$$

We want to exploit the fact that the expectation of the random variable X is easily found by computing

$$E[X] = \left(\frac{\mathrm{d}}{\mathrm{d}z} P(z) \right)\bigg|_{z=1}.$$

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 $S(z) = z^2 S(z)^2 + 1$, 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 $H = \big(\{T\}, \{(,)\}, \{T \to \frac{1}{3} : (T)T, T \to \frac{2}{3} : \epsilon\}, T\big)$ translates into the equation $T(z) = \frac{1}{3} z^2 T(z)^2 + \frac{2}{3}$, which has the solution

$$T(z) = \frac{3 - \sqrt{9 - 8z^2}}{2z^2} = \frac{2}{3} + \frac{4}{27} z^2 + \frac{16}{243} z^4 + \frac{80}{2187} z^6 + \mathcal{O}(z^8).$$

Here the coefficient of z^n is the probability that H produces a word of length n, and verifying that $T(1) = 1$ holds confirms that H is consistent. Checking this for the grammar from Example 1 yields $S(z) = \frac{3-\sqrt{9-8z}}{4}$ and $S(1) = \frac{1}{2}$; therefore it is not consistent.

As seen in Example 2, the interesting information is the coefficients "hidden" in the generating function. Extracting information about the coefficients is possible by singularity analysis. The basic idea is to use a general correspondence between the asymptotic expansion of the generating function near its dominant singularities[2] and the asymptotic expansion of the coefficients of the generating function. This method was introduced by Flajolet and Odlyzko [10]; more on singularity analysis can be found in [12]. The particular type of singularity analysis we use is the $\mathcal{O}$-transfer method. Briefly this works as follows. An asymptotic expansion of a generating function $A(z)$ that has a dominant singularity at $z = 1$ has the form

$$A(z) = h(z) + \mathcal{O}\big(g(z)\big),$$

where $h(z)$ and $g(z)$ belong to a collection of standard functions, for example $\{(1-z)^{-\alpha}\}_{\alpha\in\mathbb{R}}$ in simple cases.

Switching formally to the coefficients,[3] the $\mathcal{O}$-transfer method now provides for different functions $h(z)$ asymptotic expansions of the coefficients $[z^n]h(z)$ and the transfer theorems guarantee that $[z^n]\mathcal{O}\big(g(z)\big) = \mathcal{O}\big([z^n]g(z)\big)$ holds, which is not a triviality. In summary, the $\mathcal{O}$-transfer method permits the following transition:

$$A(z) = h(z) + \mathcal{O}\big(g(z)\big) \quad \rightsquigarrow \quad a_n = h_n + \mathcal{O}(g_n).$$

Our goal is to construct a bivariate probability generating function of the following form:

$$H(z, y) := \sum_{w\in\mathcal{L}(G)} p_w \cdot y^{\lambda(w)} \cdot z^{\gamma(w)}$$

where $\lambda(w)$ describes the "cost" and $\gamma(w)$ is a notion of "size" attributed to the flow of control or data structure encoded in the word w. The mappings λ and γ are typically homomorphisms from the set of terminal symbols of the grammar into the set of integers. In case of data structures the "cost" can be thought of as the number of nodes or other parameters of interest, where the "size" is typically the number of "items" stored in the data structure. With algorithms the "cost" encoded in a flow of control string can be interpreted as the running time, number of comparisons, number of interchanges, etc. The "size" is then the size of the input that caused this flow of control string to be observed. We also say that the variables y and z in the generating function "keep track" of the "cost" and the "size". When looking at a data structure or the flow of control to be encoded as a word, the parts of the data structure or the instructions in the flow of control that contribute to the overall cost of the structure or the execution of the algorithm are considered to be "marked" with y and z. For example, the n leaves of a trie that store the keys are each "marked" with a z; thus we have z^n and we correctly track the size n in the exponent of z. If the internal nodes are each "marked" with a y we track the "cost", here the size of the trie, measured in the number of internal nodes to store n keys in its leaves, in the exponent of the variable y.

Now the *expected* "cost" $H(n)$ of a data structure or of the execution of an algorithm captured in its flow of control of "size" n can again be found by a simple computation:

$$H(n) := \frac{[z^n]H_y(z, 1)}{[z^n]H(z, 1)} = \frac{\displaystyle\sum_{w\in\mathcal{L}_n} p_w \cdot \lambda(w)}{\displaystyle\sum_{w\in\mathcal{L}_n} p_w}$$

where $H_y(z, 1)$ is an easier to read notation than $\frac{\partial}{\partial y}H(z, y)\mid_{y=1}$, the partial derivative of $H(z, y)$ with respect to y, with y set to one after the partial derivation is obtained. The denominator serves as a normalization. This is necessary as a consistent SCFG introduces a probability distribution on the entire language generated, which is all the words of different "sizes". If we only look at a subset, here all words of "size" n, their probabilities no longer sum up to one, thus we have to divide by the probability that we observe an input of "size" n.

3. Maximum likelihood analysis

With the results and definitions presented above, we can introduce the maximum likelihood analysis method. Whether algorithms or data structures are analyzed indeed makes a difference and is explained hereafter. A maximum likelihood analysis involves the following steps:

1. Choose a formal language for the encoding.
2. Build a SCFG for the language.
3. Translate the SCFG into a PGF.
4. Determine the formula for the expectation from the PGF.
5. Acquire a set of sample inputs.
6. Train the SCFG on the sample inputs.

[2] A *singularity* is a point in the complex plane where a function f is not analytic. A *dominant singularity* is one of smallest modulus amongst all singularities.

[3] Taking coefficients of a generating function $A(z)$ is denoted by $[z^n]A(z)$ and returns a_n.

7. Find expressions for the probabilities in dependence of n.
8. Use the probabilities from step 7 in the expression from step 4 to obtain the averages.
9. Store the probabilities to allow the study of other parameters.

Ad 1 & 2. Naturally the encoding differs for algorithms and data structures. In case of the analysis of data structures we assume that n "items" are inserted into the data structure or it is set up in another way and then regarded as static. This static structure is then examined and, for example, the number of nodes, etc. is counted. To do this we have to come up with a one-to-one correspondence between the different "shapes" of the data structure and the words of a formal language. This is similar to the symbolic approach to combinatorial enumeration, where a functional description of operations and elementary combinatorial classes is used to describe combinatorial structures; see Chapter 1 in [12]. As this step depends on the data structure to be analyzed, no automation is possible. It is a creative process and requires an idea for the encoding, just like a traditional average-case analysis requires an idea of how to approach the analysis.

Although algorithms work on data structures we can not use the static view for their analysis too, as it completely disregards the dynamics of data structures maintained by non-trivial algorithms. For example, the heap grows to its full size and shrinks until it is empty when sorting with heapsort. To remedy this we record the flow of control of the algorithm. This of course includes the instructions traversing and updating the data structures and at least implicitly tracks their change over time. The "record" of the instructions executed will be a word of a formal language, which can then be parsed according to a grammar. We will describe a general method to translate a program that is the implementation of an algorithm on a certain machine model into a SCFG in Section 3.2. As a consequence, the results obtained are only valid for the particular model used. This is not necessarily a drawback as it allows us to compare implementations on different machines.

Ad 3. As described in Section 2.3, the rules of a SCFG are translated into a system of polynomials. The solution to this system is the PGF we are looking for. This step can be done efficiently for the analysis of algorithms because we choose the encoding in such a way that a linear system of equations needs to be solved. The solution is then found by simple algebraic manipulations and back substitution.

In the general case, the system of equations might not be linear. Then a closed form of the solution might not be obtainable. However, we can always use Kronecker's method of elimination by resultants [31] to reduce the system of polynomials to a single polynomial. This polynomial can then be used in Newton's polygon method (see Chapter 12.3 in [14] or Chapter 2.2 in [20]) to derive asymptotics for the leading terms in the expansion of the PGF. Exact results would not be obtainable in this case.

Ad 4. As explained in Section 2.3, the expectation is readily available from the PGF. In the case of an asymptotic approximation the Newton polygon method from step 3 has already provided the necessary expressions.

Ad 5. The set of sample inputs for the training of the grammar can originate from different sources. A practitioner can choose among several algorithms for the same problem, and he wants to know which implementation of which algorithm has the best average running time on his computer with his data set. He then can choose the machine model as described above that fits his computer and use his data set to get results that are valid for his environment.

If such a set of sample inputs is not available from an application, or a new algorithm or a variation of an existing algorithm should be analyzed, we can always generate sets of sample inputs according to various distributions. The complexity of this step depends on the type of input that has to be generated for the algorithm. In the case of the tries and sorting algorithms analyzed subsequently, random sequences of integers had to be generated. This can be done in a matter of seconds with Mathematica.

Ad 6. The sample inputs are now used to fill the data structure or run the algorithm on them. This requires an implementation of the data structure or the algorithm which might be provided by the practitioner. A tool for profiling is necessary, to track the shape of the data structure or the flow of control and encode them according to the choices made in step 1.

For this paper we implemented the tries ourselves and recorded the resulting tries, after inserting different numbers of random integer keys, as Motzkin words. For the analysis of algorithms, the virtual machine from the MIX Development Kit was used to run the implementations from [19] with the training inputs and to record the flow of control as words.

The training of the grammars was described in Section 2.2. In the general case we have to parse the words of an unambiguous context-free language which can be done with an Earley-Parser in quadratic time complexity with respect to the length of the words [13]. Counting the rules in the parse tree is efficient because there is only one parse tree per word to consider. The counts are used to calculate the relative frequencies of the rules that are a maximum likelihood estimate of the unknown probabilities governed by the distribution of the inputs.

In case of the sorting algorithms analyzed below, both steps have linear time complexity as we are only using regular languages. For the context-free Motzkin words that will be used to encode the tries an Earley-Parser has been used.

Ad 7. The relative frequencies obtained in the previous step are now used to perform a general linear least squares fitting, because we need estimations of the probabilities for all n. 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's dependence on the parameters is linear. The set of basis functions is chosen according to the algorithm to be analyzed. For example, if the algorithm contains nested loops, some of the numbers are likely to grow with $\mathcal{O}(n^2)$ or even $\mathcal{O}(n^3)$; thus we include n^2 or n^3 in the set of basis functions. Likewise, if the algorithm splits the input repeatedly into halves or operates on a tree-like structure we would expect some numbers to grow with $\mathcal{O}\big(\ln(n)\big)$ or $\mathcal{O}\big(\sqrt{n}\big)$, and thus we include them in the set of basis functions. This set of basis functions is not fixed here. It may be different for every analysis and has to be chosen carefully.

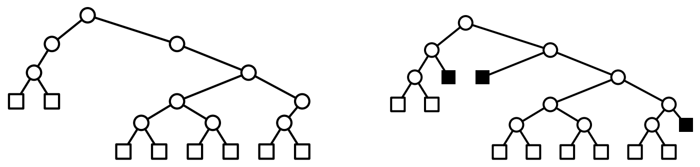

Fig. 1. A trie and its corresponding combinatorial trie.

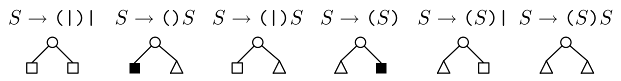

Fig. 2. Encoding of combinatorial tries as Motzkin words.

Ad 8. The expressions obtained in step 4 may be free of any probabilities due to cancellations. Then step 8 is not necessary; in fact, the maximum likelihood analysis then delivers a proof that the parameter is independent of the distribution of the inputs. Furthermore, this result is exact as no training is involved. If only some parts of the expression are without probabilities and we can prove that the other parts do not become negative for any choice of the probabilities, then the maximum likelihood analysis allows us to establish a lower bound.

In a traditional average-case analysis, a probability distribution is assumed and not questioned subsequently. Once the assumption is made the model is fixed. The analysis is then carried out within the model and the results are only valid under the assumptions made. This is only slightly different for the maximum likelihood analysis. Here the probability distribution is not assumed; the model is fixed with the choice of the encoding in step 1, described by the SCFG, and their probabilities, obtained by the maximum likelihood training in step 6. The SCFG serves as a model of the behavior of the algorithm or data structure. The question of how representative this model is is not part of the remaining steps of the analysis.

In the following analysis of tries we will find that a "context-free" model is not "expressive" enough to allow the analysis of parameters like the trie's profile. However, when we analyze the sorting algorithms afterwards, a "regular" model is "strong" enough to allow the derivation of good results for various parameters.

3.1. Maximum likelihood analysis of tries

A *binary trie* is a binary tree that stores a set of keys $K = \{k_1, \dots, k_n\}$ in a special way. The keys are considered in their binary representation as a sequence of 0's and 1's and stored in the leaves only. It can be recursively defined as follows. The trie for an empty set of keys is just a nil-pointer; the trie for a single key is just a leaf containing the key; and a trie for a set of more than one key is an internal node with its left pointer referring to a trie for those keys beginning with 0 and its right pointer referring to a trie for those keys beginning with 1. The leading bit is removed in each recursive step for the purpose of constructing the subtrees.

This construction is only possible if the binary representation of the keys is prefix-free. Then every key is stored in a leaf on the path described by the shortest prefix of its bit pattern which is unique among all keys. Conversely, each leaf can only contain a key whose bit pattern begins with the bits on the path from the root to that leaf. Thus the structure of a trie is independent of the order in which the keys are inserted. In other words, there is a unique trie for any given set of distinct keys. But several distinct sets of keys may map to the same trie structure. Note that a trie is not necessarily an extended binary tree as it may have internal nodes with just one child.

As we are interested in recording the different trie structures that might arise, we use the model of combinatorial tries introduced in [23]. Combinatorial tries are a class of generalized extended binary trees with colored leaves. The leaves may be either black or white, with the restriction that black leaves are always siblings of an internal node. Combinatorial tries now correspond to ordinary tries by interpreting a white leaf as leaf that stores a key, and a black leaf as a nil-pointer in the ordinary trie; see Fig. 1 for an example. Note that it is impossible by definition that two black leaves have the same father or that a white and a black leaf are siblings of each other, as this would correspond to a sub-trie storing no key at all or to a key that could be stored in a lower level of the trie.

We now have to choose a formal language to encode the combinatorial tries. We will use the bar–bracket representation of Motzkin words to encode the tries. Motzkin words and with them Motzkin numbers appear in different branches of mathematics: at least 14 of them are listed in the survey by Donaghey and Shapiro [8]. For example, the paths in a grid from $(0, 0)$ to $(n, 0)$ that do not cross the line $y = 0$ and consist of the steps $\nearrow$, $\rightarrow$ and $\searrow$ are counted by the Motzkin numbers and can be encoded by Motzkin words.

The set of Motzkin words over the alphabet $\{\,(\,,\ |\,,\ )\,\}$ can be defined by two conditions: for any factorization $w = uv$ of a Motzkin word w we require that $|u|_(\geq |u|_)$ and $|w|_(= |w|_)$ holds, where $|w|_y$ denotes the number of times the symbol y appears in the string w. Note that deleting all $|$'s from a Motzkin word produces a semi-Dyck word; see Chapter 10.4 in [13].

The rules of a corresponding stochastic context-free grammar and the structure (sub-tries) they encode is shown in Fig. 2. The intention is that the sub-word which encodes the left sub-trie is enclosed in a pair of brackets representing the root node; the word for the right sub-trie just follows the closing bracket. A vertical bar represents a white leaf which stores a key. Black leaves are encoded by the empty word.

Example 3. The trie from Fig. 1 becomes ((((|) |)) ( ) ((((|) |) (|) |) ((|) |).

Next we need sample inputs for the training of the grammar. Therefore we implemented the trie data structure in C and 500 tries were filled with $n \in \{500, 600, \dots, 1000\}$ integer keys chosen uniformly at random. A traversal of the resulting tries produced their encodings as Motzkin words. We built an Earley-Parser, which provided the frequency counts f_n of the rules in the derivation trees of the words. Their averages $\bar{f_n}$ are used as estimates for the probabilities of the six production rules; according to our initial discussion this yields a SCFG generating the sample set with maximal likelihood. As we are interested in totally random tries; that implies that the tries do not lean to either side, the probabilities were rounded[4] in such a way that left and right subtrees are equally likely. This leads to the following SCFG:

$$S \to \frac{1}{4} : (\,|\,)\,|, \qquad S \to \frac{3}{20} : (\,)S, \qquad S \to \frac{1}{10} : (\,|\,)S,$$
$$S \to \frac{3}{20} : (S), \qquad S \to \frac{1}{10} : (S)\,|, \qquad S \to \frac{1}{4} : (S)S.$$

If the probabilities were sampled from a real-world application as described in Section 1, such a rounding should of course be omitted as the real-world data might be biased.

We can now translate the unambiguous SCFG into a generating function. By "marking" an internal node (a pair of corresponding brackets) with y and a leaf (symbol $|$) with z we get the following implicit representation of the bivariate generating function $S(z, y)$:

$$\frac{1}{4}z^2y + \left(\frac{3}{10}y + \frac{1}{5}zy - 1\right)S(z, y) + \frac{1}{4}yS(z, y)^2 = 0. \tag{1}$$

We could solve the quadratic equation to find an explicit form of $S(z, y)$ or employ Newton's polygon method to find the series expansions of $S(z, 1)$ and $S_y(z, 1)$ at the dominant singularity, which is at $z = 1$, as it is the smallest number where $S(z, 1)$ has a branching point.

Without probabilities this would not work, because the generating function $S(z, 1)$ would enumerate the number of tries with n keys, as we have marked the white leaves by z. But there exist infinitely many tries for a fixed number of white leaves, as the internal nodes do not contribute to the "size" tracked by z and linear chains of internal nodes with a black leaf as one successor may be arbitrarily long.[5] However, as we are using SCFGs the coefficient at z^n of $S(z, 1)$ is the probability that a trie contains n keys. This probability is the sum over the probabilities of all tries with n leaves. The sum is finite, because the more internal nodes a trie possesses the lower its probability is. This is due to the fact that bigger tries need longer encodings, longer words need more grammar rules to be generated, and more grammar rules lead to more probabilities being multiplied.

Similarly, $S_y(z, 1)$ would enumerate the total number of internal nodes in all tries that store n keys, which again would be infinity. But again we are using SCFGs, and therefore we have the expected number of internal nodes in all tries of size n. Dividing $[z^n]S_y(z, 1)$ by $[z^n]S(z, 1)$ yields the expected number of internal nodes as a function of the number of keys n stored in the trie. We will determine the coefficients by means of the $\mathcal{O}$-transfer method. For this purpose we need the expansions of the generating functions around their dominant singularity $z = 1$, which are given by

$$S(z, 1) = 1 - \frac{1}{5}\sqrt{70}\sqrt{1-z} + \frac{2}{5}(1-z) + \frac{3}{100}\sqrt{70}(1-z)^{\frac{3}{2}} + \mathcal{O}\big((1-z)^2\big),$$
$$S_y(z, 1) = \frac{1}{7}\sqrt{70}(1-z)^{-\frac{1}{2}} - 2 + \frac{11}{140}\sqrt{70}\sqrt{1-z} + \mathcal{O}(1-z).$$

Now the $\mathcal{O}$-transfer method allows us to translate these expansions into asymptotics for the probability of a trie containing n keys and the expected number of internal nodes. As the tries were randomly generated we get:

Theorem 1 (*Automatically Derived*[6]). *The expected number of internal nodes of a trie built from n uniformly drawn integer keys is asymptotically given by*

$$\frac{10}{7}n - \frac{10}{7} + \mathcal{O}(n^{-1}), \quad n \to \infty.$$

It is interesting to compare this with the result from Jacquet and Régnier [27] for the Bernoulli model, which was obtained by purely analytical methods.

4 The rounding only took place in the second decimal place or even less significantly.

5 For this reason the tool of Flajolet et al. [11] is not capable of analyzing the trie data structure using the number of keys as measure of size.

6 Our previously mentioned prototypical implementation of an average-case analysis system, called MaLIAn, is already able to prove such results. For the asymptotic number of internal nodes of tries it needs about 2.3 s to compute the asymptotic presented in the theorem. Please note that the Luo system from [11] is not able to work on non-uniformly distributed objects and there is no other system to the best knowledge of the authors.

Theorem 2 (*Régnier, Jacquet [27]*). *The expected number of internal nodes of a trie built from n random integer keys is given by*

$$n \cdot \left(\frac{1}{\ln(2)} + \alpha\big(\log_2(n)\big)\right) + \mathcal{O}(1), \quad n \to \infty$$

where $\alpha\big(\log_2(n)\big)$ is a periodic function of $\log_2(n)$ of small amplitude.

Written in decimal notation, $\frac{1}{\ln(2)} = 1.44269\ldots$ vs. $\frac{10}{7} = 1.42857\ldots$, we find that the constants differ in the second decimal place. However, if we turn to the variance, which can be computed by means of $\frac{\partial^2}{\partial y^2}S(z, y)$, the results are not as close as before. We find:

Theorem 3. *The variance of the number of internal nodes of a trie built from n uniformly drawn integer keys is asymptotically given by*

$$\frac{30}{49}n + \mathcal{O}(1), \quad n \to \infty$$

where $\frac{30}{49} = 0.6122\ldots$ and Kirschenhofer and Prodinger [17] proved, for the Bernoulli model:

Theorem 4 (*Kirschenhofer, Prodinger [17]*). *The variance of the number of internal nodes of a trie built from n random integer keys is given by*

$$n \cdot \big(0.8461\ldots + \beta(\log_2(n))\big) + \mathcal{O}(1), \quad n \to \infty$$

where $\beta\big(\log_2(n)\big)$ is a periodic function of $\log_2(n)$ of small amplitude.

Thus the trained SCFG correctly represents the tries "on average" but not their variance. The reason is that the average degree of branching is tuned by the grammar training in such a way that the expected size of the trie is close to the tries from the training set. But the distribution of the internal nodes on the different levels of the trie is not the same. This is the reason for the poor result on the variance. Thus not every aspect of the training set is correctly captured by the trained SCFG. This is further discussed in the following section.

3.1.1. Other parameters

When we imagine the shape of the average trie described by the trained SCFG, we can conclude that it is quite regular. This is due to the fact that the rules and their probabilities of our SCFG are independent of the depths of the nodes. Every internal node has the same expected number of successors. The training has tuned the probabilities in such a way that the number of internal nodes is correctly estimated but not the shape.

When we want to analyze other parameters, like the height of a trie, its profile (number of internal nodes on different levels) or its external path length (accumulated length of the paths from the white leaves to the root of the trie), then our SCFG approach will perform poorly, as the probabilities cannot change at different distances from the root. For parameters like the external path length, the profile or the height, it does not resemble the true behavior of tries.

Recently, Park et al. [26] studied the profile of tries. Their results suggest that the shape of an average symmetric trie can be roughly described as follows. The first $\mathcal{O}\big(\ln(n)\big)$ levels are completely filled with internal nodes; thus we have a full binary tree. There follows a part where internal nodes and leaves are mixed. The remaining nodes are attached as linear lists to the bottom of the tree.

To better capture the profile of tries we can extend stochastic context-free grammars to *stochastic context-free schemes* (SCFSs) by allowing countably many non-terminals, terminals and rules. The scheme takes the following form:

$$\begin{array}{lll} S_i \to p_{\mathrm{s}}^{(i)} : (\,|\,)\,|, & S_i \to p_{\mathrm{rb}}^{(i)} : (\,)S_{i+1}, & S_i \to p_{\mathrm{lrb}}^{(i)} : (\,|\,)S_{i+1}, \\ S_i \to p_{\mathrm{b}}^{(i)} : (S_{i+1})S_{i+1}, & S_i \to p_{\mathrm{lb}}^{(i)} : (S_{i+1}), & S_i \to p_{\mathrm{lbl}}^{(i)} : (S_{i+1})\,|. \end{array}$$

This allows the probabilities to change with the depths. But how can we train such a scheme? The most obvious way would be to train the SCFS just like the SCFG. The rules which are unused due to the finite height of the tries of the sample, simply get the probability 0.

However, this approach has the drawback that the estimation is poor when n is not near the values that were used for training. To improve the estimation we let the probabilities depend on n and trained sets of probabilities separately for several choices for n. The different sets of probabilities then served as sample points for an extrapolation. There we made use of the observation that all probabilities exhibit the following pattern: as a function of the index we start (small index) with a constant value which later passes over into an active region where the probability may strictly decrease or increase or exhibit a local maximum; at the end a constant behavior shows up again. The location of the active phase of course depends on the size n of the words generated. Accordingly, we used piecewise defined functions together with a custom-made Mathematica procedure for extrapolation. We refrain from presenting the details, for the reader's convenience, and just mention that functions of the pattern $a + b \cdot \ln(n)$ were used to border the active phase, whereas the active phases themselves have been approximated according to patterns like $a + b \cdot i + c \cdot \ln(n)$, i the index of the production and $a, b, c \in \mathbb{Q}$.

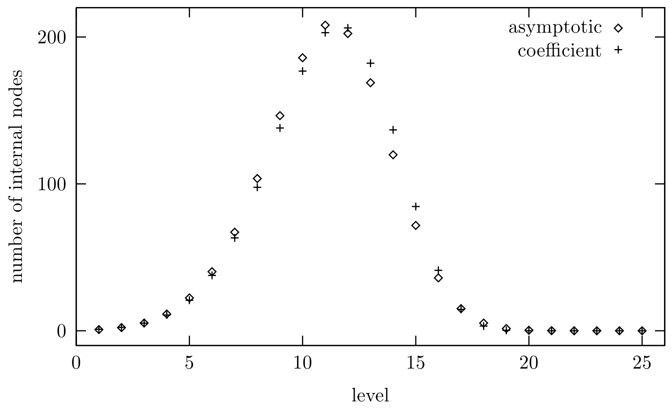

Fig. 3. Comparison of the approximate and exact number of internal nodes of a trie with 1000 keys at different levels.

Table 1
Comparison of two parameters of the sample set with the generated set of tries.

n	Average number of white leaves		Average external path length	
	Sample	Generated	Sample	Generated
100	100	96.09	7.96	8.10
500	500	487.14	10.28	10.41
1000	1000	972.53	11.30	11.40

Now, translating the SCFS into bivariate probability generating functions $S_i(z, y)$ by marking an internal node (a pair of corresponding brackets) with y and a leaf (symbol $|$) by z yields an infinite set of equations:

$$S_i(z, y) = p_s^{(i)} z^2 y + \left(p_{\text{rb}}^{(i)} + p_{\text{lb}}^{(i)} + \left(p_{\text{lrb}}^{(i)} + p_{\text{lbl}}^{(i)}\right)z\right) \cdot y \cdot S_{i+1}(z, y) + p_{\text{b}}^{(i)} y \cdot S_{i+1}(z, y)^2.$$

These equations are nonlinear recursions in i and have thus far resisted several approaches by generating function and operator methods.

Despite this we can still use the probabilities, which we have obtained from training and extrapolation, more directly and derive a recursive formula for the expected profile. This of course has nothing to do with our general approach with SCFGs or SCFSs and generating functions and is specific to this analysis of tries. But it supports our claim that probabilities obtained in a maximum likelihood fashion are useful in the analysis of algorithms and data structures.

The idea is to express the branching factor in the trie via the probabilities:

$$bf(i, n) = p_{\text{rb}}^{(i)}(n) + p_{\text{lb}}^{(i)}(n) + p_{\text{rbl}}^{(i)}(n) + p_{\text{lbl}}^{(i)}(n) + 2p_{\text{b}}^{(i)}(n) = 1 - p_{\text{s}}^{(i)}(n) + p_{\text{b}}^{(i)}(n).$$

Then the expected number of internal nodes on level i in a trie that contains n keys can be written as

$$T(1, n) = 1 \qquad T(i, n) = T(i-1, n) \cdot bf(i, n) = \prod_{j=1}^{i-1} bf(j, n).$$

Attacking this recursion with generating functions in order to get an asymptotic for the product yields a rather complicated expression. However, extracting information about the coefficient from the generating function with the saddle-point method is possible but challenging, as the exact location of the saddle point can only be approximated. Even with the approximate location an useful asymptotic could be found. In Fig. 3, the asymptotic number of internal nodes of a trie that stores 1000 keys is shown together with the exact numbers for comparison.

Another way to evaluate the SCFS approach is the following: Generating random tries according to the trained scheme and comparing their structural properties with the training set. Consequently, we generated 300 tries each with the trained SCFS for $n = 100$, 500 and 1000 and compared their number of white leaves and their average external path length against the sample set of tries. Table 1 shows some results.

The plot in Fig. 4 compares the external profile of the sample and the generated set of tries. The external profile of the sample set is shown together with the expected external profile of (according to our grammar) randomly generated tries for $n = 1000$. The expected external profile replicates the profile of the sample set quite well.

We can conclude that SCFSs are capable of realistically capturing the structural behavior of tries whereas any attempt to make use of this for a quantitative analysis has failed so far. They provide a way to generate random combinatorial objects (tries in our case) according to a probability distribution observed in practice.

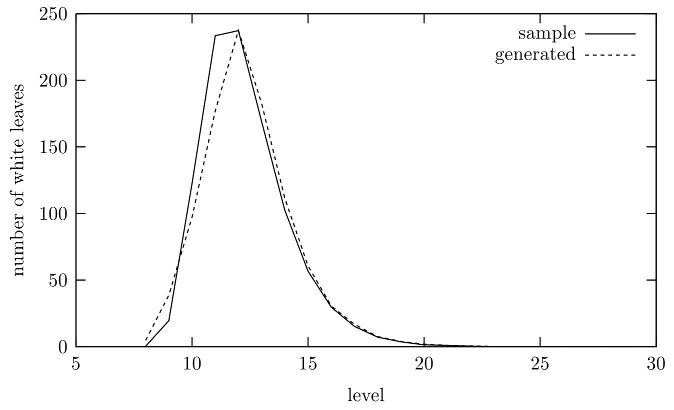

Fig. 4. Comparison of the external profile of the sample set with the generated set of tries.

3.2. Maximum likelihood analysis of sorting algorithms

In the following section we would like to show how algorithms can be analyzed with our maximum likelihood approach. As the running time of an algorithm depends on the actual implementation and the machine model used, we opt for Knuth's MIX computer as found in [18] and the implementations from [19], as they are well documented and a thorough theoretical analysis is available for later comparison. However, the maximum likelihood analysis method is not connected to any specific machine model. It works as generally as any other method for performing an average-case analysis.

When analyzing data structures we would have to find a new encoding for every new data structure we want to analyze. In the case of the analysis of algorithms we will describe a general encoding that allows us to derive the SCFG for an algorithm automatically. When analyzing the tries, the Motzkin words encoded the possible structures of the trees. Now the words will describe the possible flows of control. Basically the words will describe the sequence of the line numbers ℓ_i of the instructions executed. The algorithms in Knuth's books are written in MIXAL, the MIX assembly language, and are easily translated into the rules of a SCFG by the following three rules:

1. An unconditional jump from line i to line j becomes $L_i \to 1 : \ell_i L_j$.
2. A conditional jump instruction in line i that may jump to line j becomes $L_i \to p_i : \ell_i L_{i+1} \mid 1 - p_i : \ell_i L_j$ because the execution continues with the next instruction (jump not taken) or the jump is performed.
3. All other instructions yield $L_i \to 1 : \ell_i L_{i+1}$.

When m is the number of 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). As the grammars are right-linear we obtain regular languages over the alphabet $T = \{\epsilon, \ell_1, \dots, \ell_m\}$ and the non-terminals $N = \{L_1, \dots, L_{m+1}\}$. This is noteworthy as regular languages always have rational structure generating functions; see [21]. The whole translation can be done in a single pass over the program to be analyzed. We process the SCFGs further to reduce their size, by removing the chain rules.

Next, we define the homomorphism $h : N \cup T \to \{z, y\} \cup N$ and extend it to words from $(N \cup T)^\star$ as $h(\omega_j) := y^{\lambda(\omega_j)} \cdot z^{\gamma(\omega_j)} \cdot \alpha_j$, where $\alpha_j \in N^\star$ are all non-terminals that appear in ω_j. With it we translate the rules $L_i \to p_j : \omega_j$, $j \in \{j | r_j \in R_i\}$ into

$$L_i = \sum_{\{j|r_j \in R_i\}} p_j \cdot h(\omega_j) = \sum_{\{j|r_j \in R_i\}} p_j \cdot y^{\lambda(\omega_j)} \cdot z^{\gamma(\omega_j)} \cdot \alpha_j.$$

Thus we sum up the different right-hand sides, which are regarded as commutative now, for each non-terminal. Thus we get one equation per non-terminal, in which we keep the other non-terminals and substitute the line numbers ℓ_i's by their "costs" $y^{\lambda(\ell_i)}$, as we do not care about the actual line number but about the cost it implies. Depending on how we define λ we can analyze different parameters. In the case of running time, Table 2 shows the number of clock cycles needed by each of the MIXAL instructions. The complete table is found on page 140f in [18]. In our case the function λ performs a look-up in this table. If we just want to analyze the number of comparisons or interchanges we adjust λ accordingly. This allows us to introduce the parameter (here the running time) which we are interested in into the equations and thus into the generating function that we want to build.

We can find the generating function by solving this system of equations, as the equations are of course linear, and after eliminating the variables (the non-terminals) we get a rational generating function in z and y, as mentioned before.

In the examples following this introduction, the presentation is less formal. We do not work with the line numbers directly. We abbreviate blocks of instructions that are always executed together with lower-case letters. And we introduce the variable z in the grammar rules while building the grammar and before translating the rest of the right-hand sides. This is done to emphasize how we use one or more z's in the grammar to keep track of size of the input. Remember that 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.

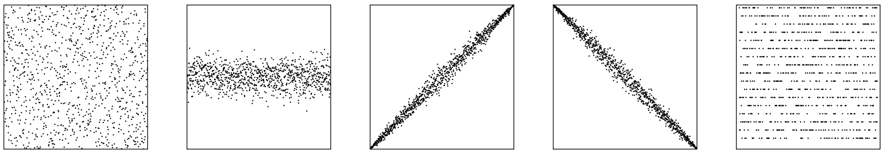

Fig. 5. From left to right: random permutation, Gauss distributed, nearly ordered ascending, nearly ordered descending, many duplicates. In each plot we show at $x = i$ the value of the i-th element of the sequence.

Table 2
Running times of the MIXAL instructions.

NOP	1	ADD	2	SUB	2	MUL	10	DIV	12	NUM	10	SLA	2	SRA	2
JSJ	1	FADD	2	FSUB	2	FMUL	10	FDIV	12	CHR	10	SLRAX	2	SRAX	2
FCMP	2							MOVE	1 + 2F	HLT	10	SLC	2	SRC	2
LDA	2	LD1	2	LD2	2	LD3	2	LD4	2	LD5	2	LD6	2	LDX	2
LDAN	2	LD1N	2	LD2N	2	LD3N	2	LD4N	2	LD5N	2	LD6N	2	LDXN	2
STA	2	ST1	2	ST2	2	ST3	2	ST4	2	ST5	2	ST6	2	STX	2
STJ	2	STZ	2	JBUS	1	IOC	1 + T	IN	1 + T	OUT	1 + T	JRED	1	JMP	1
JOV	1	JNOV	1	JL	1	JLE	1	JG	1	JGE	1	JE	1	JNE	1
INCA	1	INC1	1	INC2	1	INC3	1	INC4	1	INC5	1	INC6	1	INCX	1
DECA	1	DEC1	1	DEC2	1	DEC3	1	DEC4	1	DEC5	1	DEC6	1	DECX	1
ENTA	1	ENT1	1	ENT2	1	ENT3	1	ENT4	1	ENT5	1	ENT6	1	ENTX	1
ENNA	1	ENN1	1	ENN2	1	ENN3	1	ENN4	1	ENN5	1	ENN6	1	ENNX	1
CMPA	2	CMP1	2	CMP2	2	CMP3	2	CMP4	2	CMP5	2	CMP6	2	CMPX	2

Thus we have to make sure that the correct number of z's is inserted. A trie stores the keys in its white leaves; thus we "mark" them with a z. For the sorting algorithms to be analyzed, we "mark" each of the instructions that moves the keys in their final positions with a z. When the sorting is done 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.

The solution to the equation system still contains the probabilities p_j, and we can now adjust them according to the actual distribution of the inputs. As we want to use the maximum likelihood principle to train the probabilities from sample inputs, we generate five sample 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 100 sequences with length 10 to 860 in steps of 10. Fig. 5 shows one sequence from each of the five sets. The sequences $s_{d,s}^{(n)}$ with $d \in \{\texttt{rp}, \texttt{gd}, \texttt{noa}, \texttt{nod}, \texttt{md}\}$, $n \in \{10, 20, \ldots, 860\}$, $1 \leq s \leq 100$ were randomly generated and sorted by three algorithms running on a MIX virtual machine from the GNU MIX Development Kit. For every sequence and each of the three sorting algorithms considered below, the flow of control was recorded as a string $w_{d,s}^{(n)}$ of the line numbers. A simple top-down parser written by the authors was used to parse the strings in linear time, resulting in one parse tree $\tau_{d,s}^{(n)}$ per word. The number of occurrences of the rules r_j in the parse trees was counted while parsing all inputs of the same size and distribution. Formally, let $\#r_j(\tau)$ be the number of times the rule r_j is used in the parse tree τ; then the counts are $c_{d,j}^{(n)} = \sum_{s=1}^{100} \#r_j\big(\tau_{d,s}^{(n)}\big)$.

The counts $c_{d,j}^{(n)}$ were then transferred into Mathematica to 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's dependence on the parameters is linear. The set of basis functions was chosen after inspecting the algorithm to be analyzed. For example, if the algorithm contains nested loops, some of the numbers are likely to grow with $\mathcal{O}(n^2)$ or even $\mathcal{O}(n^3)$; thus we include n^2 or n^3 in the set of basis functions. Likewise, if the algorithm splits the input repeatedly into halves or operates on a tree-like structure we would expect some numbers to grow with $\mathcal{O}\big(\ln(n)\big)$ or $\mathcal{O}\big(\sqrt{n}\big)$, and thus we include them in the set of basis functions.

The fitting provides us with functions $c_{d,j}(n)$ that allow us to estimate for every rule r_j the number of times it appears in a parse of a "flow of control string" for an input of size n and distribution d. We then obtain the "relative frequency" estimator for each rule $r_j = (V_i, \omega_j)$ by simply dividing each function by the sum of the functions of every rule with the same left-hand side, thereby ensuring that the estimates still sum up to one. Formally, we have

$$\hat{p}_{d,j}(n) = \frac{c_{d,j}(n)}{\displaystyle\sum_{\{k \mid r_k \in R_i, r_j = (V_i, \omega_j)\}} c_{d,k}(n)}$$

which is the maximum likelihood estimator as presented in [3]. The training of the SCFGs is now complete, as those relative frequencies $\hat{p}_{d,j}(n)$ are maximum likelihood estimates for the unknown probabilities of the grammar rules.

3.2.1. Fisher information, standard error and confidence intervals

The estimates computed for the probabilities of productions obviously depend on the sample used for training. But then, how can we be sure that our estimates are appropriate and do not distort the results of our analysis? This question will be addressed in what follows.

For a SCFG $G = (N, T, R, P, S)$ with the set of rules $R = \{r_1, \ldots, r_l\}$, and a word $w \in \mathcal{L}(G)$, we call the probability $L_w(P) = \Pr[w \mid P]$ as a function in P the *likelihood* of w. For a set of words $W \subseteq \mathcal{L}(G)$, the likelihood is given by $L_W(P) = \prod_{w\in W} \Pr[w \mid P]$. The maximum likelihood training of the grammar has tuned P in such a way that the sample W has the largest probability possible (which formally means considering the roots of the first derivative of $L_W(P)$ with respect to P). It proves convenient to consider the so-called *log likelihood*

$$l_W(P) := \ln\big(L_W(P)\big) = \ln\left(\prod_{w\in W} \Pr[w \mid P]\right) = \sum_{w\in W} \ln\big(\Pr[w \mid P]\big)$$

which provides the same maximum with less effort. When G is unambiguous we can write $\Pr[w \mid P]$ as $\Pr[w \mid P] = \prod_{1\le i\le l} P(r_i)^{\#r_i(\tau(w))}$, where $\#r_i\big(\tau(w)\big)$ is the number of occurrences of production r_i in the parse tree $\tau(w)$ of the word w. This implies

$$l_W(P) = \sum_{w\in W}\sum_{1\le i\le l} \#r_i\big(\tau(w)\big)\ln(P(r_i)). \tag{2}$$

It is well known from statistics that the *Fisher information* given by the matrix

$$\big[\mathbf{I}(P)\big]_{i,j} := -\frac{\partial^2}{\partial r_i \partial r_j} l_W(P),\ 1 \le i, j \le l,$$

can be used to quantify the standard error $\mathrm{se}\big(\hat{P}(r_i)\big)$ of a maximum likelihood estimator $\hat{P}(r_i)$ for parameter $P(r_i)$. The standard error can be computed by

$$\mathrm{se}\big(\hat{P}(r_i)\big) = \sqrt{\big[\mathbf{I}^{-1}(\hat{P})\big]_{i,i}}$$

where $\mathbf{I}^{-1}(\hat{P})$ the inverse matrix of the observed Fisher information matrix $\mathbf{I}(\hat{P})$. Using (2) we find

$$-\frac{\partial^2}{\partial \hat{P}(r_i)\partial \hat{P}(r_j)} l_W(\hat{P}) = \begin{cases} 0 & \text{for } i \ne j, \\ \frac{\#r_i(W)}{\hat{P}(r_i)^2} & \text{otherwise,} \end{cases}$$

where $\#r_i(W) = \sum_{w\in W} \#r_i\big(\tau(w)\big)$ is the total number of occurrences of the rule r_i in all parse trees of all words in the set W. Having only non-zero elements on the diagonal, inverting this matrix is easy, giving rise to reciprocal values on the diagonal and leaving the rest of the matrix unchanged.

Let

$$\#_j(W) = \sum_{w\in W}\sum_{\{k \mid r_k\in R_i, r_j=(V_i,\omega_j)\}} \#r_k\big(\tau(w)\big)$$

be the number of times the non-terminal V_i from rule $r_j(V_i, \omega_j)$ appears in the parse trees of the words in the set W. The fraction of the number of times the rules r_i is used $\#r_i(W)$ and the number of times it could be used $\#_i(W)$ is the relative frequency, which is a maximum likelihood estimator

$$\hat{P}(r_i) = \frac{\#r_i(W)}{\#_i(W)}$$

for $P(r_i)$, as pointed out in the previous section. Accordingly, the standard error of the estimator $\hat{P}(r_i)$ is given by

$$\mathrm{se}\big(\hat{P}(r_i)\big) = \frac{\sqrt{\#r_i(W)}}{\#_i(W)};$$

i.e. the error is given by the square root of the number of observed applications of production r_i divided by the number of applications of productions having the same left-hand side as r_i. As a consequence, the standard error of every probability of G estimated this way decreases at least like $1/\sqrt{m}$ to zero for m, the number of observations of a given non-terminal symbol within the derivation trees of our sample.

From the standard error a confidence interval can be computed. Denoting by $z_{1-\alpha/2}$ the $(1-\alpha/2)$ quantile of the standard normal distribution, the real parameter $P(r_i)$ lies within the interval

$$\Big[\hat{P}(r_i) - \mathrm{se}\big(\hat{P}(r_i)\big)\cdot z_{1-\alpha/2}, \hat{P}(r_i) + \mathrm{se}\big(\hat{P}(r_i)\big)\cdot z_{1-\alpha/2}\Big],$$

with probability $1-\alpha$. For the sake of a mostly self-contained presentation we mention while passing that for a 99% confidence interval $z_{1-\alpha/2}$ is given by 2.5758, and for a 95% confidence interval by 1.9600.

In Sections 3.2.2–3.2.4 we present a maximum likelihood analysis of straight insertion sort, bubble sort and heapsort.

Line no.	LOC	OP	ADDRESS	Cost	Rule	
01	START	ENT1	2-N	1	$L_1 \to 1 : \ell_1 L_2$	$L_1 \to \mathtt{i}L_2$
02	2H	LDA	INPUT+N,1	2	$L_2 \to 1 : \ell_2 L_3$	$L_2 \to \mathtt{l}L_4$
03		ENT2	N-1,1	1	$L_3 \to 1 : \ell_3 L_4$	
04	3H	CMPA	INPUT,2	2	$L_4 \to 1 : \ell_4 L_5$	$L_4 \to \mathtt{c}L_5$
05		JGE	5F	1	$L_5 \to p_5 : \ell_5 L_6 \mid 1-p_5 : \ell_5 L_{10}$	$L_5 \to p_5 : \mathtt{j}L_6 \mid 1-p_5 : \mathtt{j}L_{10}$
06	4H	LDX	INPUT,2	2	$L_6 \to 1 : \ell_6 L_7$	$L_6 \to \mathtt{m}L_9$
07		STX	INPUT+1,2	2	$L_7 \to 1 : \ell_7 L_8$	
08		DEC2	1	1	$L_8 \to 1 : \ell_8 L_9$	
09		J2P	3B	1	$L_9 \to p_9 : \ell_9 L_{10} \mid 1-p_9 : \ell_9 L_4$	$L_9 \to p_9 : \mathtt{j}L_{10} \mid 1-p_9 : \mathtt{j}L_4$
10	5H	STA	INPUT+1,2	2	$L_{10} \to 1 : \ell_{10} L_{11}$	$L_{10} \to \mathtt{s}L_{12}$
11		INC1	1	1	$L_{11} \to 1 : \ell_{11} L_{12}$	
12		J1NP	2B	1	$L_{12} \to p_{12} : \ell_{12} L_{13} \mid 1-p_{12} : \ell_{12} L_2$	$L_{12} \to p_{12} : \mathtt{je} \mid 1-p_{12} : \mathtt{j}L_2$
13				0	$L_{13} \to 1 : \epsilon$	

Fig. 6. Straight insertion sort, Program S on page 81 in [19].

3.2.2. *Straight insertion sort*

The implementation of straight insertion sort in MIXAL is shown in Fig. 6, as well as its translation into a grammar.

Blocks of instructions that are always executed sequentially are abbreviated by lower-case letters. Removing the chain rules reduces the number of rules to a pair of rules for every conditional jump in the program:

$$L_1 \to 1 : \mathtt{ilc}L_5$$
$$L_5 \to p_5 : \mathtt{jm}L_9 \mid 1 - p_5 : \mathtt{j}L_{10}$$
$$L_9 \to p_9 : \mathtt{j}L_{10} \mid 1 - p_9 : \mathtt{jc}L_5$$
$$L_{10} \to p_{12} : \mathtt{sje} \mid 1 - p_{12} : \mathtt{sjlc}L_5.$$

As the first key can be seen as being already in the correct position and $n - 1$ keys remain to be sorted, we put one z in the first rule. The second z is placed at lines 10 and 11 because here the key is stored in its final position in the array. Now we replace the lower-case letters by their "cost", $i \to y^1$, $l \to y^3$, $c \to y^2$, $j \to y^1$, $m \to y^5$, $s \to y^4$, $e \to y^1$ and turn the grammar into a system of equations:

$$\begin{array}{ll}
L_1 \to 1 : y^6 z L_5 & L_1 = y^6 z L_5 \\
L_5 \to p_5 : y^6 L_9 \mid 1 - p_5 : y L_{10} & L_5 = p_5 \cdot y^6 \cdot L_9 + (1 - p_5) \cdot y \cdot L_{10} \\
L_9 \to p_9 : y L_{10} \mid 1 - p_9 : y^3 L_5 & L_9 = p_9 \cdot y \cdot L_{10} + (1 - p_9) \cdot y^3 \cdot L_5 \\
L_{10} \to p_{12} : y^4 z \mid 1 - p_{12} : y^9 z L_5 & L_{10} = p_{12} \cdot y^4 \cdot z + (1 - p_{12}) \cdot y^9 \cdot z \cdot L_5.
\end{array}$$

Back substitution yields the solution:

$$S(z, y) = \frac{p_{12} y^{11} \big(1 - p_5 (1 - p_9) y^6\big) z^2}{1 - p_5 \big(1 - p_9\big) y^9 - \Big(p_5 p_9 \big(1 - p_{12}\big) y^{16} + \big(1 - p_5\big)\big(1 - p_{12}\big) y^{10}\Big) z}.$$

Building the quotient of the expected cost of all inputs of size n and the probability for an input of size n, we get

$$S(n) := \frac{[z^n] S_y(z, 1)}{[z^n] S(z, 1)} = \frac{10n - 9 - np_5 + (7n - 6) p_5 p_9}{1 - p_5 \big(1 - p_9\big)}. \tag{3}$$

It is interesting to note that the denominator $[z^n]S(z, 1)$ is $p_{12} \cdot (1 - p_{12})^{n-2}$, which is just the probability to leave the outer loop after the remaining $n - 1$ keys have been sorted. Remarkably, the term $10n - 9$ without probabilities in the nominator is just the best-case running time of straight insertion sort, occurring for a sorted sequence.

First let us use Knuth's analysis on page 81f in [19] and derive the average case under the assumption that all inputs are equally likely. The average number of times a MIXAL instruction is executed when straight insertion sort sorts a sequence depends on N, the size of the input, $A(N)$, the average number of left-to-right minima in the sequence, and $B(N)$, the average number of inversions in the sequence.

$$A(N) = H_N - 1 \qquad B(N) = \frac{N^2 - N}{4}$$

H_N is the N-th harmonic number. Expressions for the probabilities p_5, p_9 and p_{12} can be easily derived:

$$p_5(N) = \frac{B(N)}{B(N) + N - 1 - A(N)}, \qquad p_9(N) = \frac{A(N)}{B(N)}, \qquad p_{12}(N) = \frac{1}{N - 1}. \tag{4}$$

Inserting (4) in (3) yields Knuth's original result for the average case of uniformly distributed inputs [7]

$$S_{\text{uni}}(N) = 2.25N^2 + 7.75N - 3H_N - 6 \approx 2.25N^2 + 7.75N - 3\ln(N) - 7.73.$$

[7] It is customary to speak of the *random permutation model* in the case that every permutation of the same size is assumed to be equally likely. However, we will call it the uniform model or uniform case for we need to make a distinction from the case of a stochastic grammar trained on (uniformly distributed) random permutations.

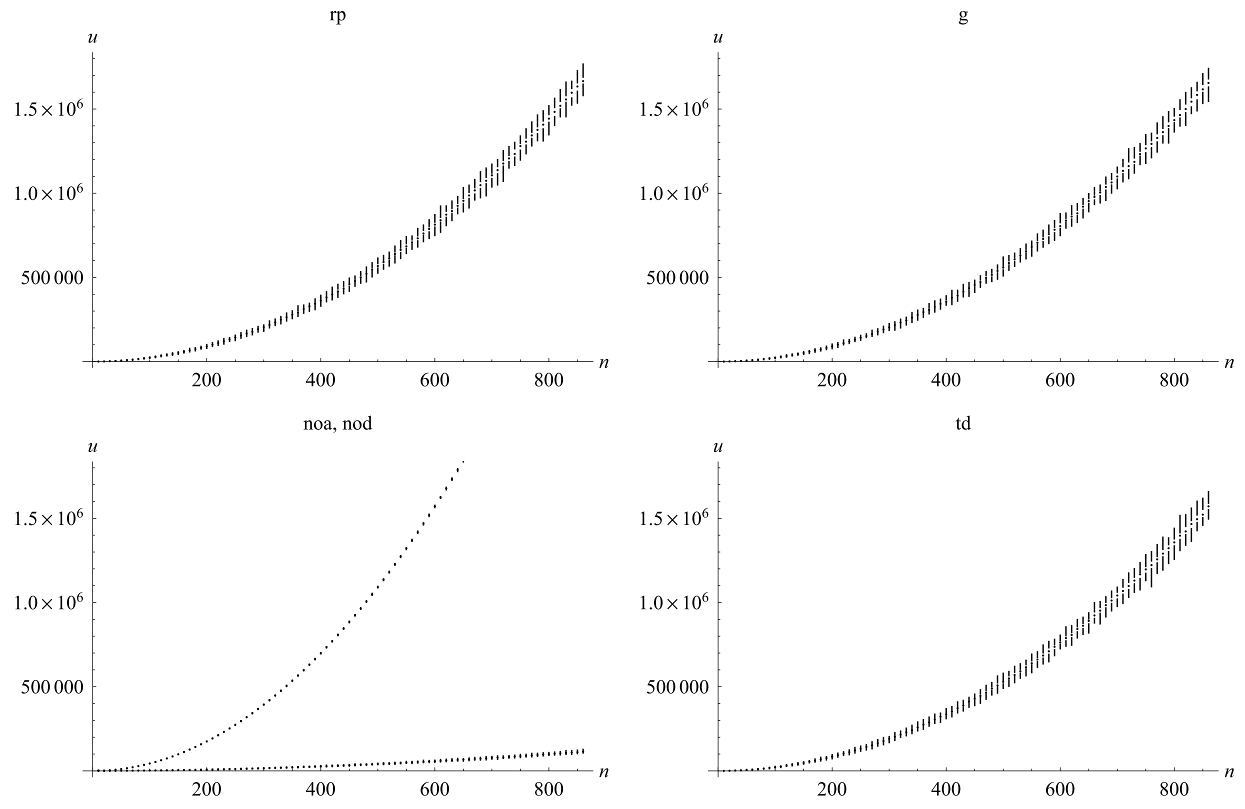

Fig. 7. The average running time of straight insertion sort observed while training the grammar.

This is no coincidence; in Section 4 we prove that an average-case analysis performed in Knuth's fashion, that is weighting each line with its expected number of executions, and a maximum likelihood analysis with probabilities derived from Knuth's counts yield the same results.

Now, instead of using the theoretically derived expression for the probabilities we will use the functions $p_{d,5}(n)$ and $p_{d,9}(n)$ built by the training of the SCFG on the five types of input $d \in \{\mathtt{rp}, \mathtt{g}, \mathtt{noa}, \mathtt{nod}, \mathtt{md}\}$, as described in Section 3.2. Fig. 7 shows the average running times as observed while training the grammar on the five sets of sequences. Following [30], the lines span the range from the minimum to the maximum observed, while the gap in each line around the median indicates the interquartile range, which represents the central 50% of the data. The susceptibility of straight insertion sort to different distributions of the inputs is clearly noticeable. Nearly ordered sequences are processed a lot faster when they are almost in ascending order and respectively a lot slower when they are in descending order (lower left plot). A lot of duplicate keys have only a small impact compared to random permutations (plots on the right).

The model function $a_5 \cdot n^2 + b_5 \cdot n + c_5$ was fitted to the counts $c_{d,5}^{(n)}$ and $a_9 \cdot n + b_9 \cdot \ln(n) + c_9$ was fitted to the counts $c_{d,9}^{(n)}$, with $a_i, b_i, c_i \in \mathbb{Q}$ being the coefficients of the linear combination of base functions as determined by the least squares fitting.

Fig. 8 shows exemplarily the 95% and 99% confidence bands for the two maximum likelihood estimators $\hat{p}_{\mathtt{g},5}$ and $\hat{p}_{\mathtt{nod},5}$. We have used the confidence bands to guide the training of the grammars, i.e. the size of the training sets. In all cases the confidence intervals become smaller with increasing input size. This follows from the observation above that the standard error decreases like $1/\sqrt{m}$ when m is the number of observations of a given non-terminal symbol. Bigger inputs take more time and thus have longer flow-of-control strings, which in turn have bigger derivation trees that contain more grammar rules; thus more non-terminals are observed. The effect is that "slower" algorithms can have smaller training sets as they run longer on each input.

Now we simply substitute the functions $\hat{p}_{d,5}(n)$ and $\hat{p}_{d,9}(n)$ in (3) for p_5 and p_9 to obtain the results for the different distributions. Dividing the leading coefficients of the nominator and denominator of (3) provides the following asymptotic results for the average-case running time of straight insertion sort for the five probability models of the inputs:

$$S_{\mathrm{rp}}(n) = 2.25 \cdot n^2 + \mathcal{O}(n \ln(n)) \qquad S_{\mathrm{gd}}(n) = 2.23 \cdot n^2 + \mathcal{O}(n \ln(n))$$
$$S_{\mathrm{noa}}(n) = 0.14 \cdot n^2 + \mathcal{O}(n \ln(n)) \qquad S_{\mathrm{nod}}(n) = 4.44 \cdot n^2 + \mathcal{O}(n \ln(n))$$
$$S_{\mathrm{md}}(n) = 2.14 \cdot n^2 + \mathcal{O}(n \ln(n)).$$

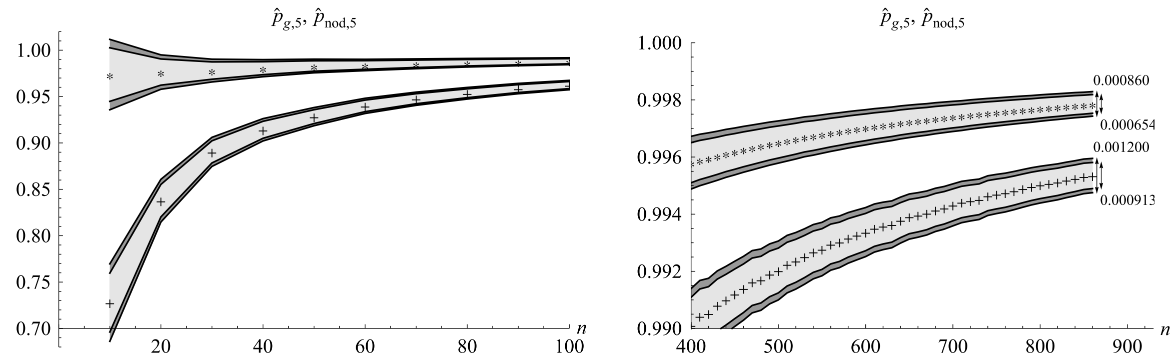

Fig. 8. 99% and 95% confidence bands of the maximum likelihood estimators $\hat{p}_{g,5}$ (symbol $+$) and $\hat{p}_{\text{nod},5}$ (symbol $*$).

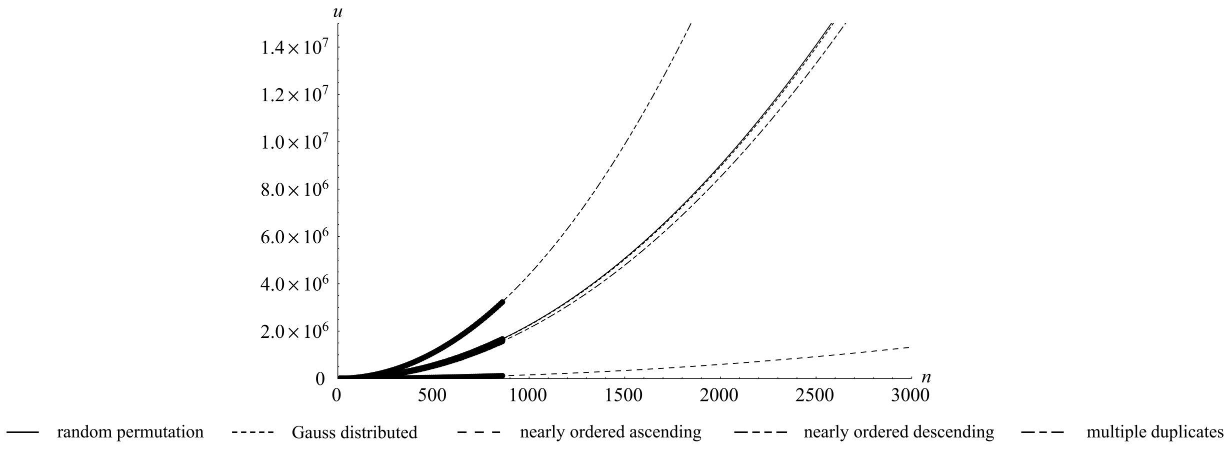

Fig. 9. The average running time of straight insertion sort obtained per maximum likelihood analysis.

The leading coefficient for the average running time is according to Knuth's analysis 2.25, and at most 4.5 in the worst case. The results above allow the reassuring observation that distributions, which according to our intuition favour worst-case inputs, also have large leading coefficients, here 4.44 for nearly ordered sequences in descending order, that are near the theoretical worst case, 4.5. Likewise for the average case.

Fig. 9 shows the final results of the maximum likelihood analysis. The thick part of the curves between 0 and 860 indicates the range for which training data was available.

Comparing the original result of the uniform case with the result for random permutations obtained with the maximum likelihood analysis, we see that maximum likelihood analysis determines the same leading term for the expected running time. Regarding the lower-order terms, however, the situation is different, as the maximum likelihood analysis introduces lower-order terms in the results, which are not present in the original one – here $n\ln(n)$ and $\ln(n)^2$ terms – which are probably due to some cancellations not happening because of small errors in the trained probabilities. This explains the appearance of $\mathcal{O}(n\ln(n))$-terms in the results above. It should be noted that the coefficients of those lower-order terms which are not present in the original result are quite small.

Applying the maximum likelihood analysis method to straight insertion sort was straightforward. In the next example an additional idea is needed.

3.2.3. *Bubble sort*

Fig. 10 shows the implementation of bubble sort in MIXAL together with its translation into a SCFG.

Typically, some tasks consist of several instructions, which are always executed successively. We abbreviate those blocks of instructions as indicated above. As intended, a word now describes the flow of control; it might start with

```
1ibcnbcsmnbcsmn...
```

which can be read as Initialize, adjust Bound, Compare, Next, adjust Bound, Compare, Swap, Mark, Next, adjust Bound, Compare, Swap, Mark, Next, ... As a result, we get the following grammar, which contains a pair of rules for every conditional jump:

$$L_1 \to 1 : \mathtt{1i}L_{14}$$

Line no.	LOC	OP	ADDRESS	Cost	Rule	
01	START	ENT1	N	1	$L_1 \to 1 : \ell_1 L_2$	$L_1 \to \mathtt{1} L_2$
02	1H	ST1	BOUND(1:2)	2	$L_2 \to 1 : \ell_2 L_3$	$L_2 \to \mathtt{i} L_{14}$
03		ENT2	1	1	$L_3 \to 1 : \ell_3 L_4$	
04		ENT1	0	1	$L_4 \to 1 : \ell_4 L_5$	
05		JMP	BOUND	1	$L_5 \to 1 : \ell_5 L_{14}$	
06	3H	LDA	INPUT,2	2	$L_6 \to 1 : \ell_6 L_7$	$L_6 \to p_8 : \mathtt{c} L_9 \mid 1 - p_8 : \mathtt{c} L_{13}$
07		CMPA	INPUT+1,2	2	$L_7 \to 1 : \ell_7 L_8$	
08		JLE	2F	1	$L_8 \to p_8 : \ell_8 L_9 \mid 1 - p_8 : \ell_8 L_{13}$	
09		LDX	INPUT+1,2	2	$L_9 \to 1 : \ell_9 L_{10}$	$L_9 \to \mathtt{sm} L_{13}$
10		STX	INPUT,2	2	$L_{10} \to 1 : \ell_{10} L_{11}$	
11		STA	INPUT+1,2	2	$L_{11} \to 1 : \ell_{11} L_{12}$	
12		ENT1	0	1	$L_{12} \to 1 : \ell_{12} L_{13}$	
13	2H	INC2	1	1	$L_{13} \to 1 : \ell_{13} L_{14}$	$L_{13} \to \mathtt{n} L_{14}$
14	BOUND	ENTX	-*,2	1	$L_{14} \to 1 : \ell_{14} L_{15}$	$L_{14} \to p_{15} : \mathtt{b} L_{16} \mid 1 - p_{15} : \mathtt{b} L_6$
15		JXN	3B	1	$L_{15} \to p_{15} : \ell_{15} L_{16} \mid 1 - p_{15} : \ell_{15} L_6$	
16	4H	J1P	1B	1	$L_{16} \to p_{16} : \ell_{16} L_{17} \mid 1 - p_{16} : \ell_{16} L_2$	$L_{16} \to p_{16} : \mathtt{e} \mid 1 - p_{16} : \mathtt{e} L_2$
17				0	$L_{17} \to 1 : \epsilon$	

Fig. 10. Bubble sort, Program B from page 107 in [19].

$$L_6 \to p_8 : \mathtt{csmn} L_{14} \mid (1 - p_8) : \mathtt{cn} L_{14}$$
$$L_{14} \to p_{15} : \mathtt{b} L_{16} \mid (1 - p_{15}) : \mathtt{b} L_6$$
$$L_{16} \to p_{16} : \mathtt{e} \mid (1 - p_{16}) : \mathtt{ei} L_{14}.$$

Next we must decide were we should introduce the z's to account for the size of the input. A difficulty with bubble sort arises from the fact that it can commit more than one key to its final position in one pass, and therefore would require different numbers of z's, which is not possible in the grammar above. To solve this problem we further reduce the grammar to

$$L_1 \to 1 : \mathtt{1i} L_{14}$$
$$L_{14} \to p_{15} \cdot p_{16} : \mathtt{be} \mid p_{15} \cdot (1 - p_{16}) : \mathtt{bei} L_{14} \mid (1 - p_{15}) \cdot p_8 : \mathtt{bcsmn} L_{14} \mid (1 - p_{15}) \cdot (1 - p_8) : \mathtt{bcn} L_{14}.$$

The first rule corresponds to the initialization that the algorithm does once. The remaining rules describe what may happen in an iteration of the main loop: the algorithm terminates, start a new pass, compares and swaps the keys or compares and does not swap the keys.

The observation that leads to a solution is that the number of compares after the last swap tells us how many keys are already at their final position. Thus we modify the grammar to ensure that only compares after the last swap are marked with a z:

$$L_1 \to \mathtt{1i} L_{14} \mid \mathtt{1}i L'_{14}$$
$$L'_{14} \to \mathtt{be}z \mid \mathtt{bei}z L'_{14} \mid \mathtt{bei}z L_{14} \mid \mathtt{bcn}z L'_{14}$$
$$L_{14} \to \mathtt{bcn} L_{14} \mid \mathtt{bcsmn} L_{14} \mid \mathtt{bcsmn} L'_{14}.$$

With the grammar above we can "loop" in L_{14} as long as we need to compare and swap the keys without introducing z's. The different right-hand sides of L'_{14} do not contain any swaps but allow us to start a new pass or to terminate the algorithm. Thus we must eventually switch from L_{14} to L'_{14}, but this is only possible after the last swap in the current pass occurred. Therefore only the compares after the last swap introduce a z.

As there are inputs that do not need any swaps, an already sorted sequence, we have to ensure that we can avoid L_{14} entirely, as it introduces at least one z. Hence the second right-hand side of L_1 allows us to start directly in L'_{14}.

The last pass of bubble sort may not need any more swaps; it just verifies that all remaining keys are at their correct position. To be able to start a new pass without introducing any swaps we add the right-hand side $\mathtt{bei}z L'_{14}$ to the source L'_{14}.

Now the probabilities of the different rules are no longer connected to the conditional jumps as they appear in Program B. But this does not make a difference as the probabilities can still be trained and used for the analysis as intended. Again we translate the grammar into a system of equations:

$$L_1 \to q_1 : \mathtt{1i} L_{14} \mid q_2 : \mathtt{1}i L'_{14}$$
$$L'_{14} \to q_3 : \mathtt{be}z \mid q_4 : \mathtt{bei}z L'_{14} \mid q_5 : \mathtt{bei}z L_{14} \mid q_6 : \mathtt{bcn}z L'_{14}$$
$$L_{14} \to q_7 : \mathtt{bcn} L_{14} \mid q_8 : \mathtt{bcsmn} L_{14} \mid q_9 : \mathtt{bcsmn} L'_{14}$$

$$L_1 = q_1 \cdot \mathtt{1i} L_{14} + q_2 \cdot \mathtt{1i} L'_{14}$$
$$L'_{14} = q_3 \cdot \mathtt{be}z + q_4 \cdot \mathtt{bei}z L'_{14} + q_5 \cdot \mathtt{bei}z L_{14} + q_6 \cdot \mathtt{bcn}z L'_{14}$$
$$L_{14} = q_7 \cdot \mathtt{bcn} L_{14} + q_8 \cdot \mathtt{bcsmn} L_{14} + q_9 \cdot \mathtt{bcsmn} L'_{14}.$$

Now we replace the remaining lower-case letters and introduce the parameter we are interested in, again the running time. According to the cost of the MIXAL instructions and the abbreviations we use in the grammar, we let $\mathtt{1} \to y^1$, $\mathtt{i} \to y^5$,

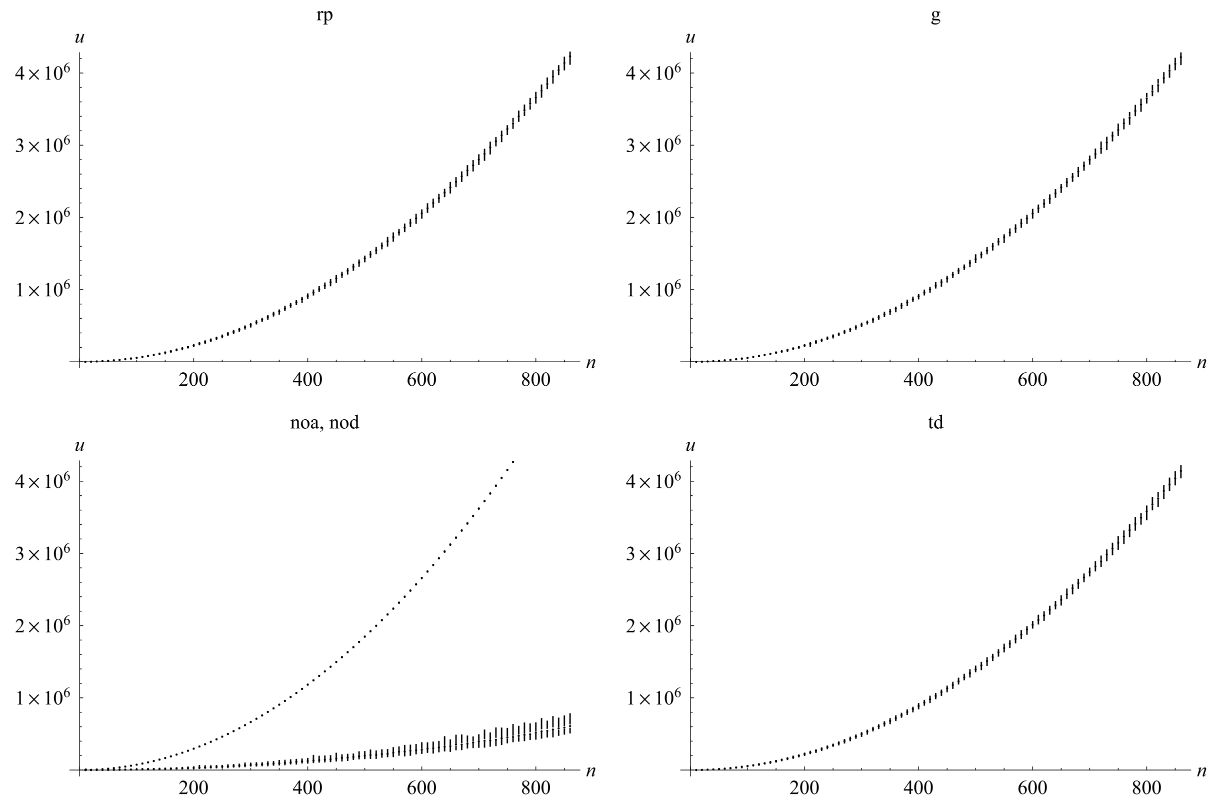

Fig. 11. The average running time of bubble sort observed while training the grammar.

b $\to y^2$, c $\to y^5$, s $\to y^6$, m $\to y^1$, n $\to y^1$, e $\to y^1$. Formally, we would use the homomorphism $h(\omega_j) = y^{\lambda(\omega_j)} \cdot z^{\gamma(\omega_j)} \cdot \alpha_j$ as introduced in Section 3.2. Now, the system of equations becomes

$$\begin{aligned}
L_1 &= q_1 \cdot y^6 \cdot L_{14} + q_2 \cdot y^6 \cdot L'_{14}\\
L'_{14} &= q_3 \cdot y^3 \cdot z + q_4 \cdot y^8 \cdot z \cdot L'_{14} + q_5 \cdot y^8 \cdot z \cdot L_{14} + q_6 \cdot y^8 \cdot z \cdot L'_{14} \qquad (5)\\
L_{14} &= q_7 \cdot y^8 \cdot L_{14} + q_8 \cdot y^{15} \cdot L_{14} + q_9 \cdot y^{15} \cdot L'_{14}.
\end{aligned}$$

Solving (5) yields the bivariate generating function:

$$B(z,y) = \frac{q_3 y^9\Big(q_2\big(1 - q_7 y^8 - q_8 y^{15}\big) + q_1 q_9 y^{15}\Big)z}{1 - q_7 y^8 - q_8 y^{15} - y^8\Big(q_5 q_9 y^{15} + (q_4 + q_6)\cdot\big(1 - q_7 y^8 - q_8 y^{15}\big)\Big)z}.$$

Building the quotient of the expected cost of all inputs of size n and the probability for an input of size n, and using $q_2 = 1 - q_1$, $q_4 = 1 - q_3 - q_5 - q_6$, $q_7 = 1 - q_8 - q_9$, we get

$$B(n) := \frac{[z^n]B_y(z,1)}{[z^n]B(z,1)} = 1 + 8n + \frac{\big(q_1(1-q_3) - (n-1)q_5\big)\big(8 + 7q_8 + 8q_9\big)}{(1-q_3)q_9}. \qquad (6)$$

Remarkably, the term without the probabilities $1 + 8n$ is again the best-case running time of bubble sort, which occurs for a sorted sequence. As before, the denominator $[z^n]B(z,1)$ is $q_3 \cdot (1-q_3)^{n-1}$, which is just the probability to leave the outer loop when n items have been sorted.

The probabilities are now obtained by training the grammar on the five sets of inputs. Fig. 11 shows the average running time measured while training the grammar. Following [30] the lines span the range from the minimum to the maximum observed. The gap in each line around the median, which should indicate the interquartile range, which represents the central 50% of the data, is too small to be seen. As expected, bubble sort is susceptible to different distributions too. So, again, nearly ordered sequences are processed a lot faster when they are almost in ascending order and respectively a lot slower when they are in descending order (lower left plot). A lot of duplicate keys have only a small impact compared to random permutations (plots on the right).

As no set of inputs contains the already sorted sequence, we can set $q_1 = 1$ and $q_2 = 0$. The dependence of the probabilities q_3, q_4 and q_9 on the number of passes is stronger than the influence of the different distributions. Probability q_3

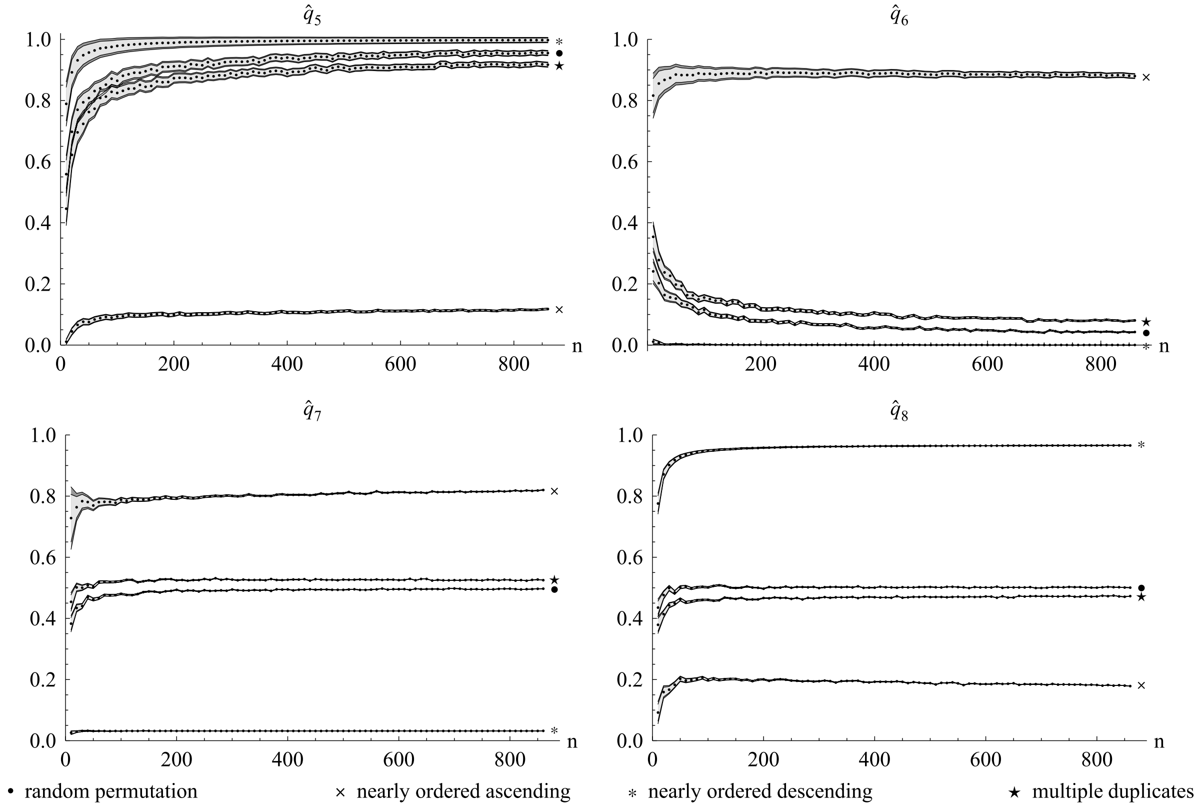

Fig. 12. 99% and 95% confidence bands of the maximum likelihood estimators $\hat{q}_5$, $\hat{q}_6$, $\hat{q}_7$ and $\hat{q}_8$.

is the probability for the termination, as this may only occur once its probability is roughly proportional to one over the number of passes. This is true for probability q_4 as well, as q_4 is the probability that a new pass is started that does not incur any swaps. This is only possible in the last pass that verifies that every key is in its correct position and that no more swaps are necessary. Probability q_9 is the probability for the transition to the part of the pass where no more swaps occur; this can only happen once per pass and thus is connected to the number of passes. All three probabilities exhibit a $1/n$ behavior.

The 95% and 99% confidence bands for selected training sets of the estimators $\hat{q}_5$, $\hat{q}_6$, $\hat{q}_7$ and $\hat{q}_8$ are shown in Fig. 12. The confidence bands for the Gauss distributed inputs are not shown, as they obscure the other bands. The different behavior for the various sets of inputs is clearly reflected by the confidence bands. Nearly ordered sequences (symbol $\times$) require a lot of compares without swaps; thus $\hat{q}_7$ and $\hat{q}_6$ are near probability one and $\hat{q}_8$ is near probability zero, and vice versa for sequences which are almost in reverse order (symbol *). The probability to start a new pass, $\hat{q}_5$, is low for almost sorted inputs (symbol $\times$). For inputs that are almost in reverse order (symbol *), $\hat{q}_5$ is high.

The placeholders of remaining probabilities q_3, q_5, q_8, q_9 in (6) are now replaced by the functions

$$\begin{aligned}
\hat{q}_{d,3}(n) &:= c_{d,3}(n)/\big(c_{d,3}(n)+c_{d,4}(n)+c_{d,5}(n)+c_{d,6}(n)\big),\\
\hat{q}_{d,5}(n) &:= c_{d,5}(n)/\big(c_{d,3}(n)+c_{d,4}(n)+c_{d,5}(n)+c_{d,6}(n)\big),\\
\hat{q}_{d,8}(n) &:= c_{d,8}(n)/\big(c_{d,7}(n)+c_{d,8}(n)+c_{d,9}(n)\big),\\
\hat{q}_{d,9}(n) &:= c_{d,9}(n)/\big(c_{d,7}(n)+c_{d,8}(n)+c_{d,9}(n)\big)
\end{aligned}$$

where the functions $c_{d,i}(n)$ are obtained by a least squares fitting of the counts $c_{d,i}^{(n)}$ with the model functions that consist of a linear combination of basic functions like powers of n, logarithms and products thereof as shown below:

$$\begin{array}{lll}
c_{d,3}(n) \sim a_3 & c_{d,6}(n) \sim a_6 \cdot n + b_6 \cdot \ln(n) + c_6 & c_{d,9}(n) \sim a_9 \cdot n + b_9 \cdot \ln(n)\\
c_{d,4}(n) \sim a_4 & c_{d,7}(n) \sim a_7 \cdot n^2 + b_7 \cdot n \cdot \ln(n) + c_7 \cdot n & \\
c_{d,5}(n) \sim a_5 \cdot n + b_5 & c_{d,8}(n) \sim a_8 \cdot n^2 + b_8 \cdot n \cdot \ln(n) + c_8 \cdot n. &
\end{array}$$

We obtain the following asymptotic expressions for the average running time of bubble sort:

$$\begin{array}{ll}
B_{\text{rp}}(n) = 5.77 \cdot n^2 + \mathcal{O}(n\ln(n)) & B_{\text{gd}}(n) = 5.75 \cdot n^2 + \mathcal{O}(n\ln(n))\\
B_{\text{noa}}(n) = 0.90 \cdot n^2 + \mathcal{O}(n\ln(n)) & B_{\text{nod}}(n) = 7.38 \cdot n^2 + \mathcal{O}(n\ln(n))\\
B_{\text{md}}(n) = 5.69 \cdot n^2 + \mathcal{O}(n\ln(n)). &
\end{array}$$

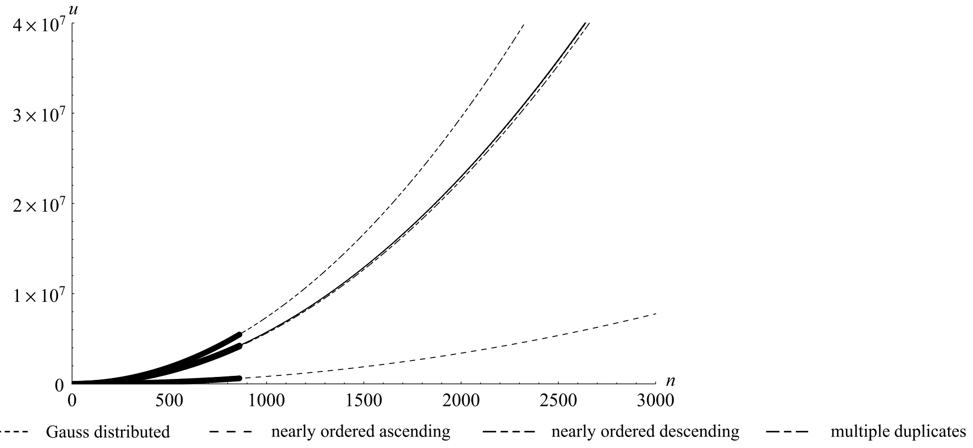

Fig. 13. The average running time of straight insertion sort obtained per maximum likelihood analysis.

In [19], the following asymptotic result is given for the average running time of bubble sort when processing uniformly distributed inputs:

$$B_{\text{uni}}(n) = 5.75 \cdot n^2 - 4(n+1)\ln(n+1) + 5.17 \cdot n + \mathcal{O}\big(\sqrt{n}\big).$$

The final results of the maximum likelihood analysis as stated above are plotted in Fig. 13. The thick part of the curves between 0 and 860 indicates the range for which training data was present. Similar to straight insertion sort, bubble sort is susceptible to the distribution of the input data. The leading coefficient for the average running time is, according to the analysis in [19], 5.75 and at most 7.5 in the worst case, which is almost reached for the sequences that are nearly in reverse order.

Modifying the grammar to allow a different number of z's for different passes was the key to making the maximum likelihood analysis work. The probabilities in the SCFG do not need to have a conditional jump as counterpart: tuning them via the grammar training to produce "flow of control strings" with the correct frequencies is sufficient. In the following analysis of heapsort no modification of the grammar is necessary.

3.2.4. *Heapsort*

Our last example is heapsort; its implementation in MIXAL is shown in Fig. 14 together with its translation into a SCFG.

As before, we abbreviate blocks of instructions which belong together, remove the chain rules and thus get a pair of rules for every conditional jump. Introducing the symbol z to track the size of the input is not complicated, because after the heap construction is finished, one key after another is placed at its final position. The code that performs this task is "marked" with a z – here lines 27 and 30 – thus two rules get marked with a z. Again we replace the lower-case letters in such a way that we track the running time in the exponent of the symbol y. The grammar and the system of equations then take the following form:

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

Back substitution yields the solution:

$$H(z,y) = \frac{p_{24}p_{29}y^{27}z^2 \cdot h(y)}{1 - p_{16}p_{21}\bar{p}_{22}y^{12} - p_{16}y^{14} \cdot g(y) - \big(p_{24}\bar{p}_{29}zy^5 + \bar{p}_{24}\big)y^{15} \cdot h(y)} \tag{7}$$

with

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

Determining the quotient of the expected cost of all inputs of size n and the probability for an input of size n, we get

$$H(n) := \frac{[z^n]H_y(z,1)}{[z^n]H(z,1)} = 2 + 5n + \frac{(n-1)\big(22 + 4p_{11}(1-p_{21}) - p_{21}\big(2 + 5p_{22}\big) - 8p_{16}(1 - p_{21}p_{22})\big)}{(1 - p_{16}\big(1 - p_{21}p_{22})\big)p_{24}}. \tag{8}$$

Line no.	LOC	OP	ADDRESS	Cost	Rule	
01	START	ENT1	N/2	1	$L_1 \to 1 : \ell_1 L_2$	$L_1 \to \mathtt{i}L_3$
02		ENT2	N-1	1	$L_2 \to 1 : \ell_2 L_3$	
03	1H	DEC1	1	1	$L_3 \to 1 : \ell_3 L_4$	$L_3 \to \mathtt{d}L_5$
04		LDA	INPUT+1,1	2	$L_4 \to 1 : \ell_4 L_5$	
05	3H	ENT4	1,1	1	$L_5 \to 1 : \ell_5 L_6$	$L_5 \to \mathtt{p}L_{18}$
06		ENT5	0,2	1	$L_6 \to 1 : \ell_6 L_7$	
07		DEC5	0,1	1	$L_7 \to 1 : \ell_7 L_8$	
08		JMP	4F	1	$L_8 \to 1 : \ell_8 L_{18}$	
09	5H	LDX	INPUT,4	2	$L_9 \to 1 : \ell_9 L_{10}$	$L_9 \to \mathtt{b}L_{11}$
10		CMPX	INPUT+1,4	2	$L_{10} \to 1 : \ell_{10} L_{11}$	
11		JGE	6F	1	$L_{11} \to p_{11} : \ell_{11} L_{12} \mid 1 - p_{11} : \ell_{11} L_{15}$	$\ell_{11} = \mathtt{j}$
12		INC4	1	1	$L_{12} \to 1 : \ell_{12} L_{13}$	$L_{12} \to \mathtt{o}L_{14}$
13		DEC5	1	1	$L_{13} \to 1 : \ell_{13} L_{14}$	
14	9H	LDX	INPUT,4	2	$L_{14} \to 1 : \ell_{14} L_{15}$	$L_{14} \to \mathtt{l}L_{15}$
15	6H	CMPA	INPUT,4	2	$L_{15} \to 1 : \ell_{15} L_{16}$	$L_{15} \to \mathtt{r}L_{16}$
16		JGE	8F	1	$L_{16} \to p_{16} : \ell_{16} L_{17} \mid 1 - p_{16} : \ell_{16} L_{23}$	$\ell_{16} = \mathtt{j}$
17	7H	STX	INPUT,3	2	$L_{17} \to 1 : \ell_{17} L_{18}$	$L_{17} \to \mathtt{s}L_{18}$
18	4H	ENT3	0,4	1	$L_{18} \to 1 : \ell_{18} L_{19}$	$L_{18} \to \mathtt{t}L_{21}$
19		DEC5	0,4	1	$L_{19} \to 1 : \ell_{19} L_{20}$	
20		INC4	0,4	1	$L_{20} \to 1 : \ell_{20} L_{21}$	
21		J5P	5B	1	$L_{21} \to p_{21} : \ell_{21} L_{22} \mid 1 - p_{21} : \ell_{21} L_9$	$\ell_{21} = \mathtt{j}$
22		J5Z	9B	1	$L_{22} \to p_{22} : \ell_{22} L_{23} \mid 1 - p_{22} : \ell_{22} L_{14}$	$\ell_{22} = \mathtt{j}$
23	8H	STA	INPUT,3	2	$L_{23} \to 1 : \ell_{23} L_{24}$	$L_{23} \to \mathtt{w}L_{24}$
24	2H	J1P	1B	1	$L_{24} \to p_{24} : \ell_{24} L_{25} \mid 1 - p_{24} : \ell_{24} L_3$	$\ell_{24} = \mathtt{j}$
25		LDA	INPUT+1,2	2	$L_{25} \to 1 : \ell_{25} L_{26}$	$L_{25} \to \mathtt{g}L_{29}$
26		LDX	INPUT+1	2	$L_{26} \to 1 : \ell_{26} L_{27}$	
27		STX	INPUT+1,2	2	$L_{27} \to 1 : \ell_{27} L_{28}$	
28		DEC2	1	1	$L_{28} \to 1 : \ell_{28} L_{29}$	
29		J2P	3B	1	$L_{29} \to p_{29} : \ell_{29} L_{30} \mid 1 - p_{29} : \ell_{29} L_5$	$\ell_{29} = \mathtt{j}$
30		STA	INPUT+1	2	$L_{30} \to 1 : \ell_{30} L_{31}$	$L_{30} \to \mathtt{e}$
31				0	$L_{31} \to 1 : \epsilon$	

Fig. 14. Heapsort, Program H from page 146f in [19].

The estimates for the probabilities are again obtained by training the grammar on the different sets of inputs. Fig. 15 shows the average running times measured while training the grammar; again, the lines span the range from the minimum to the maximum observed, while the gap in each line around the median indicates the interquartile range, which represents the central 50% of the data. As already noted in [19], we observe in Fig. 15 that heapsort is quite insusceptible to differently distributed inputs, when comparing the probabilities for the different input distributions. That is why we abstain from showing the plots of five similar relative frequencies calculated from the sorting of five types of input with heapsort.

The placeholders of the probabilities $p_{11}, p_{16}, p_{21}, p_{22}, p_{24}$ are now replaced by the functions

$$\begin{aligned}
&\hat{p}_{d,11}(n) := c_{d,11}(n)/\big(c_{d,11}(n) + \bar{c}_{d,11}(n)\big), && \hat{p}_{d,16}(n) := c_{d,16}(n)/\big(c_{d,16}(n) + \bar{c}_{d,16}(n)\big),\\
&\hat{p}_{d,21}(n) := c_{d,21}(n)/\big(c_{d,21}(n) + \bar{c}_{d,21}(n)\big), && \hat{p}_{d,22}(n) := c_{d,22}(n)/\big(c_{d,22}(n) + \bar{c}_{d,22}(n)\big),\\
&\hat{p}_{d,24}(n) := c_{d,24}(n)/\big(c_{d,24}(n) + \bar{c}_{d,24}(n)\big)
\end{aligned}$$

where then functions $c_{d,i}(n)$ and $\bar{c}_{d,i}(n)$ are the results of a least squares fitting of the counts $c_{d,i}^{(n)}$ and the counts of the complementary event $\bar{c}_{d,i}^{(n)}$ to the model functions as shown below:

$$\begin{aligned}
&c_{d,11}(n) \sim a_{11} \cdot n \cdot \ln(n) + b_{11} \cdot n + c_{11}, && \bar{c}_{d,11}(n) \sim \bar{a}_{11} \cdot n \cdot \ln(n) + \bar{b}_{11} \cdot n + \bar{c}_{11},\\
&c_{d,16}(n) \sim a_{16} \cdot n \cdot \ln(n) + b_{16} \cdot n + c_{16}, && \bar{c}_{d,16}(n) \sim \bar{a}_{16} \cdot n + \bar{b}_{16},\\
&c_{d,21}(n) \sim a_{21} \cdot n + b_{21} \cdot \ln(n) + c_{21}, && \bar{c}_{d,21}(n) \sim \bar{a}_{21} \cdot n \cdot \ln(n) + \bar{b}_{21} \cdot n + \bar{c}_{21} \cdot \ln(n) + \bar{d}_{21},\\
&c_{d,22}(n) \sim a_{22} \cdot n + b_{22}, && \bar{c}_{d,22}(n) \sim \bar{a}_{22} \cdot \ln(n) + \bar{b}_{22},\\
&c_{d,24}(n) \sim a_{24} \cdot n + b_{24}, && \bar{c}_{d,24}(n) \sim \bar{a}_{24} \cdot n + \bar{b}_{24}.
\end{aligned}$$

The trained probability functions $p_{d,11}(n)$, $p_{d,16}(n)$, $p_{d,21}(n)$, $p_{d,22}(n)$, $p_{d,24}(n)$ are now inserted into (8), and dividing the leading coefficients yields the following asymptotic results:

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

Knuth's empirical result from page 148 in [19] is $H_{\mathrm{uni}}(n) \approx 23.08 \cdot n\ln(n)$. The results of the maximum likelihood analysis are plotted in Fig. 16. The thicker parts of the curves between 0 and 860 indicate the range for which training data was present. Fig. 16 strongly resembles Fig. 15, where the training data is shown, in that the five distributions examined in this work do not have a big influence on the running time of heapsort.

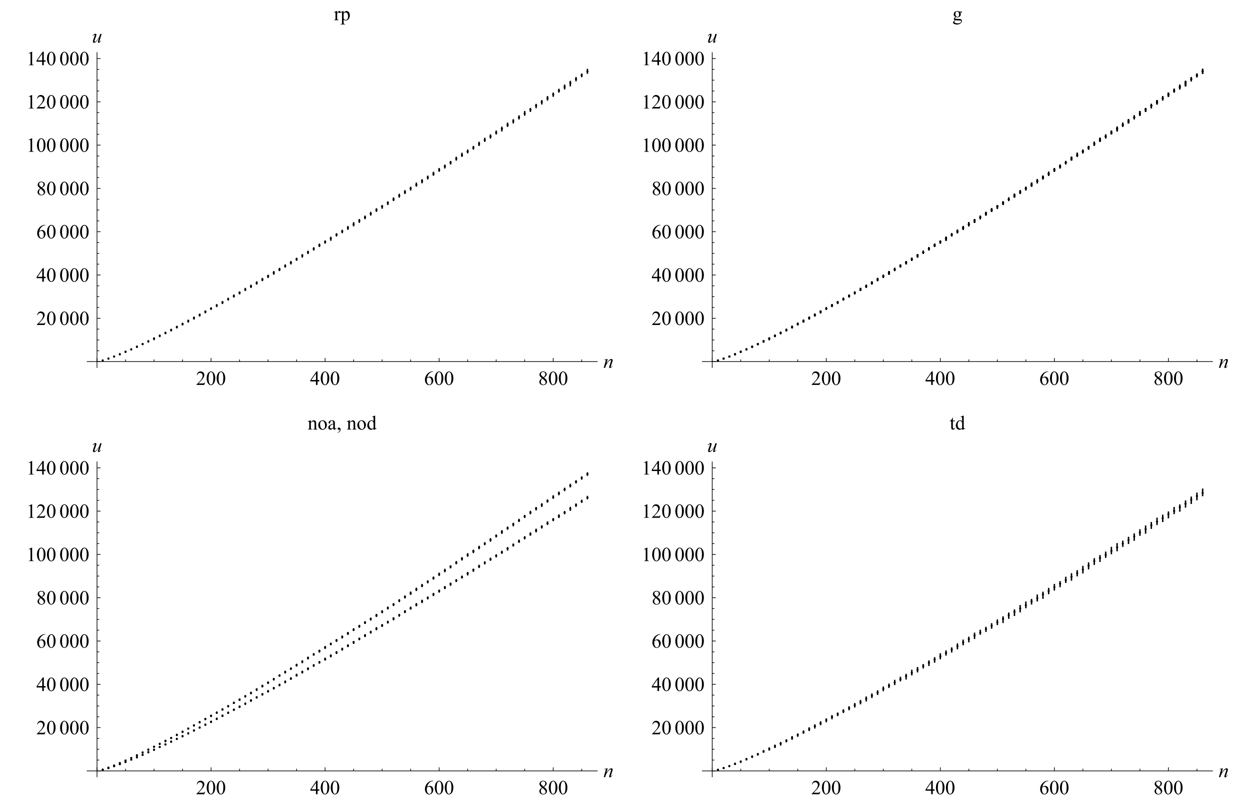

Fig. 15. The average running time of heapsort observed while training the grammar.

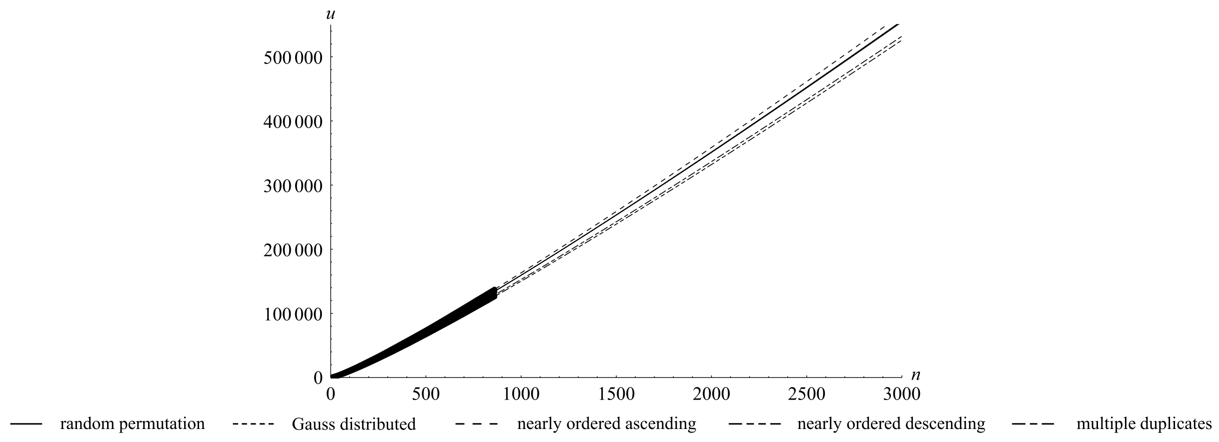

Fig. 16. The average running times of heapsort obtained per maximum likelihood analysis are shown.

It is more interesting to take a closer look at probability p_{11}. It reveals that the tree structure of the heap has a subtle influence; see Fig. 17. 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, which 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. If we allow "Fourier polynomials", which are truncated Fourier series, as basis functions in our fitting process, we can actually capture the periodic behavior, but as the change in the results is less than the variance due to the randomness, this is not worth the effort. Such periodicities are not uncommon: see Theorem 2 for example; it contains the periodic function $\alpha(\log_2(n))$.

To exemplify how simple it is to analyze other parameters besides the running time, we continue the maximum likelihood analysis of heapsort by changing the substitution of the lower-case letters to track the average number of key comparisons in the exponent of symbol y. Formally, this would require us to change the homomorphism h that is used to translate the grammars rules into the equations. Here we state just informally the new substitutions: $\mathtt{b} \to y^1$, $\mathtt{r} \to y^1$, and all other

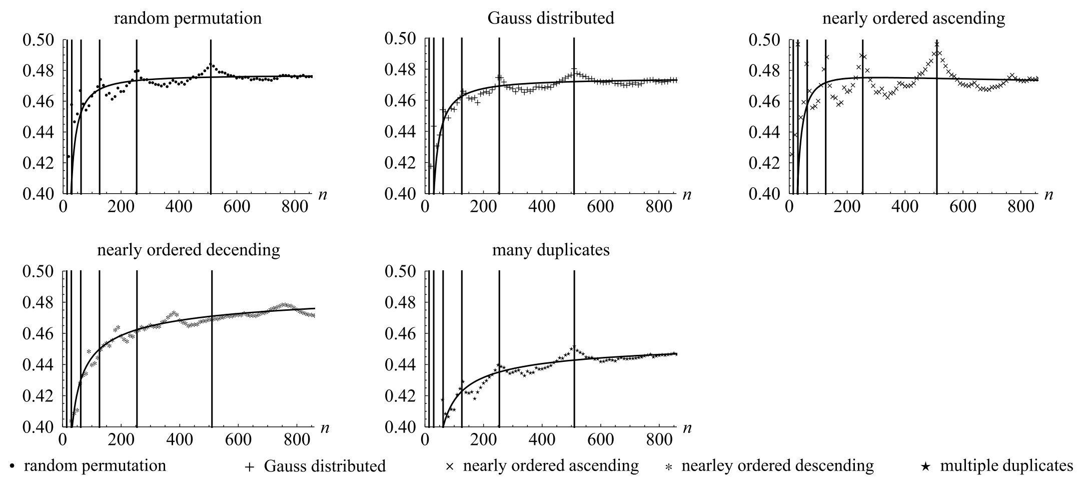

Fig. 17. Probability p_{11} exhibits a periodic oscillation; in the case of almost reversely ordered sequences (*), the oscillation is shifted.

lower-case letters are replaced by y^0; thus they are omitted. The resulting system of equations needs to be solved again, but no new training of the grammar is necessary. We simply insert the trained probabilities into the new solution, and obtain the results for the average number of comparisons for the five different distributions studied here.

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

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.

Another change to the homomorphism allows us to find the average number of interchanges. The new substitutions are $\mathtt{s} \to y^1$, $\mathtt{w} \to y^1$, $\mathtt{g} \to y^1$, and other lower-case letters are replaced by y^0. Solving the new system of equations and inserting the trained probabilities in the solution yields

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

Again, for random inputs, Knuth reports $1.443 \cdot n \cdot \ln(n) - 0.87 \cdot n - 1$ for the average number of interchanges.

4. Significance of results

While discussing the straight insertion sort example we deferred the proof that the maximum likelihood analysis method presented in this paper and Knuth's classical approach towards an average-case analysis yield the same result, when the probabilities are derived theoretically, instead of by a maximum likelihood training.

The maximum likelihood analysis method requires probabilities for the rules of the SCFG used. When analyzing algorithms, the probabilities of the rules are simply the probabilities for the various conditional jumps being taken or not taken. As already discussed, we can obtain the probabilities from a maximum likelihood training of the grammar, but we can also derive the probabilities from the analysis in Knuth's books in the following way.

As Knuth explains on page 145f in [18], the timing column of each MIXAL program in his books represents the profile, that is, the number of times the instruction on that line will be executed during the course of the program. As the instructions following a conditional jump may be the target of other jumps we must use Kirchhoff's law to deduce the number of times a jump is taken or not taken. Dividing this number by the total number of times the jump is executed, we obtain the probabilities for a jump being taken or not taken.

The probabilities derived from the profile of the program depend on the size of the input, as the characteristics like the number of left-to-right minima, etc. also depend on the size of the input.

Theorem 5. *The average-case running time of an arbitrary algorithm is the same when analyzed either with the maximum likelihood method or Knuth's classical approach, when the probabilities for the SCFG are derived from the profile instead of by a maximum likelihood training.*

Proof. The contribution to the running time of a sequence of instructions between two jumps is obviously the same in both methods, as the number of times the instructions are executed is likewise weighted with the cost of the instructions. Thus different running times would only be possible if the average number of times that a conditional jump is taken differs for

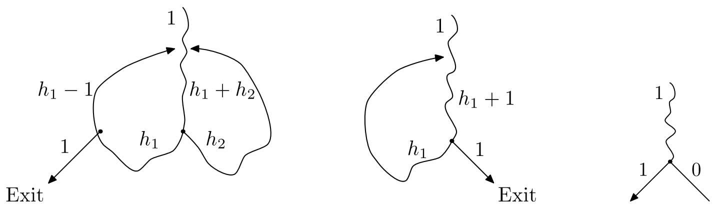

Fig. 18. The three cases in the proof of Theorem 5.

both methods. Therefore we examine an arbitrary conditional jump and distinguish three cases, shown in Fig. 18. The arrows symbolize the possible paths the flow of control may take and the labels indicate the number of times that particular branch has been taken.

1. Case one, on the left. We reach the conditional jump from the start of the program and both branches loop in such a way that we can reach the conditional jump again. One of the two branches must have an exit, otherwise the program will not terminate. Whether the branches themselves contain further loops and jumps does not matter as we can think of them as fully unrolled. Let h_1 and h_2 be the number of times the two branches are taken in Knuth's model; according to the maximum likelihood analysis we assign the probabilities $p_1 = h_1/(h_1 + h_2)$ and $p_2 = h_2/(h_1 + h_2)$ to the corresponding rules in the SCFG. Clearly the total number of times the conditional jump is executed is $h_1 + h_2$, and we interpret the loops as a repeated random experiment (recurrent non-terminal). Then the expected numbers of times the branches are taken are $E[\#_1] = p_1 \cdot (h_1 + h_2) = h_1$ and $E[\#_2] = p_2 \cdot (h_1 + h_2) = h_2$, for $\#_i$ the number of times branch i is taken, which is the same as in Knuth's analysis.
2. Case two, in the middle. If only one of the two branches loops, the other must have a frequency of 1, otherwise it would loop too. Let h_1 be the number of times the looping branch is taken in Knuth's model; then we assign the probabilities $p_1 = h_1/(h_1 + 1)$ and $p_2 = 1/(h_1 + 1)$ to the corresponding rules in the SCFG. Calculating the expected number of times the looping branch is taken yields $E[\#_1] = \sum_{i\geq 1} i \cdot (h_1/(h_1 + 1))^{i-1} \cdot 1/(h_1 + 1) - 1 = h_1$. We have to subtract one as the sum is the expected number of repetitions of the random experiment including the first jump that leaves the loop. Again, both methods determine the same contribution to the running time.
3. Case three, on the right. If both branches do not loop, then the jump is executed just once. Which branch is taken depends on the input, and as we assign probabilities 1 and 0 to the rules the contribution to the running time is the same in both models.

As the contribution to the running time is the same for a maximum likelihood analysis and Knuth's classical approach in all three cases, we can conclude that the same expected running time is calculated for both analysis methods. □

This is an important result, as it ensures that we really have computed the expected running times of the algorithms.

5. Conclusions

We have presented an approach for the description of random structures and flows of control via stochastic context-free grammars and their languages and applied it successfully to the analysis of binary tries and several sorting algorithms. Deriving asymptotic results for the average performance is possible for a large variety of parameters.

As algorithms are typically transformed into a number of assembly language instructions that are executed sequentially, finding a suitable encoding in the form of a formal language for algorithms is easily possible by the three rules given in Section 3.2, which turn an implementation of an algorithm into a regular grammar that encodes its flow of control. This grammar is, however, not suited to deriving accurate results for the variance of the parameter of interest. This is due to the fact that the regular language used to encode the flow of control is too general. It allows infinitely many words, which do not correspond to an actual flow of control for some input. Switching to a non-regular language might improve the results on the variance, but may still not be enough. This is hinted at by a result obtained from formal language theory; see Lemma 8.8 in [15]. This implies that the set of valid computations of a Turing machine is in general not a context-free language. As the valid computations are comparable to our flow-of-control strings, this suggest that context-free languages may still be "too weak" to describe the flow of control in the necessary detail for an accurate result for the variance. This led us to the extension of context-free grammars to stochastic context-free schemes as done for modeling tries. It seems promising as it allows to capture more complicated behavior. However, the mathematics behind this is considerably more difficult and deserves further investigation.

When based on context-free grammars, all the 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 data structure or algorithm is required. Thus our approach can reduce the effort required for an average-case analysis, allowing for the consideration of realistic input distributions with unknown distribution functions at the same time.

For data structures we do not have the same situation as for algorithms, which are finally translated into a list of instructions for a processor. Data structures do not exhibit such a "normal form" that would allow a general way of producing a suitable encoding. Here, imagination is needed to find an encoding.

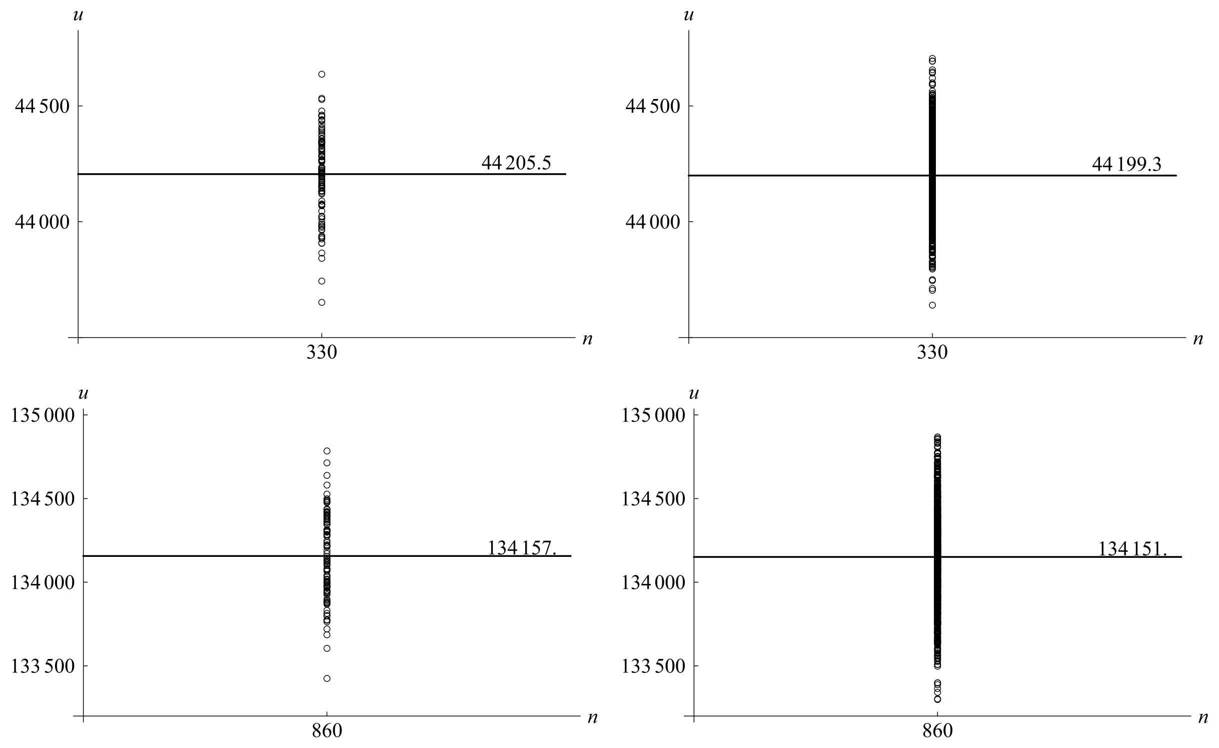

Fig. 19. The running times of 100 and 1000 runs of heapsort on random inputs of size 330 and 860 are shown together with their average.

The confidence intervals discussed in Section 3.2.1 were helpful in establishing the sample size. We empirically checked that a tenfold increase in the sample size had only an insignificant effect on the results. Fig. 19 shows the individual running times for each of the 100 respective 1000 inputs. Their average is indicated by a line. The changes in the averages were less than one per cent in the case we have checked. As the time for training the grammars increases linearly with the sample size, we decided not to increase the sample size any further. As this may be different for other algorithms and data structures, the influence of the sample size should be studied thoroughly.

A maximum likelihood analysis has several advantages over plain experimentation. After a grammar has been trained on a set of inputs, the resulting probabilities provide all the necessary information for the analysis of the algorithm or data structure. Hence we are able to conserve the results of the experiments by storing just a small number of probabilities.

With the probabilities at hand we can study any number of parameters that the chosen encoding provides. There is no need to conduct multiple, possibly expensive, experiments for additional parameters.

Moreover, if we later aim for studying a parameter we did not have in mind when setting up the experiments, we just have to use a different homomorphism for introducing the parameter. This is impossible for plain experiments when no statistics have been collected on unconsidered parameters beforehand. Then only further experiments can provide the statistics that would allow other parameters to be analyzed in that case.

Furthermore, the maximum likelihood analysis provides an expression for the average of a parameter that can be used to study the influence of the different probabilities. This is helpful for setting up the experiments, as it shows which probabilities contribute more to the total average and thus have to be measured more carefully. The expression also allows us to look at the error propagation.

The maximum likelihood analysis may produce proofs for the absence of a dependency of the parameter from the distribution of the inputs. If only parts are probability-free then a lower bound may be obtained. In the case of bubble sort the best case was rediscovered; see Eq. (6).

We like to mention that results similar to the ones found with a maximum likelihood analysis can be obtained with plain experimentation, of course without the aforementioned advantages. Another advantage of the maximum likelihood analysis over plain experimentation is that it provides a clue for a traditional analysis what the solution may look like. After arriving at equations like (3), (6) and (8), the solution for straight insertion sort, bubble sort and heapsort, a good deal has been learned about the structure of the solutions, which can be very useful if a traditional analysis is also planned.

We do not expect the maximum likelihood analysis to provide good results when an arbitrary set of basis functions is used. Whoever conducts a maximum likelihood analysis should carefully inspect the algorithm to be analyzed and choose the set of basis functions accordingly. As the fitting itself is not computationally intensive, several sets of basis functions might be used to find one that suits best.

The generation of random structures that follow a pre-trained distribution is noteworthy in its own right regarding simulation and testing. If one has to generate test data that is not totally random – perhaps a certain type of input exhibits a problematic behavior of an algorithm – one can collect some samples and train a SCFG or SCFS in order to generate more test inputs according to the trained distribution. Our results related to tries proved the potential of this idea.

Further interesting questions are: Which probability distributions can be modeled by a SCFG/SCFS? Currently it is not clear whether every distribution that we may encounter in different training sets can be captured by the SCFG/SCFS. 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?

Acknowledgements

The first author's research was supported by DFG grant NE 1379/2-1. The authors would like to thank Tim Schumann for experimenting with SCFSs and their training. We are especially grateful to the anonymous referees for many useful suggestions which helped to improve the presentation of the results of this paper considerably.

References

[1] John Aldrich, R.A. Fisher and the making of maximum likelihood 1912–1922, Statistical Science 12 (3) (1997) 162–176.
[2] T.L. Booth, R.A. Thompson, Applying probability measures to abstract languages, IEEE Transactions on Computers C-22 (5) (1973) 442–450.
[3] T. Chi, S. Geman, Estimation of probabilistic context-free grammars, Computational Linguistics 24 (2) (1998) 299–305.
[4] R. Chaudhuri, S. Pham, O.N. Garcia, Solution to an open problem on probabilistic grammars, IEEE Transactions on Computers C-32 (8) (1983) 748–750.
[5] N. Chomsky, M.-P. Schützenberger, The algebraic theory of context-free languages, in: P. Braffort, D. Hirschberg (Eds.), Computer Programming and Formal Languages, North Holland, 1963, pp. 118–161.
[6] A. Corazza, G. Satta, Cross-entropy and estimation of probabilistic context-free grammars, in: Proc. of HLT-NAACL, New York, USA, 2006, pp. 335–342.
[7] A. Corazza, G. Satta, Probabilistic context-free grammars estimated from infinite distributions, IEEE Transactions on Pattern Analysis and Machine Intelligence 29 (8) (2007) 1379–1393.
[8] R. Donaghey, L.W. Shapiro, Motzkin numbers, Journal of Combinatorial Theory Series A 23 (3) (1977) 291–301.
[9] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, Biological sequence analysis, in: Probabilistic Models of Proteins and Nucleic Acids, Cambridge University Press, 1998.
[10] P. Flajolet, A.M. Odlyzko, Singularity analysis of generating functions, SIAM Journal on Algebraic and Discrete Methods 3 (2) (1990) 216–240.
[11] P. Flajolet, B. Salvy, P. Zimmermann, Automatic average-case analysis of algorithms, Theoretical Computer Science 79 (1) (1991) 37–109.
[12] P. Flajolet, R. Sedgewick, Analytic Combinatorics, Cambridge University Press, ISBN: 9-780-521-89806-5, 2009.
[13] Michael A. Harrison, Introduction to Formal Language Theory, Addison Wesley, ISBN: 0-201-02955-3, 1978.
[14] Einar Hille, Analytic Function Theory, Vol. II, Chelsea Publishing Company, New York, ISBN: 0-8284-0270-1, 1973.
[15] John E. Hopcroft, Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison Wesley, ISBN: 0-201-02988-X, 1979.
[16] T. Huang, K.S. Fu, On stochastic context-free languages, Information Sciences (3) (1971) 201–224.
[17] P. Kirschenhofer, H. Prodinger, On some application of formulae of Ramanujan in the analysis of algorithms, Mathematika 38 (1991) 14–33.
[18] Donald E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd ed, Addison Wesley, 1997.
[19] Donald E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd ed, Addison Wesley, 1998.
[20] Steven G. Krantz, Harold R. Parks, The Implicit Function Theorem, History, Theory, and Applications, Birkhäuser, ISBN: 0-8176-4285-4, 2002.
[21] W. Kuich, Semirings and Formal Power Series: Their Relevance to Formal Languages and Automata, in: G Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, Vol. 1: Word, Language, Grammar, ISBN: 3-540-60420-0, 1997, pp. 609–677. (Chapter 9).
[22] D. Christopher, Manning and Hinrich Schütze: Foundations of Statistical Natural Language Processing, MIT Press, 1999, ISBN-10: 0-262-13360-1, ISBN-13: 978-0-262-13360-9.
[23] M.E. Nebel, The stack-size of tries: A combinatorial study, Theoretical Computer Science 270 (2002) 441–461.
[24] M.-J. Nederhof, G. Satta, Probabilistic parsing as intersection, in: Proc. of the 8th IWPT, Nancy, France, 2003, pp. 137–148.
[25] M.-J. Nederhof, G. Satta, Estimation of consistent probabilistic context-free grammars, in: Proc. of the HLT-NAACL, New York, USA, 2006, pp. 343–350.
[26] G. Park, H.-K. Hwang, P. Nicodème, W. Szpankowski, Profiles of tries, LATIN 2008: 1-11.
[27] M. Régnier, P. Jacquet, New results on the size of tries, IEEE Transactions on Information Theory 35 (1) (1989) 203–205.
[28] J.-A. Sánchez, J.-M. Benedí, Consistency of stochastic context-free grammars from probabilistic estimation based on growth transformation, IEEE Transactions on Pattern Analysis and Machine Intelligence 19 (9) (1997) 1052–1055.
[29] Martin A. Tanner, Tools for Statistical Inference, 3rd ed., Springer, ISBN: 978-0-387-94688-7, 1997.
[30] Edward R. Tufte, The Visual Display of Quantitative Information, Graphics Press, Cheshire, CT, 1990.
[31] B.L. van der Warden, Modern Algebra, Vol. II, 2nd ed., New York, 1950.
[32] C.S. Wetherell, Probabilistic languages: A review and some open questions, Computing Surveys 12 (4) (1980) 361–379.