# A GENERAL FRAMEWORK FOR OBTAINING RECURRENCE RELATIONS FOR THE AVERAGE-CASE ANALYSIS OF RECURSIVE PROCEDURES*

JOHANN BLIEBERGER AND ROLAND LIEGER

## 1. Introduction

The literature on average-case analysis of algorithms and data-structures is enormous. For example compare [9, 11, 10, 14, 20, 1, 2, 19]. Several approaches have been taken in order to automate the average-case analysis of algorithms. In the following we mention the most important recent results:

- In [21] a collection of lists manipulation programs is studied. The paper is concerned with simple programs only.
- In [18] average-case analysis is based on frequencies and not on probabilities.
- In [8] a system of great expressive power is presented, but deduction and simplification is very complicated. The system is built on integrating concepts of [12] and [16].
- In [6, 7] a system for automatic average-case analysis of algorithms is described which is based on combinatorial representation of the underlying data structures.

Note that we did not mention papers concerned with worst-case timing analysis such as [17, 15].

In this paper we are going to present a method how one can effectively derive recurrence relations from the *source code* of recursive procedures or functions, which describe the average-case time behavior of these procedures. This method is based on a fairly general source code description of the underlying recursive procedure and on additional information on the probability distribution of the parameter space of the procedure, which is supplied by the programmer.

The recurrence relation derived by our method can be used for a numerical study of the timing behavior or can be solved by the programmer or by a mathematician exactly or asymptotically. Note, however, that solving these recurrence relations is not treated in this paper.

The method of our algorithm is based on one of our recent papers (cf. [4]) where we have studied real-time recursions. We introduced a new notation that facilitats automated *worst-case* time and space analysis of real-time recursions, so-called *parameter space morphisms*. In this paper we are applying such morphisms in order to facilitate *average-case* time analysis of recursions. In the context of average-case

*Supported by the Austrian Science Foundation (FWF) under grant P10188-PHY.

*Key words and phrases.* average-case analysis, recursions, parameter space morphisms.

analysis parameter space morphisms can be thought of being a generalization of *complexity measures* or *size functions* which usually are applied when algorithms are analyzed.

If the probability distribution of the parameter space fulfills a simple condition, we can prove our major result (Theorem 5.1) which can be used for deriving recurrence relations for the timing behavior of the recursive algorithm.

## 2. Definitions and Preliminary Results

**Definition 2.1.** Essential properties of a recursive procedure $p$ are the *parameter space* $\mathcal{F}$, i.e., the set of all possible (tuples of) values of parameters of $p$, a set $\mathcal{F}_0 \subseteq \mathcal{F}$, the *terminating values* of $\mathcal{F}$, and its code. If $p$ is called with actual parameters $f_0 \in \mathcal{F}_0$, the code being executed must not contain a recursive call of $p$ to itself. If $p$ is called with actual parameters $f \in \mathcal{F} \setminus \mathcal{F}_0$, the code being executed must contain at least one recursive call of $p$ to itself.

**Definition 2.2.** We call a recursive procedure $p$ *well-defined* if for each element of $\mathcal{F}$ the procedure $p$ completes correctly, e.g. does not loop infinitely and does not terminate because of a runtime error.

From now on, when we use the term recursive procedure, we mean well-defined recursive procedure.

**Definition 2.3.** We define a set $\mathcal{R}(f) \subseteq \mathcal{F}$, $(f \in \mathcal{F} \setminus \mathcal{F}_0)$ by

$$\overline{f} \in \mathcal{R}(f)$$

iff $p(\overline{f})$ is directly called in order to compute $p(f)$. $\mathcal{R}(f)$ is called the set of *direct successors* of $f$. If $f \in \mathcal{F}_0$, the set $\mathcal{R}(f) = \emptyset$, i.e., it is empty.

## 3. Computational Model and Time Effort

The time effort $\mathcal{T}$ of a recursive procedure $p$ is a recursive function $\mathcal{T} : \mathcal{F} \to \mathbb{R}$ or $\mathcal{T} : \mathcal{F} \to \mathbb{N}$. If time is measured in integer multiples of say micro-seconds or CPU clock ticks, one can use an integer valued function $\mathcal{T}$ instead of a real valued one.

Function $\mathcal{T}$ is defined recursively depending on the source code of $p$. How the recurrence relation for $\mathcal{T}$ is derived from the source code and which statements are allowed in the source code of $p$, is described in the following subsection.

**3.1. Recurrence Relations for $\mathcal{T}$.** The source code of a recursive procedure is considered to consist of

- simple segments of linear code, the performance of which is known a priori,
- if-statements,
- loops with a known number of iterations, e.g. for-loops or discrete loops (cf. [3]),* and
- recursive calls to the procedure itself.

In terms of a context-free grammar this is reads as follows

*This means that the number of iterations does not depend on the result of one or more recursive calls.

$$
\begin{array}{rcl}
\mathit{code}(f) & ::= & \mathbf{if}\ f \in \mathcal{F}_0\ \mathbf{then}\ \mathit{non\text{-}recursive}(f)\ \mathbf{else}\ \mathit{recursive}(f)\ \mathbf{end\ if} \\
\mathit{recursive}(f) & ::= & \mathit{seq}(f) \\
\mathit{seq}(f) & ::= & \mathit{statement}(f)\ \ \{\mathit{statement}(f)\} \\
\mathit{statement}(f) & ::= & \mathit{simple}(f)\ \mid \mathit{compound}(f)\ \mid \mathit{rproc}(f \to \overline{f}) \\
\mathit{compound}(f) & ::= & \mathit{ifs}(f)\ \mid \mathit{loops}(f) \\
\mathit{ifs}(f) & ::= & \mathbf{if}\ \mathit{cond}(f)\ \mathbf{then}\ \mathit{seq}(f)\ \mathbf{else}\ \mathit{seq}(f)\ \mathbf{end\ if} \\
\mathit{loops}(f) & ::= & \mathbf{loop}\ <\!\mathit{iter}(f)\!>\ \mathit{seq}(f)
\end{array}
$$

The syntax of $\mathit{nonrecursive}(f)$ is defined exactly the same way but $\mathit{rproc}(f \to \overline{f})$ is not part of $\mathit{statement}(f)$. By $f \to \overline{f}$ we denote that the parameters $\overline{f}$ are used for the recursive call.

We use these definitions to derive a recurrence relation for the time effort $\mathcal{T}$:

$$\mathcal{T}(f) = \tau[f \in \mathcal{F}_0] + \tau[\mathit{nonrecursive}(f)] \quad \text{if } f \in \mathcal{F}_0,$$

where the first $\tau$-constant comes from evaluating the condition whether $f$ belongs to the terminating values or not and is known a priori; the second one can be computed using the method described below, but without giving rise to a recurrence relation,

$$\mathcal{T}(f) = \tau[f \in \mathcal{F}_0] + \tau[\mathit{recursive}(f)] \quad \text{if } f \notin \mathcal{F}_0,$$

where

$$
\begin{aligned}
&\mathcal{T}[\mathit{recursive}(f)] = \mathcal{T}[\mathit{seq}(f)] \\
&\mathcal{T}[\mathit{seq}(f)] = \sum \mathcal{T}[\mathit{statement}(f)] \\
&\mathcal{T}[\mathit{ifs}(f)] = \mathcal{T}[\mathit{cond}(f)] + \begin{cases} \mathcal{T}[\mathit{seq}_{\mathtt{True}}(f)] & \text{if the condition evaluates to true,} \\ \mathcal{T}[\mathit{seq}_{\mathtt{False}}(f)] & \text{otherwise.} \end{cases} \\
&\mathcal{T}[\mathit{loops}(f)] = <\!\mathit{iter}(f)\!> \cdot \mathcal{T}[\mathit{seq}(f)] \\
&\mathcal{T}[\mathit{simple}(f)] = \tau(\mathit{simple}) \\
&\mathcal{T}[\mathit{rproc}(f \to \overline{f})] = \mathcal{T}(\overline{f})
\end{aligned}
$$

where $\tau(\mathit{simple})$ is known a priori.

Note that $<\!\mathit{iter}(f)\!>$ may depend on $f$, e.g. a for-loop with iterations depending on $f$.

## 4. Parameter Space Morphisms

Originally we were confronted with the problem that an automated tool had to perform a worst-case analysis of the timing behavior of recursive procedures (cf. [4]). Since an "informal" concept such as "complexity measure" or "size" cannot be understood by a software tool, we tried to formalize (and generalize) these concepts. This gave rise to *parameter space morphisms*. In the following sections parameter space morphisms are adapted to average-case analysis.

**Definition 4.1.** Let $\mathcal{F}'$ denote an ordered set, i.e., there exists a binary relation "$\prec$" such that for all $f', g' \in \mathcal{F}'$, $f' \neq g'$, either $f' \prec g'$ or $g' \prec f'$ holds.

A *parameter space morphism* is a mapping $\mathcal{H} : \mathcal{F} \to \mathcal{F}'$ such that for each $f \in \mathcal{F}$ and for each $g \in \mathcal{R}(f)$

$$\mathcal{H}(g) \prec \mathcal{H}(f)$$

holds.

In addition, we define the probabilities

$$p_{\mathcal{H}}(f', f) = \text{prob}\{f \mid f' = \mathcal{H}(f)\}$$

which have to fulfill $\sum_{f:f'=\mathcal{H}(f)} p_{\mathcal{H}}(f', f) = 1$.

*Remark* 4.1. It is left to the reader to compare parameter space morphisms to well-known complexity measures or size functions.

**Definition 4.2.** In the following we will frequently apply $\mathcal{H}$ to subsets of $\mathcal{F}$. Let $\mathcal{G} \subseteq \mathcal{F}$ denote such a subset. Then we write $\mathcal{H}(\mathcal{G})$ to denote the multiset $\mathcal{G}' = \mathcal{H}(\mathcal{G}) = \{\mathcal{H}(g) \mid g \in \mathcal{G}\}$.

In addition, we define

**Definition 4.3.** The direct successors of $f'$ are defined by

$$\mathcal{R}_{\mathcal{H}}(f') = \bigcup_{g:f'=\mathcal{H}(g)} \{\mathcal{H}(\mathcal{R}(g))\}\,.$$

Note that $\mathcal{H}(\mathcal{R}(g))$ is a multiset and $\mathcal{R}_{\mathcal{H}}(f')$ is a set of multisets.

## 5. Recurrence Relations for Average-Case Analysis

We are interested in the expectation of $\mathcal{T}(f)$ conditioned by $\mathcal{H}(f) = f'$, i.e., the expectation is a function of $f'$. We write

$$\mathbb{E}\mathcal{T}(f') = \sum_{f:f'=\mathcal{H}(f)} p_{\mathcal{H}}(f', f)\mathcal{T}(f)$$

to denote this expectation.

Before we prove our main result, we need further definitions.

**Definition 5.1.** We define the following quantities

$$r(f_i', f_j) = \sum_{\substack{f_i:\mathcal{H}(f_i)=f_i' \\ f_j\in\mathcal{R}(f_i)}} p_{\mathcal{H}}(f_i', f_i) \tag{5.1}$$

and

$$\tilde{r}(f_i', f_j') = \sum_{f_j:\mathcal{H}(f_j)=f_j'} r(f_i', f_j) \tag{5.2}$$

and the probabilities

$$\tilde{p}(f_i', f_j', f_j) = \frac{r(f_i', f_j)}{\tilde{r}(f_i', f_j')}. \tag{5.3}$$

**Definition 5.2.** We say that a probability distribution is *memoryless* if

$$\tilde{p}(f_i', f_j', f_j) = p_{\mathcal{H}}(f_j', f_j)$$

for all $f_i'$ and $f_j' = \mathcal{H}(f_j)$.

*Remark* 5.1. Note that being memoryless is a property of the algorithm (source code) *and* of the probability distribution.

*Remark* 5.2. If a probability distribution is memoryless, the probability distribution is independent of the recursion depth of the recursive procedure.

This is a situation typical for *divide&conquer algorithms*. Thus our method works well for all such algorithms.

Note that in contrast to our method, it seems hard to determine which class of programs will be amenable to automatic treatment with the approach presented in [8].

Now, we can find simple recurrence relations of $\mathbb{E}\mathcal{T}(f')$.

**Theorem 5.1.** *If the underlying probability distribution is memoryless, we have for all* $f_i' \notin \mathcal{H}(\mathcal{F}_0)$

$$\mathbb{E}\mathcal{T}(f_i') = \mathbb{E}\tau(f_i') + \sum_{\mathcal{R}' \in \mathcal{R}_{\mathcal{H}}(f_i')} \sum_{f_j' \in \mathcal{R}'} \tilde{r}(f_i', f_j')\mathbb{E}\mathcal{T}(f_j')$$

*where* $\mathbb{E}\tau(f_i')$ *denotes the expectation of the local time behavior of the recursive procedure conditioned by* $f_i' = \mathcal{H}(f_i)$ *without taking into account recursive calls.*

*Proof.*

$$\begin{aligned}
\mathbb{E}\mathcal{T}(f_i') &= \sum_{f_i : f_i' = \mathcal{H}(f_i)} p_{\mathcal{H}}(f_i', f_i)\mathcal{T}(f_i) \\
&= \sum_{f_i : f_i' = \mathcal{H}(f_i)} p_{\mathcal{H}}(f_i', f_i)\left[\tau(f_i) + \sum_{f_j \in \mathcal{R}(f_i)} \mathcal{T}(f_j)\right] \\
&= \sum_{f_i : f_i' = \mathcal{H}(f_i)} p_{\mathcal{H}}(f_i', f_i)\tau(f_i) + \sum_{f_i : f_i' = \mathcal{H}(f_i)} p_{\mathcal{H}}(f_i', f_i) \sum_{f_j \in \mathcal{R}(f_i)} \mathcal{T}(f_j) \\
&= \mathbb{E}(\tau(f_i')) + \sum_{f_i : f_i' = \mathcal{H}(f_i)} p_{\mathcal{H}}(f_i', f_i) \sum_{f_j \in \mathcal{R}(f_i)} \mathcal{T}(f_j) \\
&= \mathbb{E}(\tau(f_i')) + \sum_{f_i : f_i' = \mathcal{H}(f_i)} p_{\mathcal{H}}(f_i', f_i) \sum_{\mathcal{R}' \in \mathcal{R}_{\mathcal{H}}(f_i')} \sum_{f_j' \in \mathcal{R}'} \sum_{\substack{f_j : f_j' = \mathcal{H}(f_j) \\ f_j \in \mathcal{R}(f_i)}} \mathcal{T}(f_j) \\
&= \mathbb{E}(\tau(f_i')) + \sum_{\mathcal{R}' \in \mathcal{R}_{\mathcal{H}}(f_i')} \sum_{f_j' \in \mathcal{R}'} \sum_{f_j : f_j' = \mathcal{H}(f_j)} \mathcal{T}(f_j) \sum_{\substack{f_i : f_i' = \mathcal{H}(f_i) \\ f_j \in \mathcal{R}(f_i)}} p_{\mathcal{H}}(f_i', f_i)
\end{aligned}$$

which by Equation (5.1) equals

$$\begin{aligned}
&= \mathbb{E}(\tau(f_i')) + \sum_{\mathcal{R}' \in \mathcal{R}_{\mathcal{H}}(f_i')} \sum_{f_j' \in \mathcal{R}'} \sum_{f_j : f_j' = \mathcal{H}(f_j)} r(f_i', f_j)\mathcal{T}(f_j) \\
&= \mathbb{E}(\tau(f_i')) + \sum_{\mathcal{R}' \in \mathcal{R}_{\mathcal{H}}(f_i')} \sum_{f_j' \in \mathcal{R}'} \sum_{f_j : f_j' = \mathcal{H}(f_j)} \frac{r(f_i', f_j)}{p_{\mathcal{H}}(f_j', f_j)} p_{\mathcal{H}}(f_j', f_j)\mathcal{T}(f_j)
\end{aligned}$$

which by Definition 5.2 and by Equation (5.3) equals

$$
\begin{aligned}
&= \mathbb{E}(\tau(f_i')) + \sum_{\mathcal{R}'\in\mathcal{R}_{\mathcal{H}}(f_i')} \sum_{f_j'\in\mathcal{R}'} \sum_{f_j : f_j' = \mathcal{H}(f_j)} \tilde{r}(f_i', f_j') p_{\mathcal{H}}(f_j', f_j) \mathcal{T}(f_j) \\
&= \mathbb{E}(\tau(f_i')) + \sum_{\mathcal{R}'\in\mathcal{R}_{\mathcal{H}}(f_i')} \sum_{f_j'\in\mathcal{R}'} \tilde{r}(f_i', f_j') \sum_{f_j : f_j' = \mathcal{H}(f_j)} p_{\mathcal{H}}(f_j', f_j) \mathcal{T}(f_j) \\
&= \mathbb{E}(\tau(f_i')) + \sum_{\mathcal{R}'\in\mathcal{R}_{\mathcal{H}}(f_i')} \sum_{f_j'\in\mathcal{R}'} \tilde{r}(f_i', f_j') \mathbb{E}\mathcal{T}(f_j')
\end{aligned}
$$

Thus the theorem is proved. $\square$

**Example.** We show how the average-case timing behavior of *Quicksort* can be derived using our method.

We assume that Quicksort is used to sort permutations of the numbers $1, \ldots, n$ for all $n \geq 1$. Thus $\mathcal{F}$ is the set of all such permutations and $\mathcal{R}(f)$ contains two sets according to how Quicksort divides the "original permutation" into two new permutations. The set $\mathcal{F}_0$ contains the empty permutation and all permutations of size 1. Let the size of a permutation $f$ be denoted by $S(f)$.

A useful parameter space morphism $\mathcal{H}$ can be defined by $\mathcal{H} : \mathcal{F} \to \mathbb{N}$ such that $\mathcal{H}(f) = S(f)$. This implies that

$$\mathcal{R}_{\mathcal{H}}(n) = \bigcup_{n=\mathcal{H}(g)} \{\mathcal{H}(\mathcal{R}(g))\} = \bigcup_{i=0}^{n-1} \{\{i, n-i-1\}\}.$$

Furthermore we assume that

$$p_{\mathcal{H}}(n, f) = \frac{1}{n!}.$$

Let $f_n, f_m \in \mathcal{F}$ denote permutations with $\mathcal{H}(f_n) = n$ and $\mathcal{H}(f_m) = m$ respectively. Then we find that

$$r(n, f_m) = \sum_{\substack{f_n : \mathcal{H}(f_n) = n \\ f_m \in \mathcal{R}(f_n)}} = \frac{1}{n!} \cdot (n-m-1)! \cdot \frac{(n-1)!}{m!\ (n-m-1)!}.$$

Since $m+1$ elements are fixed (the elements contained in $f_m$ and the "separating element"), there are $n-m-1$ elements which can be permuted in $(n-m-1)!$ ways. Each of these permutations is encountered exactly $(n-1)!\ /\ (m!\ (n-m-1)!)$ times. Hence

$$r(n, f_m) = \frac{1}{m!\ n}.$$

Furthermore

$$\tilde{r}(n, m) = \sum_{f_m : \mathcal{H}(f_m) = m} r(n, f_m) = m! \cdot \frac{1}{m!\ n} = \frac{1}{n}.$$

Thus

$$\tilde{p}(n, f_n, f_m) = \frac{r(n, f_m)}{\tilde{r}(n, m)} = \frac{1}{m!} = p_{\mathcal{H}}(m, f_m)$$

and we obtain the following recurrence relation for $T_n = \mathbb{E}\mathcal{T}(n)$

$$\begin{aligned} T_0 &= \tau_0 \\ T_n &= \mathbb{E}\tau(n) + \sum_{\mathcal{R}' \in \mathcal{R}(n)} \sum_{m \in \mathcal{R}'} \tilde{r}(n,m) T_m \\ &= \tau_1 n + \tau_2 + \frac{2}{n} \sum_{m=0}^{n-1} T_m . \end{aligned}$$

This recurrence relation can be solved by applying standard techniques (cf. e.g. [10]).

*Remark* 5.3. Note that in the previous example $\mathbb{E}\tau(n)$ can be determined without having to perform an average-case analysis because the timing behavior does only depend on $n$. This is why $n$ is that important in the analysis of Quicksort.

In general, the only non-trivial language features incorporated in the non-recursive portion of a recursive procedure are loops. Since the number of loop iterations is the most relevant parameter for average-case timing analysis, *discrete values (sizes)* are applied so successfully in analyzing the average-case timing behavior of algorithms.

**Example.** We will study insertion into binary search trees in order to compare our approach to that presented in [8].

We assume that the input (parameter) space $\mathcal{F}$ of the algorithm is the set of all permutations of the numbers $1, \ldots, n$ for all $n \geq 1^\dagger$, where the first $n-1$ elements of the permutation have already been inserted into the tree and the last ($n$th) element is to be inserted. Let the size of a permutation $f$ be denoted by $S(f)$.

A parameter space morphism $\mathcal{H}$ can be defined by $\mathcal{H} : \mathcal{F} \to \mathbb{N}$ such that $\mathcal{H}(f) = S(f)$. Furthermore we assume that

$$p_{\mathcal{H}}(n, f) = \frac{1}{n!}.$$

Next we have to determine $r(n, f_j)$. Assume that $\mathcal{H}(f_j) = m$.

The size of the permutation of the already inserted elements is $n-1$. Assume more precisely that the element is inserted into the left subtree of the root. Since the algorithm always places the first element of a permutation into the root of the tree and all the smaller elements in the left subtree, there exist exactly $(n-2)!$ permutations which have a permutation of $1, \ldots, m-1$ as a subtree before the insertion. Taking into account the completely symmetric case for insertion into the right subtree, we obtain

$$r(n, f_j) = 2 \frac{(n-2)!}{(m-1)!} \frac{1}{n!} = \frac{2}{n(n-1)(m-1)!}$$

because $f_j$ is created from exactly one of the $(m-1)!$ permutations of $1, \ldots, m-1$.

Thus

$$\tilde{r}(n,m) = \sum_{f_j : \mathcal{H}(f_j) = m} r(n, f_j) = m! \frac{2}{n(n-1)(m-1)!} = \frac{2m}{n(n-1)}$$

$^\dagger$In [8] a set of real numbers is used, but this is no essential difference (cf. Section 2.3 of [13]).

and

$$\tilde{p}(n,m,f_j) = \frac{(2/(n(n-1)(m-1)!))}{2m/(n(n-1))} = \frac{1}{m!} = p_{\mathcal{H}}(m,f_j)$$

which shows that the probability distribution is memoryless, and we can state our final recurrence relation for the average timing behavior of the algorithm $\mathbb{E}\mathcal{T}(n) = T_n$ [‡]

$$T_1 = \tau_0,$$

$$T_n = \tau_1 + \frac{2}{n(n-1)} \sum_{m=0}^{n-1} m T_m.$$

Solving this recurrence relation we obtain

$$T_n = 2(H_n - 1)\tau_1 + \tau_0,$$

where $H_n$ denotes the $n$th harmonic number $H_n = \sum_{i=1}^{n} \frac{1}{i} = \log n + \gamma + O(1/n)$.

The same example performed in [8] takes about six pages to be solved. The method developed in [8] relies on manipulating several complicated equations involving integrals and sums.

On the other hand our method uses only several simple well-known facts on binary search trees (which of course are used in [8] too) and simple manipulations on the involved probabilities to show that the proposed probability distribution is memoryless. Then by a simple application of Theorem 5.1 the recurrence relations for the timing behavior are derived.

**Example.** Next we are going to investigate insertion into binary tries (cf. e.g. [10, 20]).

First notice that tries cannot be analyzed by our method using a finite model[§] such as that described in [5] because this does not result in a memoryless probability distribution. The reason for this is that a finite model identifies each stage of the recursion depth exactly.

Instead we have to resort to an infinite model. Thus we assume that the keys are formed by bit strings of infinite length.

We define a morphism $\mathcal{H}(f) = n$ where $n$ denotes the cardinality of the set stored in the trie and we assume that $p_{\mathcal{H}}(n,f) = \frac{1}{n!}$.

By making use of the well-known partitioning process for tries we obtain ($\mathcal{H}(f_j) = k-1$)

$$r(n,f_j) = \frac{1}{2^{n-1}} \binom{n-1}{k-1} \frac{1}{k!}$$

since similar to the previous example we assume that the $n$th element is inserted into a trie which already holds $n-1$ elements, and the left subtree of the trie holds $k-1$ elements. Thus

$$\tilde{r}(n,k) = 2^{1-n} \binom{n-1}{k-1}$$

[‡]Note that $\mathbb{E}\tau(n)$ is constant for all reasonable implementations.

[§]In the finite model the key is supposed to consist of a finite number of bits.

and

$$\tilde{p}(n,k,f_j) = \frac{1}{k!} = p_{\mathcal{H}}(k,f)$$

which shows that the probability distribution is memoryless.

Applying Theorem 5.1 we derive a recurrence relation for the average timing behavior of the algorithm $\mathbb{E}\mathcal{T}(n) = T_n$[¶]

$$\begin{aligned} T_1 &= \tau_0, \\ T_n &= \tau_1 + 2^{1-n} \sum_{k=1}^{n-1} \binom{n-1}{k-1} T_k. \end{aligned}$$

This can be interpreted as a certain weighted path length and solved by standard techniques.

## 6. Conclusion

In this paper we have presented a method for deriving recurrence relations from the source code of recursive procedures. These recurrence relations describe the average-case timing behavior of the underlying recursive procedure. A fairly general source code description is given, which covers most high-level programming languages. In addition, the programmer has to supply information on the probability distribution of the parameter space of the procedure.

The method is based on a generalization of *complexity measures* or *size functions* usually applied when analyzing algorithms.

Using our method it is extremely easy to derive recurrence relations for studying the average-case timing behavior of recursive procedures and functions, even for a non-expert in the fields of analyzing algorithms and data-structures.

Moreover it should be straight-forward to implement our method in an automated average-case analysis tool.

Compared to [8] our proofs are much more simpler and less knowledge on probability theory is needed; compared to [7] the connection with the source code is much closer and less knowledge on combinatorics is needed.

## References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.
3. J. Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994.
4. J. Blieberger and R. Lieger. Worst-case space and time complexity of recursive procedures. *Real-Time Systems*, 1994. (to appear).
5. P. Flajolet. Methods in the analysis of algorithms: Evaluations of a recursive partitioning process. In *Proc. of the FCT-Conf.*, pages 141–158, Borgholm, 1983. Lecture Notes in Comp. Sc.
6. P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-upsilon-omega: An assistant algorithms analyzer. In *Proc. AECC'6*, Rome, 1988. Notes in Comp. Sc.
7. P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science*, 79(1):37–109, 1991.
8. T. Hickey and J. Cohen. Automating program analysis. *Journal of ACM*, 35(1):185–220, 1988.

[¶]Note that $\mathbb{E}\tau(n)$ is constant for all reasonable implementations.

9. D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., second edition, 1973.
10. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., 1973.
11. D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., second edition, 1981.
12. D. Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22:328–350, 1981.
13. H. M. Mahmoud. *Evolution of Random Search Trees*. John Wiley & Sons, New York, 1992.
14. K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, Berlin, 1984.
15. D. L. Metayer. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988.
16. L. H. Ramshaw. *Formalizing the analysis of algorithms*. PhD thesis, Stanford University, 1979.
17. M. Rosendahl. Automatic complexity analysis. *Functional Programming Languages and Computer Architecture*, pages 144–156, 1989.
18. V. Sarkar. Determining average program execution times and their variance. *ACM SIGPLAN Notices (Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation)*, 24(7):298–309, 1989.
19. R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, second edition, 1988.
20. J. S. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 431–524. North-Holland, 1990.
21. B. Wegbreit. Mechanical program analysis. *Communications of ACM*, 18(9):528–539, 1975.

DEPARTMENT OF AUTOMATION (183/1), TECHNICAL UNIVERSITY OF VIENNA, TREITLSTR. 1/4, A-1040 VIENNA

*E-mail*: blieb@auto.tuwien.ac.at

*E-mail*: rlieger@auto.tuwien.ac.at