New results on mathematical foundations of asymptotic complexity analysis of algorithms via complexity spaces

S. Romaguera, P. Tirado, O. Valero

Abstract

In 1995, M.P. Schellekens introduced the theory of complexity (quasi-metric) spaces as a part of the development of a topological foundation for the asymptotic complexity analysis of programs and algorithms [Electron. Notes Theor. Comput. Sci. 1 (1995), 211-232]. The applicability of this theory to the asymptotic complexity analysis of Divide and Conquer algorithms was also illustrated by Schellekens in the same reference. In particular, he gave a new formal proof, based on the use of the Banach fixed point theorem, of the well-known fact that the asymptotic upper bound of the average running time of computing of Mergesort belongs to the asymptotic complexity class of $n \log_2 n$. Motivated by the utility of the quasi-metric approach for the asymptotic complexity analysis based on the use of fixed point techniques and complexity spaces, on one hand we extend Schellekens' method in order to yield asymptotic upper bounds for a class of algorithms whose running time of computing leads to recurrence equations different from the Divide and Conquer ones, and, on the other hand, we improve the original Schellekens method by introducing a new fixed point technique for providing lower asymptotic bounds for the running time of computing of the aforesaid algorithms. We illustrate and validate the developed method applying our results to provide the asymptotic complexity class (asymptotic upper and lower bounds), among others, of the celebrated recursive algorithm that solves the problem of Hanoi Towers.

quasi-metric, complexity space, fixed point, improver, worsener, complexity class, Quicksort, Hanoi, Largetwo.

AMS codes: 47H10, 54E35, 54E50, 68Q25, 68Q30.

1 The fundamentals of asymptotic complexity analysis of algorithms via complexity spaces

Throughout this paper the letters $\mathbb{R}^+$ and $\mathbb{N}$ will denote the set of nonnegative real numbers and the set of positive integer numbers, respectively.

Our basic reference for complexity analysis of algorithms is [1].

In Computer Science the complexity analysis of an algorithm is based on determining mathematically the quantity of resources needed by the algorithm in order to solve the problem for which it has been designed. A typical resource, playing a central role in complexity analysis, is the running time of computing. Since there are often many algorithms to solve the same problem, one objective of the complexity analysis is to assess which of them is faster when large inputs are considered. To this end, it is required to compare their running time of computing. This is usually done by means of the asymptotic analysis in which the running time of an algorithm is denoted by a function $T : \mathbb{N} \to (0, \infty]$ in such a way that $T(n)$ represents the time taken by the algorithm to solve the problem under consideration when the input of the algorithm is of size n. Of course the running time of an algorithm does not only depend on the input size n, but it depends also on the particular input of the size n (and the distribution of the data). Thus the running time of an algorithm is different when the algorithm processes certain instances of input data of the same size n. As a consequence, in general it is necessary to distinguish three possible behaviors when the running time of an algorithm is discussed. These are the so-called best case, the worst case and the average case. The best case and the worst case for an input of size n are defined by the minimum and the maximum running time of computing over all inputs of the size n, respectively. The average case for an input of size n is defined by the expected value or average running time of computing over all inputs of the size n.

Given an algorithm, to determine exactly the function which describes its running time of computing is in general an arduous task. However, in most situations is more useful to know the running time of computing of an algorithm in an “approximate” way than in an exact one. For this reason the asymptotic complexity analysis of algorithms focus its interest in obtaining "approximate" running times of computing.

In order to recall pertinent notions from asymptotic complexity analysis, let us assume that $f : \mathbb{N} \to (0, \infty]$ denotes the running time of computing of a certain algorithm. In addition, consider that there exists a function $g : \mathbb{N} \to (0, \infty]$ such that there exist, simultaneously, $n_0 \in \mathbb{N}$ and $c > 0$ satisfying $f(n) \leq cg(n)$ for all $n \in \mathbb{N}$ with $n \geq n_0$ ($\leq$ and $\geq$ stand for the usual orders on $\mathbb{R}^+$). Then, the function g provides an asymptotic upper bound of the running time of the studied algorithm. Thus, if we do not know the exact expression of the function f, then the function g gives an "approximate" information of the running time of the algorithm in the sense that the algorithm takes a time to solve the problem bounded above by g. Following the standard notation, when g is an asymptotic upper bound of f we write $f \in \mathcal{O}(g)$.

Sometimes in the analysis of the complexity of an algorithm is useful to assess an asymptotic lower bound of the running time of computing. In this case the Ω-notation plays a central role. Thus the statement $f \in \Omega(g)$ means that there exist $n_0 \in \mathbb{N}$ and $c > 0$ such that $cg(n) \leq f(n)$ for all $n \in \mathbb{N}$ with $n \geq n_0$. Of course, and similarly to the $\mathcal{O}$-notation case, when the time taken by the algorithm to solve the problem f is unknown, the function g yields an "approximate" information of the running time of the algorithm in the sense that the algorithm takes a time to solve the problem bounded below by g.

It is clear that the best situation, when the complexity of an algorithm is discussed, matches up with the case in which we can find a function $g : \mathbb{N} \to (0, \infty]$ in such a way that the running time f holds the condition $f \in \mathcal{O}(g) \cap \Omega(g)$, denoted by $f \in \Theta(g)$, because, in this case, we obtain a "tight "asymptotic bound of f and, thus, a total asymptotic information about the time taken by the algorithm to solve the problem under consideration. From now on, we will say that f belongs to the asymptotic complexity class of g whenever $f \in \Theta(g)$.

Hence, from an asymptotic complexity analysis viewpoint, to determine the running time of an algorithm consists of obtaining its asymptotic complexity class.

In 1995, M.P. Schellekens introduced a new mathematical framework, known as complexity spaces, as a part of the development of a topological foundation for the asymptotic complexity analysis of algorithms ([6]). This approach is based on the notion of quasi-metric space.

Following [4], a quasi-metric on a non-empty set X is a function $d : X \times X \to \mathbb{R}^+$ such that for all $x, y, z \in X$: (i) $d(x,y) = d(y,x) = 0 \Leftrightarrow x = y$; (ii) $d(x,y) \leq d(x,z) + d(z,y)$.

Of course a metric on a non-empty set X is a quasi-metric d on X satisfying, in addition, the following condition for all $x, y \in X$: (iii) $d(x,y) = d(y,x)$.

A quasi-metric space is a pair (X,d) such that X is a non-empty set and d is a quasi-metric on X.

Each quasi-metric d on X generates a T_0-topology $\mathcal{T}(d)$ on X which has as a base the family of open d-balls $\{B_d(x,\varepsilon) : x \in X,\ \varepsilon > 0\}$, where $B_d(x,\varepsilon) = \{y \in X : d(x,y) < \varepsilon\}$ for all $x \in X$ and $\varepsilon > 0$.

Given a quasi-metric d on X, the function d^s defined on $X \times X$ by $d^s(x,y) = \max(d(x,y), d(y,x))$ is a metric on X.

A quasi-metric space (X,d) is called bicomplete if the metric space (X,d^s) is complete.

A well-known example of a bicomplete quasi-metric space is the pair $((0,\infty], u_{-1})$, where $u_{-1}(x,y) = \max\left(\frac{1}{y} - \frac{1}{x}, 0\right)$ for all $x, y \in (0,\infty]$. Obviously we adopt the convention that $\frac{1}{\infty} = 0$. The quasi-metric space $((0,\infty], u_{-1})$ plays a central role in the Schellekens approach. Indeed, let us recall that the complexity (quasi-metric) space is the pair $(\mathcal{C}, d_{\mathcal{C}})$, where

$$\mathcal{C} = \{f : \mathbb{N} \to (0,\infty] : \sum_{n=1}^{\infty} 2^{-n} \frac{1}{f(n)} < \infty\}$$

and $d_{\mathcal{C}}$ is the bicomplete quasi-metric on $\mathcal{C}$ defined by

$$d_{\mathcal{C}}(f,g) = \sum_{n=1}^{\infty} 2^{-n} \max\left(\frac{1}{g(n)} - \frac{1}{f(n)}, 0\right).$$

According to [6], since every reasonable algorithm, from a computability viewpoint, must hold the "convergence condition" $\sum_{n=1}^{\infty} 2^{-n} \frac{1}{f(n)} < \infty$, it is possible to associate each algorithm with a function of $\mathcal{C}$ in such a way that such a function represents, as a function of the size of the input data, its running time of computing. Because of this, the elements of $\mathcal{C}$ are called complexity functions. Moreover, given two functions $f, g \in \mathcal{C}$, the numerical value $d_{\mathcal{C}}(f,g)$ (the complexity distance from f to g) can be interpreted as the relative progress made in lowering the complexity by replacing any program P with complexity function f by any program Q with complexity function g. Therefore, if $f \neq g$, the condition $d_{\mathcal{C}}(f,g) = 0$ can be read as the program P is at least as efficient as the program Q (indeed, note that $d_{\mathcal{C}}(f,g) = 0 \Leftrightarrow f(n) \leq g(n)$ for all $n \in \mathbb{N}$). In fact, the condition $d_{\mathcal{C}}(f,g) = 0$ implies that $f \in \mathcal{O}(g)$.

Notice that the asymmetry of the complexity distance $d_{\mathcal{C}}$ plays a central role in order to provide information about the increase of complexity whenever a program is replaced by another one. A metric will be able to yield information on the increase but it, however, will not reveal which program is more efficient.

The applicability of the theory of complexity spaces to the asymptotic complexity analysis of algorithms was illustrated by Schellekens in [6]. In particular, he gave, among other things, a new proof of the well-known fact that that the function $f \in \mathcal{C}$, given by $f(1) = c > 0$ and $f(n) = n \log_2 n$ for all $n \in \mathbb{N}$ with $n > 1$, is an asymptotic upper bound of the average running time of computing of Mergesort. To this end, he introduced a method, based on the below quasi-metric version of Banach's fixed point theorem, to analyze the running time of computing of the general class of Divide and Conquer algorithms (observe that Mergesort is a Divide and Conquer algorithm).

Theorem 1. *Let f be a mapping from a bicomplete quasi-metric space (X,d) into itself such that there exists $s \in [0,1)$ satisfying*

$$d(f(x), f(y)) \leq s d(x,y), \tag{1}$$

for all $x, y \in X$. Then f has a unique fixed point.

Let us recall that a mapping f from a quasi-metric space (X,d) into itself holding inequality (1) is said to be contractive with contractive constant s.

Next we provide a general view of the aforenamed method with the aim of motivating our subsequent work.

A Divide and Conquer algorithm solves a problem of size n ($n \in \mathbb{N}$) splitting it into a subproblems of size $\frac{n}{b}$, for some constants a, b with $a, b \in \mathbb{N}$ and $a, b > 1$, and solving them separately by the same algorithm. After obtaining the solution of the subproblems, the algorithm combines all subproblem solutions to give a global solution to the original problem. The recursive structure of a Divide and Conquer algorithm leads to a recurrence equation for the running time of computing. In many cases the running time of a

Divide and Conquer algorithm is the solution to a recurrence equation of the form

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ aT(\frac{n}{b}) + h(n) & \text{if } n \in \mathbb{N}_b \end{cases}, \tag{2}$$

where $\mathbb{N}_b = \{b^k : k \in \mathbb{N}\}$, $c > 0$ denotes the complexity on the base case (i.e. the problem size is small enough and the solution takes constant time), $h(n)$ represents the time taken by the algorithm in order to divide the original problem into a subproblems and to combine all subproblems solutions into a unique one ($h \in \mathcal{C}$ with $h(n) < \infty$ for all $n \in \mathbb{N}$).

Notice that for Divide and Conquer algorithms, it is typically sufficient to obtain the complexity on inputs of size n with n ranges over the set $\mathbb{N}_b$ ([1]).

Mergesort is a typical and well-known example of a Divide and Conquer algorithm whose running time of computing satisfies the recurrence equation (2) (see [1] for a fuller description).

In order to compute the running time of computing of a Divide and Conquer algorithm satisfying the recurrence equation (2), it is necessary to show that such a recurrence equation has a unique solution and, later, to obtain the asymptotic complexity class of such a solution. The method introduced by Schellekens allows to show that the equation (2) has a unique solution, and provides an upper asymptotic complexity bound of the solution in the following way:

Denote by $\mathcal{C}_{b,c}$ the subset of $\mathcal{C}$ given by

$$\mathcal{C}_{b,c} = \{f \in \mathcal{C} : f(1) = c \text{ and } f(n) = \infty \text{ for all } n \in \mathbb{N} \backslash \mathbb{N}_b \text{ with } n > 1\}.$$

Since the quasi-metric space $(\mathcal{C}, d_{\mathcal{C}})$ is bicomplete ([5]) and the set $\mathcal{C}_{b,c}$ is closed in $(\mathcal{C}, d_{\mathcal{C}}^s)$, we have that the quasi-metric space $(\mathcal{C}_{b,c}, d_{\mathcal{C}}|_{\mathcal{C}_{b,c}})$ is bicomplete.

Next we associate a functional $\Phi_T : \mathcal{C}_{b,c} \to \mathcal{C}_{b,c}$ with the recurrence equation (2) of a Divide and Conquer algorithm given as follows:

$$\Phi_T(f)(n) = \begin{cases} c & \text{if } n = 1 \\ \infty & \text{if } n \in \mathbb{N} \backslash \mathbb{N}_b \text{ and } n > 1 \\ af(\frac{n}{b}) + h(n) & \text{otherwise} \end{cases}. \tag{3}$$

Of course a complexity function in $\mathcal{C}_{b,c}$ is a solution to the recurrence equation (2) if and only if it is a fixed point of the functional Φ_T. Then, Schellekens proved ([6]) that

$$d_{\mathcal{C}}|_{\mathcal{C}_{b,c}}(\Phi_T(f), \Phi_T(g)) \leq \frac{1}{a} d_{\mathcal{C}}|_{\mathcal{C}_{b,c}}(f, g) \tag{4}$$

for all $f, g \in \mathcal{C}_{b,c}$. So, by Theorem 1, the functional $\Phi_T : \mathcal{C}_{b,c} \to \mathcal{C}_{b,c}$ has a unique fixed point and, thus, the recurrence equation (2) has a unique solution.

In order to obtain the upper asymptotic complexity bound of the solution to the recurrence equation (2), Schellekens introduced a special class of functionals known as improvers.

Let $C \subseteq \mathcal{C}$. A functional $\Phi : C \to C$ is called an improver with respect to a function $f \in C$ provided that $\Phi^n(f) \leq \Phi^{n-1}(f)$ for all $n \in \mathbb{N}$. Of course $\Phi^0(f) = f$. Observe that an improver is a functional which corresponds to a transformation on programs in such a way that the iterative applications of the transformation yield, from a complexity point of view, an improved program at each step of the iteration. Note that under the assumption that the functional Φ is monotone, to show that Φ is an improver with respect to $f \in C$ is equivalent to verify that $\Phi(f) \leq f$.

Taking into account the exposed facts, Schellekens stated the following result ([6]).

Theorem 2. *A Divide and Conquer recurrence of the form (2) has a unique solution f_T in $\mathcal{C}_{b,c}$. Moreover, if the functional Φ_T associated with (2) is an improver with respect to some function $g \in \mathcal{C}_{b,c}$, then the solution to the recurrence equation satisfies that $f_T \in \mathcal{O}(g)$.*

He also obtained an asymptotic upper bound of the running time of computing of Mergesort in order to illustrate the usefulness of Theorem 2. In the particular case of Mergesort (average case), the running time of computing satisfies the following particular case of recurrence equation (2):

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(\frac{n}{2}) + \frac{n}{2} & \text{if } n \in \mathbb{N}_2 \end{cases} . \tag{5}$$

It is clear that Theorem 2 shows that the recurrence equation (5) has a unique solution f_T^M in $\mathcal{C}_{2,c}$. In addition, Schellekens proved that the functional Φ_T induced by the recurrence equation (5) is an improver with respect to a complexity function $g_k \in \mathcal{C}_{2,c}$ (with $k > 0$, $g_k(1) = c$ and $g_k(n) = kn\log_2(n)$ for all $n \in \mathbb{N}_2$) if and only if $k \geq \frac{1}{2}$. Therefore, by Theorem 2, we conclude that $f_T^M \in \mathcal{O}(g_{\frac{1}{2}})$, i.e. Theorem 2 provides a formal proof, based on fixed point techniques, of the well-known fact that the running time of computing (average case) f_T^M of Mergesort is in $\mathcal{O}(n \log_2 n)$, i.e. that the complexity function $g_{\frac{1}{2}}$, or equivalently $\mathcal{O}(n \log_2 n)$, gives an asymptotic upper bound of f_T^M. Furthermore, in [6] it is pointed out that an asymptotic lower bound of the running time of Mergesort (average case) belongs to $\Omega(n \log_2 n)$ (following standard arguments which are not based on the use of fixed point techniques). So Mergesort running time (average case) belongs to the complexity class $\Theta(n \log_2 n)$.

Of course, Schellekens' method, without meaning to compete with the standard and classical techniques to analyze the complexity of algorithms, has the advantage of allowing to apply similar ideas to those presented by D.S. Scott ([7], [8]) in modelling the meaning of recursive denotational specifications of algorithms via fixed point techniques in such a way that the notion of "iterative approximations", typical of the topological Scott framework, is captured trough the concept of improver functional.

2 Extending the applicability of complexity spaces: new algorithms and recurrence equations

In spite of it seems natural that the complexity analysis of Divide and Conquer algorithms always leads up to Divide and Conquer recurrence equations of type (2), this is not the case. Sometimes this kind of recursive algorithms yields recurrence equations that differ from (2). A well-known example of this sort of situations is provided by Quicksort (worst case) ([2]). In particular, the running time of computing (worst case) of the aforesaid algorithm is the solution to the recurrence equation given exactly as follows:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n-1) + jn & \text{if } n \geq 2 \end{cases}, \tag{6}$$

with $j > 0$ and where c is the time taken by the algorithm in the base case. Observe that in this case it is not necessary to restrict the input size of the data to the set $\mathbb{N}_b$ for some $b \in \mathbb{N}$ with $b > 1$.

Although the recurrence equations associated to the running time of computing of Mergesort and Quicksort do not belong to the same class for the cases discussed above, it is clear that the main relationship between both algorithms is given by the fact that them belong to the Divide and Conquer algorithms class and, thus, they are recursive algorithms.

Of course, the class of recursive algorithms is wider than the Divide and Conquer. An illustrative example of recursive algorithm, which does not belong to the Divide and Conquer family, is provided by Hanoi. Hanoi solves the Towers of Hanoi puzzle (see [2] and [3]). In this case, under the uniform cost criterion assumption, the running time of computing is the solution to a recurrence equation given by

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n-1) + d & \text{if } n \geq 2 \end{cases}, \tag{7}$$

with $c, d > 0$ and where c represents the time taken by the algorithm to solve the base case. Notice that does not make sense distinguish three possible running time behaviors for Hanoi, since the distribution of the input data is always the same for each size n.

The fact that the class of recursive algorithms is wider than the Divide and Conquer inspires to wonder two questions. On one hand, whether one can obtain a family of recurrence equations in such a way that the complexity analysis of those algorithms whose running time of computing is a solution either to recurrence equations of type (6) and (7) or to a Divide and Conquer one can be carried out from it. On the other hand, whether such a complexity analysis can be done via an extension of the fixed point technique of Schellekens.

Clearly, the recurrence equations that yield the running time of computing of the above aforesaid algorithms can be considered as particular cases of the following general one:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ aT(n-1) + h(n) & \text{if } n \geq 2 \end{cases}, \tag{8}$$

with $c > 0$, $a \geq 1$ and $h \in \mathcal{C}$ such that $h(n) < \infty$ for all $n \in \mathbb{N}$.

Clearly, the discussion of the complexity of the Divide and Conquer algorithms introduced in Section 1 can be carried out from the family of recurrence equations of type (8). This is possible because the running time of computing of the aforementioned algorithms leads to recurrence equations can be seen as a particular case of our last general family of recurrence equations. Indeed, a Divide and Conquer recurrence equation

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ aT(\frac{n}{b}) + h(n) & \text{if } n \in \mathbb{N}_b \end{cases} \tag{9}$$

can be transformed into the following one

$$S(m) = \begin{cases} c & \text{if } m = 1 \\ aS(m-1) + r(m) & \text{if } m > 1 \end{cases}, \tag{10}$$

where $S(m) = T(b^{m-1})$ and $r(m) = h(b^{m-1})$ for all $m \in \mathbb{N}$. Recall that $\mathbb{N}_b = \{b^k : k \in \mathbb{N}\}$. Of course, the analysis, in the Shellekens spirit, of the recurrence equation (10) allows immediately to study Divide and Conquer recurrence equations.

As well as the exposed advantage, the relevance of the family of recurrence equations of type (8) is intensified by the fact that the running time of certain non-recursive algorithms also matches up with the solution to a recurrence equation that can be retrieved as a particular case of the general recurrence equation (8). A good example is provided by Largetwo. This algorithm finds the two largest entries in one-dimensional array of size $n \in \mathbb{N}$ with $n > 1$ (for a deeper discussion we refer the reader to [2]). The running time of computing of Largetwo (average case) can be associated with the solution to the recurrence equation given as follows:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n-1) + 2 - \frac{1}{n} & \text{if } n \geq 2 \end{cases}, \tag{11}$$

where c is, again, the time taken by the algorithm in the base case, i.e. when the input data is a one-diemensional array with only one element or the array does not contain input data. Notice that Largetwo needs inputs data with size at least 2.

In what follows our purpose is to demonstrate that the Schellekens fixed point technique can be used satisfactorily to discuss the complexity of those algorithms whose running time of computing yields with a recurrence equation of type (8). In particular, we prove that the aforesaid recurrence equation has a unique solution and, in addition, we obtain the complexity class (asymptotic upper and lower bounds) of such a solution. Similarly to Schellekens' approach, our technique to obtain the asymptotic upper bound is based on the use of the improver functional induced by the recurrence equation. Nevertheless, we introduce a new kind of functionals, that we have called "worsener" functionals, with the aim of obtaining the asymptotic lower bound of the solution to the recurrence equation. In order to provide the complexity class of an algorithm whose running time satisfies a recurrence equation of type (8) we prove that it is enough to search among all complexity functions for which the functional associated to the recurrence equation is simultaneously an improver and a worsener. Finally, in order, on one hand, to validate our new results and, on the other hand, to show the potential applicability of the developed theory to complexity analysis in Computer Science, we shall discuss the running time of Quicksort (worst case), Hanoi and Largetwo (average case), respectively.

3 The new fixed point technique in complexity analysis

Is this section we provide the new fixed point technique to show the existence and uniqueness of the solution to the recurrence equations of type (8) and the announced mathematical method to obtain the complexity class of those algorithms whose running time satisfies the recurrence equation under study.

3.1 The existence and uniqueness of solution

Consider the subset $\mathcal{C}_c$ of $\mathcal{C}$ given by

$$\mathcal{C}_c = \{f \in \mathcal{C} : f(1) = c\}.$$

Define the functional $\Psi_T : \mathcal{C}_c \rightarrow \mathcal{C}_c$ by

$$\Psi_T(f)(n) = \begin{cases} c & \text{if } n = 1 \\ af(n-1) + h(n) & \text{if } n \geq 2 \end{cases} \tag{12}$$

for all $f \in \mathcal{C}_c$. It is clear that a complexity function in $\mathcal{C}_c$ is a solution to the recurrence equation (8) if and only if it is a fixed point of the functional Ψ_T.

The next result supplies the bicompleteness of the pair $(\mathcal{C}_c, d_{\mathcal{C}}|_{\mathcal{C}_c})$.

Proposition 3. *The subset $\mathcal{C}_c$ is closed in $(\mathcal{C}, d^s_{\mathcal{C}})$.*

Proof. Let $g \in \overline{\mathcal{C}_c}^{d^s_{\mathcal{C}}}$ and $(f_i)_{i\in\mathbb{N}} \subset \mathcal{C}_c$ with $\lim_{i\to\infty} d^s_{\mathcal{C}}(g, f_i) = 0$.

First of all we prove that $g \in \mathcal{C}$. Indeed, given $\varepsilon > 0$, there exist $i_0, n_0 \in \mathbb{N}$ such that $d^s_{\mathcal{C}}(g, f_i) < \frac{\varepsilon}{2}$ whenever $i \geq i_0$ and $\sum_{n=n_0+1}^{\infty} 2^{-n} \frac{1}{f_{i_0}(n)} < \frac{\varepsilon}{2}$. Whence

$$\begin{aligned}
\sum_{n=n_0+1}^{\infty} 2^{-n} \frac{1}{g(n)} &= \sum_{n=n_0+1}^{\infty} 2^{-n} \left(\frac{1}{g(n)} - \frac{1}{f_{i_0}(n)} + \frac{1}{f_{i_0}(n)} \right) \\
&\leq \sum_{n=n_0+1}^{\infty} 2^{-n} \left| \frac{1}{g(n)} - \frac{1}{f_{i_0}(n)} \right| + \sum_{n=n_0+1}^{\infty} 2^{-n} \frac{1}{f_{i_0}(n)} \\
&\leq d^s_{\mathcal{C}}(g, f_{i_0}) + \sum_{n=n_0+1}^{\infty} 2^{-n} \frac{1}{f_{i_0}(n)} \\
&< \varepsilon.
\end{aligned}$$

It follows that $g \in \mathcal{C}$.

Now suppose for the purpose of contradiction that $g \notin \mathcal{C}_c$. Then $g(1) \neq c$. Put $\varepsilon = 2^{-1} | \frac{1}{g(1)} - \frac{1}{c} |$. Then there exists $i_0 \in \mathbb{N}$ such that $d^s_{\mathcal{C}}(g, f_i) < \varepsilon$ whenever $i \geq i_0$. Thus

$$\sum_{n=1}^{\infty} 2^{-n} \left| \frac{1}{g(n)} - \frac{1}{f_i(n)} \right| < \varepsilon$$

whenever $i \geq i_0$. As a result we have that

$$\varepsilon = 2^{-1} | \frac{1}{g(1)} - \frac{1}{c} | \leq \sum_{n=1}^{\infty} 2^{-n} \left| \frac{1}{g(n)} - \frac{1}{f_{i_0}(n)} \right| < \varepsilon,$$

which is a contradiction. So $g(1) = c$.

Therefore we have shown that $\overline{\mathcal{C}_c}^{d^s_{\mathcal{C}}} = \mathcal{C}_c$. ■

Since the metric space $(\mathcal{C}, d^s_{\mathcal{C}})$ is complete and, by the preceding proposition, the subset $\mathcal{C}_c$ is closed in $(\mathcal{C}, d^s_{\mathcal{C}})$ we immediately obtain the following consequence.

Corollary 4. *The quasi-metric space $(\mathcal{C}_c, d_{\mathcal{C}}|_{\mathcal{C}_c})$ is bicomplete.*

Theorem 5. *The functional Ψ_T is a contraction from $(\mathcal{C}_c, d_{\mathcal{C}}|_{\mathcal{C}_c})$ into itself with contractive constant $\frac{1}{2a}$.*

Proof. Let $f, g \in \mathcal{C}_c$. Then

$$\begin{aligned}
d_{\mathcal{C}}|_{\mathcal{C}_c}(\Psi_T(f),\Psi_T(g)) &= \sum_{n=1}^{\infty} 2^{-n}\max\left(\frac{1}{\Psi_T(g)(n)}-\frac{1}{\Psi_T(f)(n)},0\right)\\
&= \sum_{n=2}^{\infty} 2^{-n}\max\left(\frac{1}{ag(n-1)+h(n)}-\frac{1}{af(n-1)+h(n)},0\right)\\
&= \sum_{n=2}^{\infty} 2^{-n}\max\left(\frac{af(n-1)-ag(n-1)}{a^2g(n-1)f(n-1)+s(n)},0\right)\\
&\leq \sum_{n=2}^{\infty} 2^{-n}\max\left(\frac{af(n-1)-ag(n-1)}{a^2g(n-1)f(n-1)},0\right)\\
&= \sum_{n=2}^{\infty} 2^{-n}\max\left(\frac{1}{ag(n-1)}-\frac{1}{af(n-1)},0\right)\\
&= \frac{1}{2a}d_{\mathcal{C}}|_{\mathcal{C}_c}(f,g),
\end{aligned}$$

where $s(n)=ah(n)(f(n-1)+g(n-1))+h(n)^2$ for all $n\geq 2$.

Now the existence and uniqueness of the fixed point $f_T\in\mathcal{C}_c$ of Ψ_T follow from the fact that $a\geq 1$ and Corollary 4 and Theorem 1. ■

From the above theorem we can immediately gather that a recurrence equation of the form (8) has a unique solution f_T in $\mathcal{C}_c$ which matches up with the running time of computing of the algorithm under study considered in each case.

3.2 The complexity class of the solution

Next we provide a method (Theorem 7 below) to describe the complexity of those algorithms whose running time of computing satisfies a recurrence equation of type (8). To this end we need the following auxiliary result.

Lemma 6. *Let C be a subset of $\mathcal{C}$ such that the quasi-metric space $(C,d_{\mathcal{C}}|_C)$ is bicomplete and let $\Psi:C\to C$ be a contraction with fixed point $f\in C$ and contractive constant s. Then the following statements hold:*

1) If there exists $g\in C$ with $d_{\mathcal{C}}|_C(\Psi(g),g)=0$, then $d_{\mathcal{C}}|_C(f,g)=0$.

2) If there exists $g\in C$ with $d_{\mathcal{C}}|_C(g,\Psi(g))=0$, then $d_{\mathcal{C}}|_C(g,f)=0$.

Proof. 1) Assume that there exists $g \in C$ such that $d_{\mathcal{C}}|_C(\Psi(g), g) = 0$. Suppose for the purpose of contradiction that $d_{\mathcal{C}}|_C(f, g) > 0$. Then we have that

$$\begin{aligned} d_{\mathcal{C}}|_C(f,g) &\leq d_{\mathcal{C}}|_C(f,\Psi(g)) + d_{\mathcal{C}}|_C(\Psi(g),g) = d_{\mathcal{C}}|_C(f,\Psi(g)) \\ &\leq d_{\mathcal{C}}|_C(f,\Psi(f)) + d_{\mathcal{C}}|_C(\Psi(f),\Psi(g)) \\ &= d_{\mathcal{C}}|_C(\Psi(f),\Psi(g)) \leq s d_{\mathcal{C}}|_C(f,g). \end{aligned}$$

From the preceding inequality we deduce that $1 \leq s$, which is imposible. So $d_{\mathcal{C}}|_C(f, g) = 0$.

2) Similar arguments to those given in the proof of 1) remain valid to prove the thesis of 2). ■

Note that if a complexity function f represents the running time of computing of an algorithm under study, the fact that there exists a complexity function g satisfying the condition $d_{\mathcal{C}}|_C(\Psi(g), g) = 0$ ($d_{\mathcal{C}}|_C(g, \Psi(g)) = 0$) in the preceding lemma provides an asymptotic upper (lower) bound of the aforesaid running time, since $d_{\mathcal{C}}|_C(f, g) = 0$ ($d_{\mathcal{C}}|_C(g, f) = 0$) implies that $f \in \mathcal{O}(g)$ ($f \in \Omega(g)$).

In the light of Lemma 6 we observe that in order to get an asymptotic upper bound of the running time of computing of an algorithm whose running time matches up with the fixed point of a contraction $\Psi : C \to C$ ($C \subseteq \mathcal{C}$), it is enough to check if such a mapping satisfies the condition $\Psi(g) \leq g$ for any complexity function even if Ψ is not monotone, i.e. it is unnecessary to check if Ψ is an improver with respect to a complexity function. Motivated by this reason, in the remainder of this paper, given $C \subseteq \mathcal{C}$ and a contraction $\Psi : C \to C$ we will say that Ψ is contractive improver (cont-improver for short) with respect to a complexity function $g \in C$ whenever $\Psi(g) \leq g$. Notice that an improver in our sense is an improver in the original sense of Schellekens. Moreover, the computational meaning of improver functionals remains valid for the cont-improver ones. Indeed, if Ψ is a cont-improver with respect to the complexity function g then $\Psi^n(g) \leq \Psi^{n-1}(g)$ for all $n \in \mathbb{N}$, since

$$d_{\mathcal{C}}|_C(\Psi^n(g), \Psi^{n-1}(g)) \leq \frac{1}{(2a)^{n-1}} d_{\mathcal{C}}|_C(\Psi(g), g) = 0$$

for all $n \in \mathbb{N}$.

Inspired by statement 2) in Lemma 6 we introduce a new kind of functionals that we call worseners. Let $C \subseteq \mathcal{C}$, a contraction $\Psi : C \to C$ is said to be a worsener with respect to a function $f \in C$ provided that $f \leq \Psi(f)$.

Observe that if Ψ is a worsener with respect to $f \in C$, then

$$d_{\mathcal{C}}|_C(\Psi^{n-1}(f), \Psi^n(f) \leq \frac{1}{(2a)^{n-1}} d_{\mathcal{C}}|_C(f, \Psi(f)) = 0$$

for all $n \in \mathbb{N}$. It follows that the computational meaning of a worsener functional is dual to the meaning of a cont-improver functional. In fact, a worsener is a functional which corresponds to a transformation on programs in such a way that the iterative applications of the transformation yield a worsened, from a complexity point of view, program at each step of the iteration.

In the next result we obtain the announced method to provide the complexity class of an algorithm whose running time of computing satisfies a recurrence equation of type (8).

Theorem 7. *Let $f_T \in \mathcal{C}_c$ be the (unique) solution to a recurrence equation of type (8). Then the following facts hold:*

1) *If the functional Ψ_T associated to (8), and given by (12), is a cont-improver with respect to some function $g \in \mathcal{C}_c$, then $f_T \in \mathcal{O}(g)$.*

2) *If the functional Ψ_T associated to (8), and given by (12), is a worsener with respect to some function $g \in \mathcal{C}_c$, then $f_T \in \Omega(g)$.*

Proof. Let $f_T \in \mathcal{C}_c$ be the (unique) solution to the recurrence equation (8). Assume that Ψ_T is an improver with respect to $g \in \mathcal{C}_c$. Then we have $\Psi_T(g) \leq g$. Hence we obtain that $d_{\mathcal{C}}|_{\mathcal{C}_c}(\Psi_T(g), g) = 0$. It immediately follows, by statement 1) in Lemma 6, that $d_{\mathcal{C}}|_{\mathcal{C}_c}(f_T, g) = 0$ and, thus, $f_T \leq g$. Consequently $f_T \in \mathcal{O}(g)$. So we have proved 1).

To prove 2) suppose that Ψ_T is a worsener with respect to $g \in \mathcal{C}_c$. Then $g \leq \Psi_T(g)$. Whence we deduce that $d_{\mathcal{C}}|_{\mathcal{C}_c}(g, \Psi_T(g)) = 0$. Thus statement 2) in Lemma 6 yields that $d_{\mathcal{C}}|_{\mathcal{C}_c}(f_T, g) = 0$. Hence $g \leq f_T$ and we conclude that $f_T \in \Omega(g)$. This finishes the proof.

■

Note that the solution to a recurrence equation of type (8) satisfies that $f_T \in \mathcal{O}(g) \cap \Omega(h)$ whenever Ψ_T is a cont-improver and a worsener with respect to $g \in \mathcal{C}_c$ and $h \in \mathcal{C}_c$, respectively. Consequently, Theorem 7 yields the complexity class of algorithms whose running time of computing satisfies a recurrence equation of type (8) when there exist $l \in \mathcal{C}_c$, $r, t > 0$ and $n_0 \in \mathbb{N}$

such that $g(n) = rl(n)$ and $h = tl(n)$ for all $n > n_0$ and, besides, Ψ_T is a cont-improver and a worsener with respect to g and h respectively, because, in such a case, $f_T \in \Theta(l)$.

3.3 Analyzing the running time computing of some algorithms

We end the paper showing that the developed method is useful to analyze the asymptotic complexity of Divide and Conquer algorithms, recursive algorithms and even non-recursive algorithms. To this aim we validate our results retrieving as an immediate consequence of Theorem 7 the well-known asymptotic complexity class of Quicksort (worst case), Hanoi and Largetwo (average case).

Quicksort: The running time of computing of Quicksort (worst case) is the solution to the recurrence equation

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n-1) + jn & \text{if } n \geq 2 \end{cases}, \tag{13}$$

where $c, j > 0$. It is clear that the preceding recurrence equation can be retrieved from (8) as a particular case when we fix $a = 1$ and $h(n) = jn$ for all $n \in \mathbb{N}$. Then, taking

$$\Psi_T(f)(n) = \begin{cases} c & \text{if } n = 1 \\ f(n-1) + jn & \text{if } n \geq 2 \end{cases} \tag{14}$$

for all $f \in \mathcal{C}_c$, Theorem 5 guarantees the existence and uniqueness of the solution (in $\mathcal{C}_c$), which matches up with the running time of computing of Quicksort (worst case), to the above recurrence equation. Denote such a solution by f_T^Q. It is not hard to see that Ψ_T is a cont-improver with respect to the complexity function $h_r \in \mathcal{C}_c$ (i.e. $\Psi_T(h_r) \leq h_r$) if and only if $r \geq \max\{\frac{3j}{5}, \frac{c}{4} + \frac{j}{2}\}$, where the complexity function h_r is given by

$$h_r(n) = \begin{cases} c & \text{if } n = 1 \\ rn^2 & \text{if } n \geq 2 \end{cases}$$

Hence we obtain, by statement 1) in Theorem 7, that the running time of Quicksort (worst case) holds $f_T^Q \in \mathcal{O}(h_{\max\{\frac{3j}{5}, \frac{c}{4}+\frac{j}{2}\}})$.

In addition, it is not hard to see that Ψ_T is a worsener with respect to the complexity function h_s (i.e. $h_s \leq \Psi_T(h_s)$) if and only if $s \leq \min\{\frac{j}{2}, \frac{c}{4} + \frac{j}{2}\}$, whence we deduce, by statement 2) in Theorem 7, that $f_T^Q \in \Omega(h_{\min\{d, \frac{2c+d}{3}\}})$.

Therefore we obtain that $f_T^H \in \mathcal{O}(h_{\max\{\frac{3j}{5}, \frac{c}{4}+\frac{j}{2}\}}) \cap \Omega(h_{\min\{\frac{j}{2}, \frac{c}{4}+\frac{j}{2}\}})$. Hence $f_T^Q \in \Theta(n^2)$, which is in accordance with the Quicksort (worst case) complexity class that can be found in the literature ([1], [2]).

Hanoi: The running time of computing of Hanoi is the solution, under the uniform cost criterion assumption, to the recurrence equation

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n-1) + d & \text{if } n \geq 2 \end{cases}, \tag{15}$$

where $c, d > 0$. It is clear that the preceding recurrence equation can be retrieved from (8) as a particular case when we fix $a = 2$ and $h(n) = d$ for all $n \in \mathbb{N}$. Then, taking

$$\Psi_T(f)(n) = \begin{cases} c & \text{if } n = 1 \\ 2f(n-1) + d & \text{if } n \geq 2 \end{cases} \tag{16}$$

for all $f \in \mathcal{C}_c$, Theorem 5 guarantees the existence and uniqueness of the solution (in $\mathcal{C}_c$), which matches up with the running time of computing of Hanoi, to the above recurrence equation. Next we denote such a solution by f_T^H. It is not hard to see that Ψ_T is a cont-improver with respect to the complexity function $h_r \in \mathcal{C}_c$ (i.e. $\Psi_T(h_r) \leq h_r$) if and only if $r \geq \max\{d, \frac{2c+d}{3}\}$, where the complexity function h_r is given by

$$h_r(n) = \begin{cases} c & \text{if } n = 1 \\ r\,(2^n - 1) & \text{if } n \geq 2 \end{cases}$$

So, by statement 1) in Theorem 7, we deduce that the running time of Hanoi satisfies $f_T^H \in \mathcal{O}(h_{\max\{d, \frac{2c+d}{3}\}})$.

Furthermore, it is easily seen that Ψ_T is a worsener with respect to the complexity function h_s (i.e. $h_s \leq \Psi_T(h_s)$) if and only if $s \leq \min\{d, \frac{2c+d}{3}\}$, whence we deduce, by statement 2) in Theorem 7, that $f_T^H \in \Omega(h_{\min\{d, \frac{2c+d}{3}\}})$.

Therefore we deduce hat $f_T^H \in \mathcal{O}(h_{\max\{d, \frac{2c+d}{3}\}}) \cap \Omega(h_{\min\{d, \frac{2c+d}{3}\}})$. Thus $f_T^H \in \Theta(2^n)$, which is in accordance with the Hanoi complexity class that can be found in the literature ([2]).

Largetwo: The running time of computing of Largetwo (average case) is the solution to the recurrence equation

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n-1) + 2 - \frac{1}{n} & \text{if } n \geq 2 \end{cases}, \tag{17}$$

where $c > 0$. It is clear that the preceding recurrence equation can be retrieved from (8) as a particular case when we fix $a = 1$ and $h(n) = 2 - \frac{1}{n}$ for all $n \in \mathbb{N}$. Then, taking

$$\Psi_T(f)(n) = \begin{cases} c & \text{if } n = 1 \\ f(n-1) + 2 - \frac{1}{n} & \text{if } n \geq 2 \end{cases} \tag{18}$$

for all $f \in \mathcal{C}_c$, Theorem 5 guarantees the existence and uniqueness of the solution (in $\mathcal{C}_c$), which matches up with the running time of computing of Largetwo (average case), to the above recurrence equation. Let us denote such a solution by f_T^L. It is not hard to see that Ψ_T is a cont-improver with respect to the complexity function $h_r \in \mathcal{C}_c$ (i.e. $\Psi_T(h_r) \leq h_r$) if and only if $r \geq \max\{\frac{5}{6+3\log_2(\frac{2}{3})}, \frac{2c+3}{2+2d}\}$, where the complexity function h_r is given by

$$h_r(n) = \begin{cases} c & \text{if } n = 1 \\ r\left(2\left(n-1\right) - \log_2 n + d\right) & \text{if } n \geq 2 \end{cases}$$

So we deduce, by statement 1) in Theorem 7, that the running time of Largetwo (average case) satisfies $f_T^L \in \mathcal{O}(h_{\max\{\frac{5}{6+3\log_2(\frac{2}{3})}, \frac{2c+3}{2+2d}\}})$.

Moreover, a straightforward computation shows that Ψ_T is a worsener with respect to the complexity function h_s (i.e. $h_s \leq \Psi_T(h_s)$) if and only if $s \leq \min\{1, \frac{2c+3}{2+2d}\}$, whence we deduce, by statement 2) in Theorem 7, that $f_T^L \in \Omega(h_{\min\{1, \frac{2c+3}{2+2d}\}})$.

Therefore we obtain hat $f_T^L \in \mathcal{O}(h_{\max\{\frac{5}{6+3\log_2(\frac{2}{3})}, \frac{2c+3}{2+2d}\}}) \cap \Omega(h_{\min\{1, \frac{2c+3}{2+2d}\}})$. Thus $f_T^L \in \Theta(n \log_2 n)$, which is in accordance with the Largetwo (average case) complexity class that can be found in the literature ([2]).

Acknowledgements

The authors thank the support from the Spanish Ministry of Science and Innovation, grant MTM2009-12872-C02-01.

References

[1] G. Brassard, P. Bratley, *Algorithms: Theory and Practice*, Prentice Hall, New Jersey, 1988.

[2] P. Cull, M. Flahive, R. Robson, *Difference equations: From rabbits to chaos*, Springer, New York, 2005.

[3] E.F. Ecklund Jr., P. Cull, *Towers of Hanoi and analysis of algorithms*, The American Mathematical Monthly **92** (1985), 407-420.

[4] H.P.A. Künzi, *Nonsymmetric distances and their associated topologies: About the origins of basic ideas in the area of asymmetric topology*, in: Handbook of the History of General Topology, ed. by C.E. Aull and R. Lowen, vol. 3, Kluwer, Dordrecht, 2001.

[5] S. Romaguera, M.P. Schellekens, *Quasi-metric properties of complexity spaces*, Topology Appl. **98** (1999), 311-322.

[6] M.P. Schellekens, *The Smyth completion: a common foundation for denonational semantics and complexity analysis*, Electron. Notes Theor. Comput. Sci. **1** (1995), 211-232.

[7] D.S. Scott, *Outline of a mathematical theory of computation*, in: Proc. 4th Annual Princeton Conference on Information Sciences and Systems, pp. 169-176 (1970).

[8] D.S. Scott, *Domains for denotational semantics*, LNCS **140** (1982), 577-613.

S. Romaguera and P. Tirado
Instituto Universitario de Matemática Pura y Aplicada
Universitat Politècnica de València
46022 Valencia, Spain
E-mail:sromague@mat.upv.es, pedtipe@mat.upv.es

O. Valero
Departamento de Ciencias Matemáticas e Informática
Universidad de las Islas Baleares
07111 Palma de Mallorca, Spain
E-mail: o.valero@uib.es