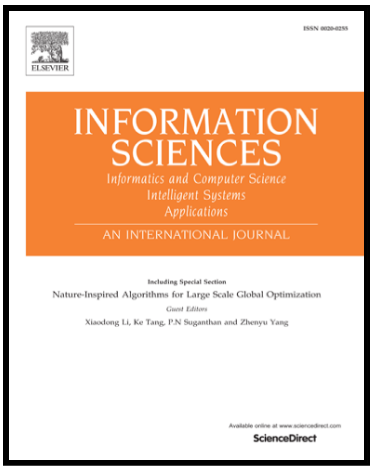
Accepted Manuscript



Computable Aggregations

Javier Montero, Ramón González-del-Campo, Luis Garmendia, Daniel Gómez, J. Tinguaro Rodríguez

PII: S0020-0255(17)31005-8
DOI: 10.1016/j.ins.2017.10.012
Reference: INS 13187

To appear in: *Information Sciences*

Received date: 12 April 2017
Revised date: 2 October 2017
Accepted date: 6 October 2017

Please cite this article as: Javier Montero, Ramón González-del-Campo, Luis Garmendia, Daniel Gómez, J. Tinguaro Rodríguez, Computable Aggregations, *Information Sciences* (2017), doi: 10.1016/j.ins.2017.10.012

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

ACCEPTED MANUSCRIPT

Computable Aggregations

Javier Montero[a,f], Ramón González-del-Campo[b,f], Luis Garmendia[c,f], Daniel Gómez[d,f], J. Tinguaro Rodríguez[e,f]

[a]*Faculty of Mathematics, email: monty@mat.ucm.es*
[b]*Faculty of Informatics, email: rgonzale@estad.ucm.es*
[c]*Faculty of Informatics, email: lgarmend@fdi.ucm.es*
[d]*Faculty of Statistics, email: dagomez@estad.ucm.es*
[e]*Faculty of Mathematics, email: jtrodrig@mat.ucm.es*
[f]*Complutense University of Madrid, Spain*

Abstract

In this paper, we postulate computation as a key element in assuring the consistency of a family of aggregation functions so that such a family of operators can be considered an aggregation rule. In particular, we suggest that the concept of an aggregation rule should be defined from a computational point of view, focusing on the computational properties of such an aggregation, i.e., on the manner in which the aggregation values are computed. The new algorithmic definition of aggregation we propose provides an operational approach to aggregation, one that is based upon lists of variable length and that produces a solution even when portions of data are inserted or deleted. Among other advantages, this approach allows the construction of different classifications of aggregation rules according to the programming paradigms used for their computation or according to their computational complexity.

Keywords:
Aggregation operator, aggregation function, aggregation rule.

1. Introduction

The standard definition of aggregation function (see [9]) assumes that our data set is not fixed. At least because it is possible that some expected observation might be lost or some unexpected observations might appear. Aggregation should be open to different kinds of data sets. In fact, from a scientific point of view, we should assure that once a specific methodology has been chosen to analyze those data, that methodology will be *consistently*

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

applied. For example, the arithmetic mean is a unique concept for aggregation, regardless of the number of real values to be summarized, and the median is similarly a clear aggregation concept regardless of the number of observations. However, using different operators for different numbers of aggregated items (e.g., the arithmetic mean when the number of items is even and the median when the number of items is odd) does not in general look very consistent, nor does randomly choosing the operator whenever a new data element arrives. Moreover, quite often it is required that such an aggregation methodology be defined in advance of the reception of data, as a guarantee to avoid arbitrariness in the aggregation process. However, we are all aware that in practice we cannot guarantee the cardinality of the data set we will finally obtain and that data should be first explored and most probably preprocessed as soon as they reach us.

The above computational issue has been emphasized by many authors. For example, in [11] it was pointed out that each ordered weighted aggregation (or OWA) operator, as defined in [32], can be considered only as long as the data set meets the essential OWA restriction on its cardinality. OWA's original definition did not offer any solution for proceeding when some value is lost or when some data unexpectedly appear. Aggregation by means of OWA operators implies updating weights, which requires a general formula or an algorithm to estimate these weights. A particular solution was the recursive approach proposed by Cutello and Montero, which was initially proposed within the OWA framework but was soon translated into a more general context [13]. In such a recursive approach, each aggregation depends on the previous aggregation, and the aggregation is obtained by means of a sequence of binary operators, whereas the classical approach under associativity assumes a unique binary operator along the whole aggregation process. The main results of this recursive approach were obtained in [1], taking advantage of previous results on the so-called *general associativity equation* [27] (see also [8]).

Recursivity was originally proposed to deal with the computational problem associated with any aggregation model. The key question was how to actually find or check a solution. This computational argument was also addressed in [12]. Furthermore, although recursivity is not the only option for approaching consistency in an aggregation function (see, e.g., [9]), the classical definition of an aggregation function refers to a sequence of aggregation operators that might be seen as arbitrary, with no apparent relationship connecting them. The way we reckon the first values or items to be aggregated

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

might be far from the way we reckon subsequent values. In fact, it can be proved that in some cases the way in which we aggregate the first values determines how we should aggregate the rest of the values (see [1]). In case such a sequence of aggregation operators depends on certain parameters, we should somehow assure that such a sequence of parameters produces a consistent aggregation function (i.e., an *aggregation rule*). This was the key argument in [11]: we need an automatic procedure to reshape weights when the number of aggregated items changes.

To assure consistency in a given sequence of aggregation operators, we need some kind of *transversal* property. However, such a transversal property assuring consistency is not unique. The above recursive approach, for example, comes from a computational argument and requires that each aggregated value be obtained from the previous aggregated value and the new item. The arithmetic mean is recursive in the sense of [13]. The median, however, is not recursive as keeping all previous data is always required. Alternative definitions for consistency have been proposed in the past, some of them assuring that a new value will not produce a drastically different result when a new item is added, at least when this new value has a value similar to the previous aggregated value. This alternative approach corresponds somehow to the notion of *stability* (see, e.g., [14] and [30], but also [10]). Such *stability* allows the estimation of missing values using previously aggregated values (see [19] and [5] for more details). Other conditions can be imposed to avoid discontinuities when data are somewhat similar (see, e.g., [23]).

In any event, we should be aware that not every family of aggregation operators defines an *aggregation rule*. Being under an aggregation rule means that, once data are consistent, the way to proceed is univocally defined, regardless of the cardinality of the data. Obviously, any commutative and associative binary operator defines a consistent aggregation rule. Some families of aggregation operators, however (see, e.g., [24] and [9]), are defined around some specific analytic property they share, and they do not define an aggregation rule unless they contain instructions for building the aggregation from the data. Moreover, even when such instructions are provided, such a family of operators might be perceived as inconsistent when they are applied.

In fact, the classical definition of an aggregation operator assumes that the cardinality of the data is given:

$$Ag_n : [0,1]^n \to [0,1]$$

where it is usually assumed that the aggregation operator Ag_n is mono-

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

tone, non-decreasing, and has boundary conditions $Ag_n(0, \ldots, 0) = 0$ and $Ag_n(1, \ldots, 1) = 1$.

Hence, an aggregation function has usually been defined as a family of aggregation operators that allows to solve aggregation with any possible cardinality of data [10]:

$$\{Ag_n : [0,1]^n \to [0,1]\}_{n=1}^{\infty}$$

each Ag_n being a monotone, non-decreasing aggregation operator with boundary conditions $Ag_n(0, \ldots, 0) = 0$ and $Ag_n(1, \ldots, 1) = 1$.

Such a definition of an aggregation function should be understood simply as a general framework: as in practice we cannot *a priori* assure the cardinality of the data, we need such a family of aggregation operators to manage data of any cardinality. As pointed out in [18], some standard conditions are not so natural and should not be imposed so readily. In particular, it is extremely restrictive to assume that the whole family of operators can be obtained by commutativity and associativity from a unique $Ag : [0,1]^2 \to [0,1]$ mapping (see also [28] for a first criticism of associativity in group decision making). However, we cannot accept that aggregation operators can drastically change with the number of data under aggregation. The fact is that the original definition of aggregation function [9] imposes the above monotonicity and boundary conditions for each aggregation operator plus an additional common-sense condition for $n = 1$ ($Ag_1(x) = x$ for all $x \in [0,1]$). However, no transversal condition is imposed in the sequence of aggregation operators. Such a definition is maintained in [6] even though the need to build models from practice is explicitly noted. This lack of a constructive approach appears to be a call to aggregation functions that are based upon a unique commutative and associative binary operator, or to those aggregation functions that can be given in terms of a compact mathematical formula, and does not address key computational issues. This formal approach is a serious limitation in practice, where a key issue is the determination of whether we will be able to evaluate the proposed aggregation index from the data. Moreover, it is interesting to note with [18] that even monotonicity, like commutativity, might be an excessive condition in some frameworks (e.g., not all means are monotonic, and data are not always permutable; see also [21]).

In this study, we develop the research initiated in [20], going back to the main computability argument underlying Cutello and Montero's recursivity [12, 13]. This issue was also highlighted in [29], where the authors proposed the term *aggregation rule* for those families of aggregation functions that

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

share some transversal condition to guarantee consistency of their aggregation operators. Aggregation in [29] was viewed from the point of view of its formal specification and implementation method. This approach allowed the definition of two kinds of aggregation rules (basic and non-basic). It also allowed the consideration of the functional and imperative paradigms of programming. A similar computational argument can be found in [4] and [15].

The main purpose of this work is to study families of aggregation operators that can be defined by means of an algorithm or a function that can solve all possible declared aggregations. Particular attention should be devoted to the possibility of partial reuse of previous computations when new values arrive. In general, we shall be interested on how we can produce an efficient computation of such an aggregation. This computational issue becomes critical in big data environments, where aggregations must be successfully computed by parts in a distributed parallel calculus in many nodes (mapping) and easily reduced in a multi-node Apache Hadoop system using the MapReduce programming paradigm. As standard aggregation operators being computed using SQL in relational databases (such as MAX, MIN, AVG, and SUM) must be maintained when data are modified, it is worth studying how they are actually computed. Whereas the recursive aggregation rules [1, 13, 18, 25] focus on the need for a recursive reckoning, here we will propose a purely computational approach to aggregation. Let us also point out that in some applications (for example, in image processing when we want to merge several images into one), operators are not formally defined as mappings; instead, we code a program that will solve the fusion problem.

The paper is organized as follows: in Section 2, we review key definitions as a basis for the computational approach proposed in Section 3. In Section 4, we analyze a basic classification of such computational aggregation rules that depends on their programming paradigms. In Section 5, we consider the classification of computational aggregation rules in terms of their computational complexity, pointing out the expected key role of our approach in big data. Some examples are provided, with attention given to both programming paradigms and computational complexity. The paper concludes with a final section highlighting the relevance of our computational approach to aggregation.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

2. Preliminaries

In this section, we shall review the key concepts needed to develop our proposal: the concept of computational complexity, the concept of list, and other standard concepts relevant to algorithms and programming. Let us first review the classical definition of aggregation operators.

Definition 2.1. *[6] An aggregation operator is a mapping $Ag : [0,1]^n \to [0,1]$ that satisfies*

1. $Ag(0,0,\ldots,0) = 0$ *and* $Ag(1,1,\ldots,1) = 1$.
2. *Ag is monotonic.*

This classical definition can be naturally extended to a more general class by replacing the unit interval $[0,1]$ with a more general lattice, which in the fuzzy field is traditionally assumed to be a complete lattice [17].

Definition 2.2. *Let $\mathcal{T}$ be a lattice with maximum $1_{\mathcal{T}}$ and minimum $0_{\mathcal{T}}$, and let $\mathcal{T}^n$ be the natural lattice of n elements of type $\mathcal{T}$ (i.e., $\mathcal{T}^n = \underbrace{\mathcal{T} \times \cdots \times \mathcal{T}}_{n\ times}$). A generalized aggregation operator is a mapping $Ag : \mathcal{T}^n \to \mathcal{T}$ such that it satisfies*

1. $Ag(\underbrace{0_{\mathcal{T}}, 0_{\mathcal{T}}, \ldots, 0_{\mathcal{T}}}_{n\ times}) = 0_{\mathcal{T}}$ *and* $Ag(\underbrace{1_{\mathcal{T}}, 1_{\mathcal{T}}, \ldots, 1_{\mathcal{T}}}_{n\ times}) = 1_{\mathcal{T}}$.
2. *Ag is monotonic with respect to the lattice's order.*

Let us note that although in this general definition of aggregation operator monotonicity has been kept, the relevant contribution of such a definition is that it presents the possibility of managing more general objects, for example, a grayscale digital photograph, understood as a matrix of $n \times m$ pixels in $[0,255]$. In this image processing framework, $[0]^{n\times m}$ is the white image and $[1]^{n\times m}$ is the black image. A color image is similar but with pixels in $[0,255]^3$.

As already pointed out above, we should focus on the means for finding each aggregation in practice, i.e., the procedure that will find the right aggregation in each case, which suggests the existence of an algorithm that defines our aggregation rule and the aggregation of each specific set of data.

The concept of computational complexity cost of an algorithm is a key issue in computer science as a measure of quality and usability of programs. We review classical notions using definitions 2.1 to 2.5 (see, e.g., [31]).

ACCEPTED MANUSCRIPT

Definition 2.3. *Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function. The set of functions in the order of f, $O(f)$, is defined as follows:*

$$O(f) \equiv \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}\ \forall n \geq n_0,\ g(n) \leq cf(n)\}$$

The order of f contains all the functions that grow more slowly than f.

Definition 2.4. *Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function. The computational complexity of f, $\Theta(f)$, is defined as follows:*

$$\Theta(f) \equiv \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c, d \in \mathbb{R}^+, n_0 \in \mathbb{N}\ \forall n \geq n_0,\ df(n) \leq g(n) \leq cf(n)\}$$

Definition 2.5. *Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be two functions. It is said that f has a lower complexity than g if $O(f) \subset O(g)$.*

Proposition 2.1. *Let q, a be two real numbers such that $q > 1$ and $a > 1$. Then*

$$O(1) \subset O(log(n)) \subset O(n) \subset O(n^q) \subset O(a^n) \subset O(n!)$$

In the following definition, the most common types of complexity are introduced.

Definition 2.6. *Let $g : \mathbb{N} \to \mathbb{R}^+$ be a function. Then*

- g has constant complexity if g belongs to $\Theta(1)$, i.e., if g grows as fast as $f(n) = 1$.
- g has logarithmic complexity if g belongs to $\Theta(log(n))$, i.e., if g grows as fast as $f(n) = log(n)$.
- g has linear complexity if g belongs to $\Theta(n)$, i.e., if g grows as fast as $f(n) = n$.
- g has polynomial complexity if g belongs to $\Theta(n^q)$ with $q > 1$, i.e., if g grows as fast as $f(n) = n^q$.
- g has exponential complexity if g belongs to $\Theta(a^n)$ with $a > 1$, i.e., if g grows as fast as $f(n) = a^n$.
- g has factorial complexity if g belongs to $\Theta(n!)$, i.e., if g grows as fast as $f(n) = n!$.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

Definition 2.7. *The computational complexity cost of an algorithm is the order of the function that gives the computing time of the algorithm.*

In addition to the above basic concepts in computational complexity, we also need to review what a *list* is.

Definition 2.8. *[3] A list L is an abstract data type (ADT) that represents a sequence of values. A list can be defined by its behavior, and its implementation must provide at least the following operations:*

- *Test whether a list is empty.*
- *Add a value.*
- *Remove a value.*
- *Compute the length of a list (the number of values in the list).*

A list can be defined under a template data type. For example, a list $L < [0,1] >$ is a list of values in $[0,1]$.

Now we review the concepts of algorithm, computer program, procedural programming, and declarative programming (see again [31]). These concepts will play a key role in the introduction of computational aggregation in the next section.

Definition 2.9. *In mathematics and computer science, an algorithm is a self-contained step-by-step set of operations to be performed.*

Definition 2.10. *A computer program, or simply a program, is a sequence of instructions written to perform a specified task on a computer.*

Definition 2.11. *A program written in procedural programming is a program that uses statements that change an algorithm's state. A statement is the smallest independent element that expresses some action to be carried out.*

Procedural programming is ordinarily a set of instructions written in a high-level language that commands the computer to perform a specified action.

Definition 2.12. *A program written in declarative programming is a program whose structure and elements express the logic of a computation without describing its control flow.*

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

Definition 2.13. *A program written in functional programming is a program that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.*

Definition 2.14. *A program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain.*

Functional programming and logic programming are particular cases of declarative programming.

Now that we have reviewed the above classical concepts of programming, algorithms, lists, and computational complexity, let us address our computational approach to aggregation.

3. Computable aggregations

The key idea of this paper is to focus on the way aggregations are actually made. Standard literature uses to explore nice properties that can be imposed on a mapping (or a set of mappings) so that they can be considered an aggregation. Hence, it is usually assumed that the formal definition of such a mapping is somehow given or that such a mapping will somehow be chosen. However, such a strictly mathematical view might be misleading. In the same way that a solution to a real problem usually comes before a theorem shows the framework in which such a solution works, and that the proof of such a theorem is usually obtained before the theorem statement itself, decision makers often face each problem with no mapping in mind, but rather with an intuition of some desired behavior. Decision makers gain insight into data from their knowledge and objectives, and then they start to grasp ways of summarizing and understanding the problem they are facing. Such an explorative process does not mean that a formal mapping has been defined. The way we normally learn is from particular cases, and then we might be able to guess a general formula representing the output of our reasoning process. The procedure typically comes prior to any definition of mapping, although with experience we are often able to provide such an underlying mapping soon after (quite often what happens is that we simply choose among some few formulae we are familiar with).

For example, most people are not conscious that $SUM(x_1; x_2; \ldots; x_n) = \sum_{i=1}^{n} x_i$ is not understood in their mind as a mapping, but as a procedure: "do $y = 0$, and then do $y = y + x_i$, from $i = 1$ to n". Sometimes we

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

simply have the reckoning procedure, but not the mapping. The existence of such a mapping might be assured if our procedure is consistent (i.e., each input always produces the same output, although we should be aware of the many uncertainties affecting the process). Obviously, we might not be able to list the complete mapping. Quite often, we do not try to list the complete mapping, perhaps because we know it will be not possible. Perhaps the only thing we want is to ensure that we will be able to calculate a few values, just the ones we need.

In fact, by restricting ourselves to those aggregations, we can actually avoid strange theoretical concerns. For example, from a purely mathematical point of view we can create mappings that are well defined as mappings, but that we shall never be able to determine (the existence of a value or mapping does not imply we can know or determine such a value or mapping). In practice, we often unconsciously assume that the aggregations $Ag : \mathcal{T}^n \to \mathcal{T}$ we are considering are "explicit" in the sense that there is a computable algorithm or program that allows the calculation of $Ag(x_1, \ldots, x_n) \in \mathcal{T}$ for each possible $(x_1, \ldots, x_n) \in \mathcal{T}^n$. Therefore, the key element in our aggregation should be the algorithm or program we have. Of course, the classical operators such as maximum, minimum, mean, mode, and median (and at least to some extent also OWA operators and the Choquet integral) that we use in practice are all computable; otherwise, we would not be able to use them in practice. They are computable aggregations in the sense that we can define an algorithm to calculate the aggregated value of any finite family of real values. We do not provide in the definition of those aggregations the outputs for all possible inputs; we simply provide a way to find those values when they are needed (notice, however, that the algorithm behind a mapping might not be unique, and that some algorithms might present specific advantages over other algorithms that produce the same aggregated values).

Our point here is that, in practice, we first design an evaluation procedure, and then (sometimes) we may be able to find the mapping (and as pointed out above, sometimes we even do not try to define the mapping and simply stay attached to the program we have). Our point here is that we should focus on how we actually reach solutions, instead of focusing on general analytical properties or general mathematical expressions, which usually come at the end of a long learning process. Most solutions to practical problems come from *ad hoc* procedures, quite often based upon sequential local intuitions. If we are lucky enough, those procedures can be made as compact as a universal formula, though this is not always the case. Hence, we should naturally focus

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

our study on ways that possible solutions can be consistently found. It is such a consistent procedure (the calculation instructions) that should be used to assure the final consistency of results, at least if the procedure is inspired by some unifying underlying concept. Once we know how to find a solution, we can ask ourselves how the goodness of the proposed solution can be checked. These are key issues in programming and the design of algorithms. Aggregation in practice should not be viewed as the application of any given formula but as the creation of a consistent procedure. And the main properties of an aggregation should be obtained from the properties of the program or algorithm that allows its implementation.

Let us, then, introduce our "computable aggregations".

Definition 3.1. *Let $L < T >$ be a list of n elements with type T. A computable aggregation is a program P that transforms the list $L < T >$ into an element of T.*

Remark 3.1. *Let us note that in general, it is not necessary to have an explicit function of the aggregation process to define a computable aggregation. Nevertheless, for the class of general aggregation operators, it is possible to define the computable aggregation associated with a generalized aggregation Ag as the pair (Ag, P) in the following way:*

Let $L < T >$ be a list of elements with type T. A computable aggregation associated with a generalized aggregation $Ag : L < T > \to T$ is a pair (Ag, P) such that $Ag(L) = P(L)$, where P is a program that verifies the generalized aggregation properties.

In addition, notice that we are not imposing any additional condition, such as monotonicity. The only thing we are imposing is that we have an implementable program for transforming a list of elements into a single element.

Therefore, computable aggregations can be classified by the computational complexity of the programs that compute them (linear, logarithmic, parabolic, etc.) or by their programming paradigm (iterative, recursive, rules, MapReduce, etc.), or even by program strategy (divide and conquer, backtracking, greedy, etc.)

Example 3.1. *The pair (Ag_{mean}, P_{mean}) is a computable aggregation associated with the aggregation function $Ag_{mean} : [0,1]^n \longrightarrow [0,1]$ defined as*

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

$Ag_{mean}(x_1, \ldots, x_n) = \frac{1}{n}\sum_{i=1}^{n} x_i$, where the program P_{mean} is defined as follows:

Let $L = (x_1, \ldots, x_n)$ be a list of values in $[0, 1]$. The P_{mean} program function written in C++ is

```
float arithmetic_mean(float L[], int n){
    float acum;
    int i;

    acum=0;
    if (n!=0){
        for(i=0;i++;i<n)
            acum=acum+L[i];
        return acum/n;
    };
}
```

The proposed concept of computable aggregation has been therefore defined linking an explicit or implicit expression of a "classical aggregation" to the existence of an algorithm that computes it. In fact, sometimes, the properties of an aggregation are only visible when aggregated values are computed. For example, as has already been pointed out, the idea of recursion is very closely related to the idea of how we can compute the value of an aggregation (recursiveness will always help us to compute aggregated values as we can take advantage of previous calculations). Anyway, our computational approach allows an understanding of "what" each aggregation is, as distinct from "how" the aggregation is made. In addition, it produces different natural classifications of aggregation procedures according to their associated programs or algorithms, as will be shown in the next two sections.

4. Types of computable aggregations by programming paradigm

Many classifications of aggregation functions have been developed that are based on the analytical properties of their operators, such as self-identity [33], stability [19, 30], migrativity [8], recursivity [13], and conjunctions/disjunctions [7, 16]. The concept of computable aggregation here proposed lends relevance to the way in which the aggregation process is computed and allows

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

alternative classifications of aggregation processes according to the way each aggregation is computed and programmed.

In this section, we propose a classification of computable aggregations by the programming paradigm that computes them, considering that the expression of each aggregation will be strongly related to the properties of its algorithm. For example, if an aggregation has a recursive expression, logic or functional programming paradigms might be more suitable for its implementation. In this section, some computable aggregations are classified by the programming paradigms usually applied for their computation. Note that computable aggregations can belong to two or more categories. For example, arithmetic means can be computed using a procedural, logical, or functional paradigm.

Definition 4.1. *Let $L < T >$ be a list of elements with type T and let P be a computable aggregation, $P : L < T > \to T$. We will say that the computable aggregation P is procedural when the program P is a procedural program.*

Example 4.1. *The program in Example 3.1 is a procedural program for computing the arithmetic mean.*

Definition 4.2. *Let $L < T >$ be a list of values in T and let P be a computable aggregation, $P : L < T > \to T$. We will say that the computable aggregation P is functional when the program P is a functional program.*

Example 4.2. *The pair $(Ag_{mean}, P_{mean-func})$ is a functional computable aggregation associated with the aggregation operator Ag_{mean}. Let X be a list of values in $[0, 1]$. The program $P_{mean-func}$ written in Haskell computes the arithmetic mean of n elements:*

```
sum [] = 0
sum (x:xs) = x + sum xs
long [] = 0
long (x:xs) = 1 + long xs
mean x = (sum x) / (long x)
```

Definition 4.3. *Let L be a list of values in $[0, 1]$ and let P be a computable aggregation operator, $P : L \to [0, 1]$. We will say that P is a logic computable aggregation when the program P is a logic program.*

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

Example 4.3. *The pair $(Ag_{mean}, P_{mean-logic})$ is a logic computable aggregation associated with the aggregation operator Ag_{mean}. Let L be a list of values in $[0,1]$. The program $P_{mean-logic}$ written in Prolog computes the arithmetic mean of n elements:*

```
add ([],0).
add ([L|Ls],S):- add(Ls,S1), S is 1 + S1+1.
long([],0).
long([C|L],Length):- longitud(L,L1),
                     Length is L1+1.
arithmetic_mean (L,A):- add(L,S),
                        long(L,Length),
                        A is S / Length.
```

From previous examples, we can see that the same aggregation operator Ag_{mean} could be considered a procedural, functional, or logic computable aggregation, depending on the way in which it is implemented.

5. Types of computable aggregations by computational complexity

One of the most important properties of an algorithm that computes a computable aggregation is its computing time. In this section, some computable aggregations are classified according to the complexity of the applied algorithm.

Definition 5.1. *A computable aggregation P has complexity $\Theta(t(n))$ if the program P presents this computational complexity.*

Definition 5.2. *A generalized aggregation operator Ag is approachable with complexity $\Theta(t(n))$ if there exists a computational aggregation P associated with Ag having computational complexity $\Theta(t(n))$.*

Proposition 5.1. *The aggregation operator Ag_{mean} is approachable with linear complexity.*

Proof. Trivial by Example 3.1. $\square$

Proposition 5.2. *The pair (Ag_{geo}, P_{geo}) where $Ag_{geo}(\overrightarrow{x}) = \sqrt[n]{x_1 x_2 \cdots x_n}$ and P_{geo} is defined in Haskell as*

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

```
    prod [] = 1
    prod (x:xs) = x * prod xs
    geometricmean l = sqrt (prod x)
```

is approachable with linear complexity.

Proof. Direct. □

Proposition 5.3. *The pair (Ag_{har}, P_{har}) where*

$$Ag_{har}(\overrightarrow{x}) = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}}$$

and P_{har} is defined in C++ as

```
    float harmonic_mean(float L[], int n){
        float acum;
        int i;

        acum=0;
        for(i=0;i++;i<n){
            acum=acum+1/L[i]
        };
        return n/acum;
    }
```

*is a computable aggregation with linear complexity. (*Note: *all values in* `L` *are assumed to be nonzero.)*

Proposition 5.4. *The pair (Ag_{min}, P_{min}) where $Ag_{min}(x) = min\{x_1, \ldots, x_n\}$ and P_{min} is defined in C++ as*

```
    float minimum(float L[], int n){
        float acum;
        if (n>0){
            acum=+infinitum;
        };
        for(i=0;i++;i<n){
            if(L[i]<acum){
                acum=L[i];
            }
```

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

```
        };
        return acum;
    }
```

is a computable aggregation with linear complexity.

In a similar way, we can see that it is possible to find a computable aggregation with linear complexity associated with the maximum and product aggregation operators Ag_{max} and Ag_{prod}.

Proposition 5.5. *Let F be the OWA operator defined by $F(a_1, \dots, a_n) = \sum_{k=1}^{n} w_k * b_k$, where b_j is the j-th largest element in $\{a_1, \dots, a_n\}$ and (w_k) is a list that satisfies $\sum_{k=1}^{n} w_k = 1$. Then F is approachable with complexity $\Theta(n * log(n))$.*

Proof. Taking into account that the sorting procedure of a set of items $\{a_1, \dots, a_n\}$ using an efficient algorithm (for example Quicksort) has $n*log(n)$ complexity and the computation of $\sum_{k=1}^{n} w_k * b_k$ has complexity n, it is very easy to find a computable aggregation P_{OWA} with complexity $n * log(n)$, and thus F will be approachable with complexity $\Theta(n * log(n))$. $\square$

Proposition 5.6. *The pair $(Ag_{Shapley}, P_{Shapley})$ is a computable aggregation approachable with complexity $\Theta(n2^n)$, where*

$$Ag_{Shapley}(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{\mid S \mid !(n - \mid S \mid -1)!}{n!} (v(S \cup \{i\}) - v(S))$$

and $P_{Shapley}$ is defined as follows:

```
float v(int S[],int n){
//Returns the value of v
    ...
}

int fact(int n){
//Returns the factorial of n

    int aux;

    if(n>1)
```

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

```
        aux=n*fact(n-1)
    else
        aux=1;
    return aux;
}

int card(int S[],int n){
//Returns the cardinality of S
    int aux=0;

    for(i=0;i++;i<n)
        if (S[i]>0)
            aux++;
    return aux;
}

next(int S[],int n, int i){
//Returns the next subset of S
    int k=n-1;

    while(S[k]>0||(k==i-1))
        k--;
    S[k]=1;
    while((k<n)&&(k!=i-1))
       S[k]=0;
}

float shapley(int n, int i){
//Help: the element "i" is in S if and only if S[i-1]=1

    int aux, aux2, aux3, aux4;
    float v1, v2;
    int S[n];                //Subsets of S
    int acum=0;

    for(i=0;i++;n)
        S[i]=0; // S is the empty set
    for(i=0;i++;pow(2,n-1)){
```

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

```
        aux=card(S);        //aux contains |S|
        aux2=fact(aux);     //aux2 contains |S|!
        aux3=fact(n-aux-1); //aux3 contains (n-|S|-1)!
        aux4=fact(n);       //aux4 contains n!
        v1=v(S);
        S[i-1]=1;           //S is the union of S and {i}
        v2=v(S);
        acum=+ (aux2*aux3/aux4)*(v2-v1);
        next(S,n,i);
    };
    return acum;
}
```

Proof. It is straightforward to check as since each loop has complexity $\Theta(n)$ and there exist 2^{n-1} loops, this computable aggregation operator is approachable with complexity $\Theta(n2^n)$. □

Finally, in the next subsection, we explore the natural link of our computational approach to big data, where the concept of computational complexity is also essential.

5.1. Computable aggregations and big data

The emergence of the big data approach in the early years of this century changed traditional data warehouse environments and programming paradigms, leading to the introduction of the MapReduce paradigm in (now classical) environments such as Apache Hadoop, which can work on several servers or nodes. Data are automatically distributed in Hadoop Distributed File System (HDFS) files on every node of the cluster, where the maps are run; after processing on each node, the information from all nodes is then reduced into one decision node, which provides the required output. When the computational complexity of the processing step is high or the quantity of data is very large, big data technologies allow the distribution of data and computations, shortening and bounding the response times of the intended information management tasks.

As aggregation processes are common and essential tools to many information management procedures, including many of the current big data applications, we think the computational approach to aggregation here proposed may be helpful to these developing technologies, as it allows an under-

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

standing of such essential aggregation steps in terms of relevant categories such as computational complexity and scalability.

Therefore, to introduce this aspect of our point, let us first review some concepts typically encountered in the context of big data information processing, focusing in particular on the notion of map-reducibility.

Definition 5.3. *A computational aggregation P is map-reducible if the program P has been programmed using the MapReduce programming paradigm.*

Using this definition, we can naturally classify an aggregation function as map-reducible or non-map-reducible according to the existence of a computational aggregation associated with a program verifying such conditions.

Definition 5.4. *A generalized aggregation Ag is approachable in a map-reducible way if there exists a computational aggregation P associated with Ag that is map-reducible.*

Example 5.1. *The arithmetic mean is approachable in a map-reducible way, as the sums of the inputs can be spread across many nodes and then reduced into one node.*

Example 5.2. *The median is not approachable in a map-reducible way because, as all of the inputs are needed to compute it, it cannot be separately computed from smaller distributed portions of data.*

Example 5.3. *An example with Scala (a functional programming language in an Apache Spark shell on an HDFS file distributed in a Hadoop environment) to compute the arithmetic mean using the MapReduce paradigm is the following:*

```
val data = sc.textFile(hdfs://file.csv)
data.map(row => (row, (row.data, 1)))
.reduceByKey(_ |+| _)
.mapValues { case (total, count) =>
  total.toDouble / count
}
.collect()
```

As a summary of some of the ideas discussed in this paper, Table 1 shows the classification of some well-known aggregation functions from the point of view of their computational complexity and of their scalability. Let us point

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

Table 1: Classifications of some well-known aggregation functions in terms of their complexity and scalability.

Aggregation Operator	Computable?	Complexity	Map-Reducible?
$Ag_{max} = Max\{x_1, \ldots, x_n\}$	YES	$\Theta(n)$	YES
$Ag_{min} = Min\{x_1, \ldots, x_n\}$	YES	$\Theta(n)$	YES
$Ag_{mean} = \sum_{i=1}^{n} \frac{1}{n} x_i$	YES	$\Theta(n)$	YES
Ag_{median}	YES	$\Theta(nlog(n))$	NO
Ag_{OWA}	YES	$\Theta(nlog(n))$	NO
$Ag_{Shapley}$	YES	$\Theta(n2^n)$	YES
$H_n = n/(\sum_{i=1}^{n}(1/x_i))$	YES	$\Theta(n)$	YES
$Q_n = \prod_{i=1}^{n} x_i^i$	YES	$\Theta(n)$	YES
$P_n = \prod_{i=1}^{n} x_i$	YES	$\Theta(n)$	YES
$A_n^f = A_n^f(x_1, \ldots, x_n)$	YES	$\Theta(n * c(n))$	NO
$A_n^b = A_n^b(x_1, \ldots, x_n)$	YES	$\Theta(n * c(n))$	NO

out that, in this table, A_n^f (respectively A_n^b) denotes the usual forward (respectively backward) aggregation function, and, similarly, $c(n)$ represents the complexity of the binary operator associated with these A_n^f and A_n^b aggregations. Obviously, every aggregation rule built from the successive application of a unique commutative and associative binary operator will be computable, and its complexity will be linearly related to the complexity of that binary operator.

6. Conclusions

The main proposal in this paper is to view aggregation from a strictly computational approach. The concept of aggregation is in this way defined using as input a list of data values from a template, which can take different formats, from values within the unit interval to images or any kind of heterogeneous spaces. Similarly, output can be within the unit interval or any kind of informative multidimensional summarization. Future research can therefore explore theoretical and practical considerations of recent works on linguistic representation, for example those of [2] and [22].

In particular, we have emphasized how an aggregation process usually starts in practice from a reckoning procedure, which can be given in terms of a program or algorithm. It is this procedure, if consistent, that will assure the unity of concept of the operators that is desired to be considered

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

an aggregation rule. The successive application of a commutative and associative binary operator indeed assures consistency in the aggregation of any finite data set, but outside this case, we need to ensure that the different operators we are going to use represent the same aggregation concept. By analyzing how aggregated values are obtained in practice, we have also shown how these computational aggregations can be classified from the point of view of their programming paradigms or by the computational complexity of the algorithms that implement them. Any computational argument can be translated into aggregation rules under this approach, where the key issue is how the aggregation is obtained.

Moreover, this computational approach seems the natural aggregation approach in any framework in which computational complexity is an essential feature. In big data, aggregation needs to be considered from a computational point of view, restricting approaches to those that can be practically implemented. The formal properties of an aggregation can then be related to the construction of its algorithms, and improving its computational efficiency means improving the aggregation procedure. It is the available algorithm (in general not unique) that represents the key property of an aggregation procedure, and no aggregation procedure is implementable unless a (preferably efficient) algorithm to produce the aggregated values is available. This computational approach to aggregation implies a kind of consistent definition of the whole aggregation procedure, but, of course, *consistency* is not univocally defined. Future studies should also search for alternative notions of consistency, ones that might depend on additional computational and storing limitations as well as on the interests and capabilities of users (a decision maker or another machine). Our procedures should be implementable and, because of that, useful.

Acknowledgment

This research has been partially supported by the Government of Spain (grant TIN2015-66471-P), the Government of Madrid (grant S2013/ICCE-2845), and Complutense University (UCM Research Group 910149).

References

[1] A. del Amo, J. Montero, and E. Molina. Representation of consistent recursive rules. *European Journal of Operational Research* 130:29–53, 2001.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

[2] K. Atanassov, I. Georgiev, E. Szmidt, J. Kacprzyk. Multidimensional intuitionistic fuzzy quantifiers. Proceedings of the IEEE International Conference on Intelligent Systems (IS'16), Sofia, Bulgaria, 2016, pp. 530-534

[3] G. Barnett, L. Del Tonga. *Data Structures and Algorithms.* DotNet-Slackers, 2008.

[4] G. Beliakov, J. Dujmovic. Extension of bivariate means to weighted means of several arguments by using binary trees. *Information Sciences* 331:137–147, 2016.

[5] G. Beliakov, D. Gómez, S. James, J. Montero, J.T. Rodríguez. Approaches to learning strictly-stable weights for data missing values. *Fuzzy Sets and Systems*, in press, http://doi.org/10.1016/j.fss.2017.02.003.

[6] G. Beliakov, A. Pradera, T. Calvo. *Aggregations Functions: A guide for Practitioners.* Springer, 2007.

[7] U. Bentkowska, A. Król. Preservation of fuzzy relation properties based on fuzzy conjunctions and disjunctions during aggregation process. *Fuzzy Sets and Systems* 291:98–113, 2016.

[8] H. Bustince, B. de Baets, J. Fernández, R. Mesiar, J. Montero. A generalization of the migrativity property of aggregation functions. *Information Sciences* 191:76–85, 2012.

[9] T. Calvo, A. Kolesárová, M. Komorníková, R. Mesiar. Aggregation operators: properties, classes and constructions methods. *Studies in Fuzziness and Soft Computing* 97:3–104, 2002.

[10] T. Calvo, R. Mesiar. Stability of aggregation operators. Proceedings of the European Society for Fuzzy Logic and Technologies Conference (EUSFLAT'01), Leicester, U.K., pp. 475–478, 2001.

[11] V. Cutello, J. Montero. Recursive families of OWA operators. *Proceedings of the IEEE International Conference on Fuzzy Systems (FUZZIEEE 1994)*, Orlando, USA, 1994, pp. 1137–1141.

[12] V. Cutello, J. Montero. Hierarchical aggregation of OWA operators: basic measures and related computational problems. *Uncertainty, Fuzziness and Knowledge-Based Systems* 3:17–26, 1995.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

[13] V. Cutello, J. Montero. Recursive connective rules. *International Journal of Intelligent Systems* 14:3–20, 1999.

[14] J. Dujmovic. Aggregation operators and observable properties of human reasoning. Advances in Intelligent Systems and Computing 228:5–16, 2013. .

[15] J. Dujmovic. An efficient algorithm for general weighted aggregation. *Proceedings of the International Summer School on Aggregation Operators (AGOP'15)*, Katowice, Poland, 2015, pp. 115–120.

[16] J. Dujmovic, H.L. Larsen. Generalized conjunction/disjunction. *International Journal of Approximate Reasoning* 46:423-446, 2017.

[17] J.A. Goguen L-fuzzy sets. *Journal on Mathematical Analysis and Applications* 18:145–174, 1967.

[18] D. Gómez and J. Montero. A discussion on aggregations operators. *Kybernetika*, 40:107–120, 2004.

[19] D. Gómez, K. Rojas, J. Montero, J.T. Rodríguez, G. Beliakov. Consistency and stability in aggregation operators, an application to missing data problems *International Journal of Computational Intelligence Systems* 7:595–604, 2014.

[20] R. González del Campo, L. Garmendia, J. Montero. Expansible Computable Aggregation Rules. *Proceedings of the International Conference on Intelligent Systems and Knowledge Engineering (ISKE'15)*, Taipei, Taiwan, 2015, pp. 8–11.

[21] M. Grabisch, J.L. Marichal, R. Mesiar, E. Pap. Aggregation functions: means. *Information Sciences* 181:1–22, 2011.

[22] J. Kacprzyk, S. Zadrony. Linguistic summarization of the contents of web server logs via the ordered weighted averaging (owa) operators. Fuzzy Sets and Systems 285:182–198, 2016.

[23] E.P. Klement, A. Kolesárová. 1-Lipschitz aggregation operators and quasi-copulas. *Kybernetika* 39:615–629, 2003.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

[24] E.P. Klement, R. Mesiar, E. Pap. A Universal Integral as Common Frame for Choquet and Sugeno Integral. IEEE Transactions on Fuzzy Systems 18:178–187, 2010.

[25] A. Kolesárová, R. Mesiar, J. Montero. Sequential aggregation of bags. *Information Sciences* 294:305–314, 2015.

[26] V. López, J. Montero, J.T. Rodríguez. Formal specification and implementation of computational aggregation functions. *Computing Engineering and Information Science* 4:523-528, 2010.

[27] K. T. Mak. Coherent continuous systems and the generalized functional equation of associativity. *Mathematics of Operations Research* 12:597–625, 1987.

[28] J. Montero. A note on Fung-Fu's theorem. *Fuzzy Sets and Systems* 13:259–269, 1985.

[29] J.T. Rodríguez, V. López, D. Gómez, B. Vitoriano, J. Montero. A computational definition of aggregation rules. *Proceedings of the IEEE International Conference on Fuzzy Systems (FUZZIEEE'10)*, Barcelona, Spain, 2010, pp. 1–5.

[30] K. Rojas, D. Gómez, J. Montero, J.T. Rodríguez. Strictly stable families of aggregation operators. *Fuzzy Sets and Systems* 228:44–63, 2013.

[31] C. H. Papadimitriou. *Computational complexity.* Addison Wesley Longman, 1994.

[32] R.R. Yager. Families of OWA operators. *Fuzzy Sets and Systems* 59:125–148, 1993.

[33] R.R. Yager, A. Rybalov. Noncommutative self-identity aggregation. *Fuzzy Sets and Systems* 85:73–82, 1997.

ACCEPTED MANUSCRIPT