Complex Problems: Granularity is Necessary, Granularity Helps

Oscar N. Garcia[1], Vladik Kreinovich[2], Luc Longpré[2], and Hung T. Nguyen[3]

[1]Department of Computer Science and Engineering,
Wright State University, Dayton, OH 45435, USA, email ogarcia@cs.wright.edu

[2]Department of Computer Science, University of Texas at El Paso
El Paso, TX 79968, USA, emails {vladik,longpre}@cs.utep.edu

[3]Department of Mathematical Sciences, New Mexico State University
Las Cruces, NM 88003, USA, email hunguyen@nmsu.edu

1 Introduction

Many real-life problems, when formulated in precise mathematical terms, become computationally intractable (NP-hard; for exact definitions, see, e.g., [3, 6]). For example, a simple hit-and-miss problem (to hit an object while avoiding hitting several others, see, e.g., [4]) which is important for military, medical, manufacturing, and other applications, can be proven to be NP-hard.

In some cases, experts agree that the problem itself is complex and difficult to solve. However, in many other situations, although the general problem is complex from a theoretical viewpoint, in practice, experts successfully solve its instances.

One reason for this seeming contradiction was emphasized by L. Zadeh (see, e.g., [11]):

- if we formulate the problem in too much detail, then the problem may indeed look very complex, and a general class of such detailed problems is indeed intractable;
- however, in many practical applications, we can go to a higher level of abstraction, to a higher level of granularity, and on this level of granularity, the problem becomes much easier to solve.

In this paper, we prove several results aimed at the mathematical formalization of this idea. These results deal with the classical problem which was historically the first to be proven to be NP-hard: satisfiability problem for formulas of (classical) propositional logic. We prove the following two results:

- first, for each instance of the general satisfiability problem, there is a proper level of abstraction which makes this particular instance easy to solve;
- however, finding this "proper level of abstraction" is, in general, an computationally intractable problem (i.e., expert's knowledge is really needed to make it efficient).

These results prompt the following heuristic for solving satisfiability problems: instead of trying to solve this problem directly, we look for an appropriate abstraction (granularity) level, and then try to solve the corresponding higher-level problem. We show that:

- this heuristic does indeed lead to an efficient algorithm (it was partially outlined in [8]), and that
- to choose the best algorithmic implementation of this idea, one can successfully use a general symmetry-group approach developed by two of the authors (V.K. and H.T.N.) in their recent monograph [9] (this approach also explains the most frequently used fuzzy and neural heuristics).

2 Many important practical problems are complex

Some problems are difficult to solve: a brief reminder. Some real-life problems are *feasible* in the sense that there exists an algorithm which solves all instances of this general problem in reasonable time (to be more precise, in *polynomial time*, i.e., in time which is bounded by a polynomial of the length of the input). Many real-life problems, however, when formulated in precise mathematical terms, become computationally intractable (NP-hard).

Crudely speaking, NP-hard means that if we are able to solve all instances of this general problem in reasonable time, then we will be able to produce an algorithm which solves *all* general problems (from a certain well-defined class NP) in polynomial time (for exact definitions, see, e.g., [3, 6]). Since most computer scientists believe that such a general algorithm is impossible, NP-hardness means that no reasonable-time algorithm can solve all the instances of the given general problem.

NP-hardness is not only a negative result. At first glance, once we proved that a problem is NP-hard, all we got was a *negative* result. However, NP-hardness has its *positive* side too (see, e.g., [3, 6, 10]): once we have proven that a certain general problem is NP-hard, this means that the solution of any other problem from the above-mentioned class NP can be reduced to solving an appropriate instance of this particular NP-hard problem. Thus, it is not necessary to develop, for each complicated problem, its own heuristic algorithm: it is sufficient to have a good heuristic algorithm for *one* NP-hard problem, and this will help us solve all other complicated (NP-hard) problems.

We mentioned that it is sufficient to concentrate on a *single* NP-hard general problem. Which NP-hard problem should we concentrate on? It is natural to use the NP-hard problem which have been studied for the longest period of time and which, therefore, have accumulated the largest amount of heuristics: namely, on the so-called *propositional satisfiability* problem.

Propositional satisfiability: historically the first example of an NP-hard problem. Historically the first problem proven to be NP-hard was the so called *propositional satisfiability problem.* In this problem, we start with n Boolean (propositional) variables $x_1, \ldots, x_n$, i.e., variables which can only take values "true" (= 1) and "false" (= 0). We can combine these variables into *propositional formulas* by using the logical connectives "and" (&), "or" ($\vee$), and "not" ($\neg$). A formula is *satisfiable* if there exist values v_i of the variables x_i for which the resulting formula F is true. Now, we can formulate the *satisfiability problem*: given a propositional formula F, check whether this formula is satisfiable.

This problem is known to be NP-hard. Moreover, it is known (see, e.g., [3]) that this general problem remains NP-hard if instead of *arbitrary* propositional formulas, we consider only propositional formulas in the *conjunctive normal form* (CNF), i.e., only propositional formulas of the type $D_1 \,\&\, \ldots \,\&\, D_m$, where each of D_j is a *disjunction*, i.e., a formula of the type $a_1 \vee \ldots \vee a_p$, and each a_k is a *literal*, i.e., either a variable x_i, or its negation $\neg x_k$.

A related NP-hard problem: actually finding the satisfying vector. It is known that the following *related problem* is NP-hard: given a satisfiable CNF formula F, find the values v_i which make this formula true. The fact that this new problem is NP-hard is reasonably easy to prove: Indeed, we can show that if we had a reasonable-time algorithm U for solving this problem, then, by using U as a building block, we would be able to design a new reasonable-time algorithm U' which solves the original satisfiability problem.

As we have just mentioned, in our new algorithm U', we will be using the algorithm U as a building block. In the related problem (solved by the algorithm U), the input is always a satisfying formula. We want our new algorithm U' to be applicable to *arbitrary* formulas, not necessarily satisfying ones, because the whole purpose of the new algorithm is to check whether a given formula is satisfiable. Thus, we are planning to apply U to formulas which may not be satisfiable.

Since the algorithm U is a reasonable-time algorithm, this means that there is a reasonable upper bound $t(n)$ on time that it takes for U to work on satisfiable formulas of length n. This upper bound does not necessarily mean that U cannot go on for a longer time (or even loop infinitely long) when applied to non-satisfiable formulas. To prevent such a waste of computer time, we will modify U slightly: namely, a modified algorithm U^* first reads the formula F, measures its length n in bits, calculates $t(n)$, and then sets an alarm on the time $t(n)$. If the alarm clock gets activated before the computations are over, we simply stop the computations. (Since for satisfiable formulas, computations stop before the time $t(n)$, this modification does not affect the ability of the algorithm U to produce satisfying vectors.)

Let us now describe the desired algorithm U'. Given a formula F:

- First, we apply the modified algorithm U^* to the given formula F.
- Then:
  - If the algorithm U^* does not produce a Boolean vector at all, we return the answer "F is not satisfiable" and stop.
  - If the algorithm U^* does return Boolean vector, we substitute the corresponding values into the formula F. Then:
    - If the result is "true", i.e., if the values $v_1, \ldots, v_n$ produced by the algorithm U^* satisfy the formula F, then we can conclude that F is satisfiable.
    - Otherwise, if the values v_i do not make F true, we conclude that the formula F is not satisfiable.

It is easy to check that the answer produced by this new algorithm U' is indeed correct:

- If the vector v_i satisfies F, then F is satisfiable by definition of satisfiability.
- If there is no vector produced, or if the produced vector does not satisfy the formula F, then the formula is not satisfiable, because otherwise, the algorithm U would have produced the satisfying vector in time $\le t(n)$ (and hence, the modified algorithm U^* would have produced the exact same satisfying vector).

An example of a practical NP-hard problem: hit-and-miss problem. Propositional satisfiability is an example of a rather artificial mathematical problem. It does occur in practice (e.g., in designing digital circuits), but such direct applications are rare. A much more frequent practical situation is when a practical problem is formulated in a different language and then reformulated as an equivalent satisfiability problem.

As an example of such a practical NP-hard problem, we can take the "hit-and-miss" problem (see, e.g., [4]): hit a given object (in a real or imaginary space) while trying not to his several other given objects. This problem has many important applications:

- in *military* applications, we must direct a military strike (e.g., a missile, a torpedo, or a bomb) so that it would hit the desired military target and at the same time avoid both our positions and civilian objects;
- in *medical* applications, we want to direct the X-rays, ultrasound, or medicines in such a way that the target (e.g., a malignant tumor) is hit, while the surrounding healthy tissue is not;
- in *manufacturing* applications, we, e.g., want to paint a certain part of the car without spreading the paint over the other parts, etc.

Let us show that this problem is indeed NP-hard. (This result is rather straightforward, but since we could not find its exact formulation, we decided to present it here, for the sake of completeness.)

Often, this problem is considered in real geometric space, 2-D or 3-D spaces, but we can also consider the problem in an artificial multi-D space. For example, when we want to avoid hitting a tissue with an ultrasound, then:

- At first glance, it is sufficient to consider a simple 3-D geometric miss statement.
- However, the 3-D geometric miss problem is an oversimplification.

In reality, e.g., it may be OK to pass some ultrasound through the healthy tissue if its frequency f or its amplitude A is within certain limits. To describe a more realistic problem, we must describe this tissue not as a set in a 3-D space of actual spatial coordinates (x_1, x_2, x_3), but rather as a set in a 5-D space (x_1, x_2, x_3, f, A) whose components describe both a point location (x_1, x_2, x_3) and the characteristics (f, A) of the ultrasound which may be damaging to the tissue at this location. Then, the corresponding 5-D hit-and-miss problem is much more realistic.

In view of this comment, we will consider a general hit-and-miss problem in an n-dimensional space of n-dimensional points $x = (x_1, \ldots, x_n)$.

In any computer, there are only finitely many possible states, so we can represent only finitely many possible values of each of the coordinates x_i. We will show that the hit-and-miss problem is NP-hard even in the simplest case, when we only allow *two* different values of each coordinate. If we only have two values, then we need only one bit (with values 0 or 1) to represent each of these values. Thus, without losing generality, we can assume that each coordinate x_i can take exactly two values: 0 and 1. In this case, the entire universe is the set $\{0,1\}^n$ of Boolean vectors $x = (x_1, \ldots, x_n)$ $(x_i \in \{0,1\})$ of length n.

How can we describe the areas (sets) which we want to hit or miss? From the mathematical viewpoint, *sets* $S \in \{0,1\}^n$ are in 1-1 correspondence with *predicates* $P(x)$:

- to each set S, we can put into correspondence a predicate $x \in S$ which is true if and only if the element x belongs to the set S;
- vice versa, to each predicate $P(x)$ defined on $\{0,1\}^n$, we can put into correspondence the set $S = \{x \mid P(x)\}$ of all the vectors x which satisfy the property P.

Thus, describing a set is equivalent to describing the corresponding predicate. In logic, there is a traditional way of describing predicates of arbitrary complexity:

- we start with *elementary formulas*; e.g., in the binary (Boolean) case, when the coordinates themselves take only two values and can, therefore, be interpreted as "true" and "false", we can start with formulas x_i which simply coincide with these coordinates;
- then, we combine the elementary formulas by using the *logical connectives*, and get the so-called *propositional formulas*;
- finally, we can also use *quantifiers* $\forall x$, $\exists x$ and get the most general formulas; to be more precise:
  - if we only use quantifiers over *objects*, we get *first-order formulas*;
  - if we use quantifiers over *properties* as well, we get the so-called *second order* formulas.

We will show that the hit-and-miss problem is NP-hard even if we only consider the simplest case of propositional formulas.

We are now ready for a precise result:

Definition 1.

- *By an instance P of a hit-and-miss problem, we mean the tuple $\langle n, m, H, M_1, \ldots, M_m \rangle$, where n and m are positive integers, and H and M_i are propositional formulas of n variables.*
- *Let an instance $P = \langle n, m, H, M_1, \ldots, M_m \rangle$ be given.*
  - *We say that a Boolean vector x hits a set H if $x \in H$.*
  - *We say that a Boolean vector x misses a set M_i if $x \notin M_i$.*
  - *We say that a Boolean vector is a solution to the instance P if it hits the set H and misses all the sets $M_1, \ldots, M_m$.*
- *By a hit-and-miss problem, we mean the following problem: given an instance P, find a solution x.*

Proposition 1. *Hit-and-miss problem is NP-hard.*

For reader's convenience, all the proofs are placed in the last section.

3 Granularity is necessary

In many practical situations, we face the following phenomenon:

- from a purely *theoretical* viewpoint, the corresponding problem is very complex, *unsolvable* by a reasonable time algorithm;
- however, in *practice*, experts *successfully solve* many instances of this problem.

One reason for this seeming contradiction was emphasized by L. Zadeh (see, e.g., [11]):

- if we formulate the problem in too much detail, then the problem may indeed look very complex, and a general class of such detailed problems is indeed intractable;
- however, in many practical applications, we can go to a higher level of abstraction, to a higher level of granularity, and on this level of granularity, the problem becomes much easier to solve.

In this paper, we prove several results aimed at the mathematical formalization of this idea.

4 Granularity is useful

In this section, we will show, on the example of propositional satisfiability problem for CNF formulas, that for every instance of a general problem, there is a proper level of abstraction which makes this particular instance easy to solve. Since, as we mentioned, the satisfiability problem is NP-hard (i.e., universal), and it is arguably the most well-studied of all such universal problems, this result shows that granularity can indeed help.

How we can define granularity for propositional formulas. Before we formulate our result, let us describe what is the natural meaning of granularity levels for satisfiability problems. Propositional variables x_i, which take the values "true" and "false", can be thus viewed as elementary properties of an analyzed object.

> For example, in a black-and-white image, each value x_i may describe whether the i-th pixel is white or not.

The idea of going to a next level of abstraction means that instead of considering only individual properties (or, in the image example, individual pixels), we also consider more abstract properties, i.e., *combinations* of elementary properties. For example, we may divide all original vectors into several classes $S_1, \ldots, S_c$.

> E.g., in case of black-and-white images, we can divide the images into images in which most pixels are black, and images in which most pixels are white.

Each of these classes S_i is, by itself, described by a combination of elementary formulas, i.e., by a *propositional formula* F_i. The fact that we are adding these new notion means that we are, in effect, introducing, for each of the new classes, a new variable f_i, with an additional formula $f_i \leftrightarrow F_i$ (i.e., in terms of &, $\vee$, and $\neg$, $(f_i \& F_i) \vee (\neg f_i \& \neg F_i)$) which serves as a *definition* of the new notion. The fact that each vector belongs to one of c classes can be described by a formula $f_1 \vee \ldots \vee f_c$. Thus, instead of the original formula F, we consider a new formula

$$F \& (f_1 \vee \ldots \vee f_c) \& (f_1 \leftrightarrow F_1) \& \ldots \& (f_c \leftrightarrow F_c). \quad (1)$$

Such a method was considered, e.g., in [1, 2, 5]; different versions of this method are known as *hypersplitting* and *hyperresolution*.

In this section, we will show that for each instance F of the above-described related propositional satisfiability problem, there exists an appropriate granularity level $\{F_1, \ldots, F_c\}$ for which the equivalent problem (1) is easy to solve.

Definition 2. *Let n be a positive integer.*

- *By a granularity level L, we mean a finite list of propositional formulas $F_1, \ldots, F_c$ with n variables $x_1, \ldots, x_n$. We will also call L an abstraction level.*
- *For every propositional formula F, and for every granularity level $L = \langle F_1, \ldots, F_c \rangle$, by a reformulation of the formula F on an abstraction level L, we mean the formula* (1).

We will show that by choosing an appropriate abstraction level, we can always reduce a problem to a problem from a simply solvable class:

Definition 3. *Let U be a polynomial time algorithm, and let F be a satisfiable propositional formula. We say that a formula F is U-simple if the algorithm U produces a satisfying vector for this formula.*

(Checking whether a vector is satisfying is a straightforward (and thus, fast) task.)

Proposition 2. *There exists a polynomial-time algorithm U for which, for every satisfiable propositional formula F, there exists an abstraction level L such that the reformulation of the formula F on the abstraction level L is U-simple.*

5 Appropriate granularity level is difficult to find

In the previous section, we have shown that every problem becomes easy to solve if we reformulate it on the proper level of abstraction.

- At first glance, this result may sound very optimistic.
- However, it is not as optimistic as it may sound, because it may be difficult to find this "proper level of abstraction".

Namely, in this section, we will show that finding the corresponding "proper level of abstraction" is, in general, an computationally intractable problem, i.e., *expert's knowledge is needed* to make it efficient.

Proposition 3. *Let U be a polynomial-time algorithm. Then, the following problem is NP-hard: given a satisfying propositional formula F, find an abstraction level L for which the reformulation of F on the level L is U-simple.*

6 Symmetries lead to good heuristics in finding the appropriate granularity level

The above results motivate a heuristic for solving complex problems. Although our results do not lead to a universal feasible *algorithm* for solving complex problems, we can still use them as a heuristic for solving such problems: namely, instead of trying to solve each instance problem directly, we can look for an appropriate abstraction (granularity) level, and then try to solve the corresponding higher-level problem.

One possibility of using this heuristic: symmetries. In the previous section, we have mentioned that to find an appropriate abstraction level, we must have some additional knowledge. One of the possible ways to describe this knowledge is by using *symmetries*, i.e., statements that some properties are invariant under certain transformations. Symmetries are very successful in modern physics; moreover, it has been proven that an arbitrary property can be reformulated in terms of appropriate symmetries [7].

In [9], we have shown that a general symmetry-group approach explains the most frequently used fuzzy and neural heuristics. It is therefore reasonable to use this approach for choosing the abstraction level as well. We will show that this idea indeed leads to good heuristics.

First case: symmetric formulas. The first case is when we have explicit symmetries, e.g., if we have a propositional formula F which is invariant under arbitrary permutations of certain variables $x_{i_1}, \ldots, x_{i_k}$. In this case, we can naturally divide the set $\{0,1\}^n$ into $c = k+1$ classes $S_0, S_1, \ldots, S_c$, where S_r is the class of all Boolean vectors in which exactly r of the variables $x_{i_1}, \ldots, x_{i_k}$ are true. This idea was successfully used in [1, 2] and led to polynomial-time algorithms for several classes of propositional formulas for which previously known methods like *resolution method* (which forms the basis of most automatic theorem provers) require exponential time.

General case. In the general case, when permutations are not symmetries, we can still use symmetries. Namely, instead of following standard theorem proving techniques, in which at any given moment of time, each statement is either assumed to be true or assumed to be false, we may follow the idea of fuzzy logic, and describe, at each moment of time, the current *degree of belief* of the expert in each statement. Then, the question is: how can we update these degrees of belief. It turns out [5] that the general group-symmetry approach enables us to find the optimal formulas for such update, and that the resulting algorithm is indeed working reasonably well.

Comment. Interestingly, this method not only allows us to often find a satisfying vector, but it also enables us, when we do not know whether the original formula is permutation-symmetric or not, to guess possible permutation symmetries [5]. This is important because checking symmetry is an NP-hard problem:

Definition 4. *We say that a propositional formula F with variables $x_1, \ldots, x_n$ is permutation-invariant with respect to variables x_1 and x_2 if for every Boolean vector $x = (x_1, \ldots, x_n)$, we have*

$$F(x_1, x_2, x_3, \ldots, x_n) \leftrightarrow F(x_2, x_1, x_3, \ldots, x_n).$$

Proposition 4. *The problem of checking whether a given propositional formula is permutation-invariant w.r.t. x_1 and x_2 is NP-hard.*

7 Proofs

Proof of Proposition 1. We can prove NP-hardness by reducing the propositional satisfiability problem for CNF formulas (a known NP-hard problem) to the hit-and-miss problem. Indeed, if we have a CNF formula $D_1 \& \ldots \& D_m$, then we can take $H = \{0, 1\}^n$, and take $\neg D_i$ as M_i. Then, a Boolean vector x is a solution to thus constructed hit-and-miss problem if and only if it solves the original satisfiability problem. Thus, if we were able to solve the hit-and-miss problem in reasonable time, we would also be able to solve the satisfiability problem in reasonable time, and hence, since satisfiability is NP-hard, any other problem from the class NP. Thus, by definition, the hit-and-miss problem is NP-hard. The proposition is proven.

Proof of Proposition 2. Since F is satisfiable, it has a satisfying vector $(v_1, \ldots, v_n)$. As F_1, we can now take a formula $a_1 \& \ldots \& a_n$, where for every i:

- if $v_i = 1 =$"true" then $a_i = x_i$; and
- if $v_i = 0 =$"false" then $a_i = \neg x_i$.

As F_2, we will take $\neg F_1$. As U, we can take the following three-stage algorithm:

- On the first stage, this algorithm checks, syntactically, whether a formula is of the type (1). If not, it stops; if yes, it continues.
- On the second stage, the algorithm U checks whether the formula F_1 indeed has the form $a_1 \& \ldots \& a_n$, where for each i, the literal a_i is either a variable x_i or its negation $\neg x_i$. It the formula F_1 does not have this form, the algorithm U stops; otherwise, it continues.
- On the final stage, the algorithm U substitutes the values $v_i = 1$ if $a_i = x_i$ and $v_i = 0$ if $a_i = \neg x_i$ into the formula F. If the result is "true", then the values $v_1, \ldots, v_n$ are returned as the desired satisfying vector.

One can easily see that:

- thus constructed algorithm U is indeed a satisfying vector, and that
- for every satisfying formula F, its reformulation on the above described abstraction level L is indeed U-simple.

The proposition is proven.

Proof of Proposition 3. Indeed, if we could have an easy algorithm for finding the appropriate abstraction level for an arbitrary satisfying propositional formula F, then, by applying the algorithm U to the reformulated formula, to find its satisfying vector in polynomial time.

Thus, the NP-hard problem of finding a satisfying vector for a satisfiable propositional formula can be reduced to our problem and hence, our problem is also NP-hard. The proposition is proven.

Proof of Proposition 4. To show that the symmetry-checking problem is NP-hard, let us reduce the known NP-hard problem (namely, the propositional satisfiability problem) to this one. Indeed, let $F(x_1, \ldots, x_n)$ be an arbitrary propositional formula. Let us define a new formula $G(y_1, y_2, x_1, \ldots, x_n)$ with two extra variables as $y_1 \& \neg y_2 \& F(x_1, \ldots, x_n)$. For this new formula, $G(0, 1, x_1, \ldots, x_n)$ is always false, while the truth value of $G(1, 0, x_1, \ldots, x_n)$ coincides with the truth values of $F(x_1, \ldots, x_n)$. Thus, the new formula G is permutation-invariant with respect to its first two variables if and only the original formula F attains only the values "false" (i.e., is not satisfiable).

This reduction proves that checking permutation-invariance is indeed NP-hard. The proposition is proven.

Acknowledgments. This work was supported in part by NASA under cooperative agreement NCC5-209, by NSF grant No. DUE-9750858, by the United Space Alliance grant No. NAS 9-20000 (PWO C0C67713A6), and by the Future Aerospace Science and Technology Program (FAST) Center for Structural Integrity of Aerospace Systems, effort sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-95-1-0518.

References

[1] E. Ya. Dantsin, "Two tautology proof systems based on the splitting method", *J. Soviet Math.*, 1983, Vol. 22, No. 3.

[2] E. Ya. Dantsin, "Algorithmics of propositional satisfiability problems", [8], pp. 5-22.

[3] M. E. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, Freeman, San Francisco, 1979.

[4] J. Goutsias, R. P. S. Mahler, and H. T. Nguyen (eds.), *Random sets: theory and applications*, Springer-Verlag, N.Y., 1997.

[5] V. Kreinovich, "Semantics of S. Yu. Maslov's iterative method," [8], pp. 23–51.

[6] V. Kreinovich, A. Lakeyev, J. Rohn, and P. Kahl, *Computational complexity and feasibility of data processing and interval computations*, Kluwer, Dordrecht, 1997.

[7] V. Kreinovich and L. Longpré, "Unreasonable effectiveness of symmetry in physics", *International Journal of Theoretical Physics*, 1996, Vol. 35, No. 7, pp. 1549–1555.

[8] V. Kreinovich and G. Mints (eds.), *Problems of reducing the exhaustive search*, American Mathematical Society (AMS Translations — Series 2, Vol. 178), Providence, RI, 1997.

[9] H. T. Nguyen and V. Kreinovich, *Applications of continuous mathematics to computer science*, Kluwer, Dordrecht, 1997.

[10] B. Traylor and V. Kreinovich, "A bright side of NP-hardness of interval computations: interval heuristics applied to NP-problems", *Reliable Computing,* 1995, Vol. 1, No. 3, pp. 343–360.

[11] L. A. Zadeh, *Information Granularity, Fuzzy Logic, and Computing with Words*, unpublished talk at the International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU'96), Granada, Spain, July 1, 1996.