Finding Least Expensive Tolerance Solutions and Least Expensive Tolerance Revisions: Algorithms and Computational Complexity

Inna Pivkina[1] and Vladik Kreinovich[2]

[1]Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
email ipivkina@cs.nmsu.edu
[2]Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968
vladik@utep.edu

Abstract

For an engineering design, tolerances in design parameters are selected so that within these tolerances, we guarantee the desired functionality. Feasible algorithms are known for solving the corresponding computational problems: the problem of *finding* tolerances that guarantee the given functionality, and the problem of *checking* whether given tolerances guarantee this functionality.

In this paper, we show that in many practical problems, the problem of choosing the *optimal* tolerances can also be solved by a feasible algorithm. We prove that a slightly different problem of finding the optimal tolerance *revision* is, in contrast, computationally difficult (namely, NP-hard). We also show that revision programming algorithms can be used to check whether a given combination of tolerance changes is optimal under given constraints – and even to find a combination that is optimal under given constraints.

1 Formulation of the Problem

The notion of the tolerances. One of the main purposes of engineering is to design objects with given functionality, be it computer chips or tall buildings. By solving the corresponding optimization problem, we can find the optimal values $x_1, \ldots, x_n$ of the corresponding parameters.

In practice, it is not possible to exactly maintain the values of the design parameters. As a result, we must find *tolerances* for these parameters x_i, i.e., ranges $\mathbf{x}_i = [\underline{x}_i, \overline{x}_i]$ of possible values of these parameters, so that an arbitrary combination of values $x_i \in \mathbf{x}_i$ from these ranges $\mathbf{x}_i$ still guarantees the design's functionality.

The notion of the tolerance solution. In precise terms, functionality usually means that certain quantities $y_1, \ldots, y_m$ depending on the parameters x_i must lie within given ranges $[\underline{y}_j, \overline{y}_j]$. For example, for all possible regimes of a computer chip, its temperature cannot exceed the given heat threshold, and its overall current cannot exceed the capacity of the related battery; for a tall building, for all winds within a given range, the deviation of the upper floor from its nominal position should be bounded from both sides by given bounds, etc.

Tolerances are usually reasonably narrow; as a result, within the corresponding ranges, we can safely ignore quadratic and higher order terms in the dependence $y_j = f_j(x_1, \ldots, x_n)$ of the quantities y_j on the design parameters x_i. Thus, we can safely assume that the dependence of y_j on x_i is linear:

$$y_j = y_j^{(0)} + \sum_{i=1}^{n} a_{ji} \cdot x_i.$$

In these terms, the requirement that $\underline{y}_j \le y_j \le \overline{y}_j$ can be rewritten as follows:

$$\underline{y}_j \le y_j^{(0)} + \sum_{i=1}^{n} a_{ji} \cdot x_i \le \overline{y}_j. \tag{1}$$

Just like we cannot manufacture objects with the exactly given values of design parameters, we also do not know the exact values of the coefficients describing the dependence between physical quantities. In particular, usually, we do not know the exact values of the coefficients $y_j^{(0)}$ and a_{ji}; we only know the *intervals* $\mathbf{y}_j^{(0)}$ and $\mathbf{a}_{ji}$ of possible values of these parameters. To guarantee the desired functionality, we must make sure that the condition (1) holds for all possible values $y_j^{(0)} \in \mathbf{y}_j^{(0)}$ and $a_{ji} \in \mathbf{a}_{ji}$ of these coefficients.

Definition 1

- *By a* tolerance problem, *we mean a tuple consisting of positive integers n and m and intervals $\mathbf{y}_j^{(0)}$ ($1 \le j \le m$) and $\mathbf{a}_{ji}$ ($1 \le j \le m$, $1 \le i \le n$).*
- *By the* tolerance solution set *of a tolerance problem, we mean the set of all the vectors x that satisfy the condition (1) for all $y_j^{(0)} \in \mathbf{y}_j^{(0)}$ and $a_{ji} \in \mathbf{a}_{ji}$.*

Historical comment. The notion of a tolerance solution set was first considered and analyzed in [20, 21, 22]; it was also analyzed, e.g., in [1, 5, 7, 8, 9, 15, 25, 26, 27].

Tolerance intervals $\mathbf{x}_i$ must be selected in such a way that the above condition holds for all $x_i \in \mathbf{x}_i$. Intervals that satisfy this requirement are called *tolerance solution* to the interval-valued system (1).

Definition 2 *We say that intervals* $\mathbf{x}_1, \ldots, \mathbf{x}_n$ *form a* tolerance solution *to a given tolerance problem if for every element* $x = (x_1, \ldots, x_n) \in \mathbf{x}_1 \times \ldots \times \mathbf{x}_n$, *the condition (1) is satisfied for all* $y_j^{(0)} \in \mathbf{y}_j^{(0)}$ *and* $a_{ji} \in \mathbf{a}_{ji}$.

In other words, intervals $\mathbf{x}_i$ form a tolerance solution if and only if the corresponding box $\mathbf{x}_1 \times \ldots \times \mathbf{x}_n$ is a subset of the tolerance solution set X.

Computational complexity of the corresponding computational problem: known results. One of the main computational problems related to design is to check whether there are values $x_1, \ldots, x_n$ that satisfy given constraints – and if there are such values, to find some such values. It is known that there exist feasible (polynomial-time) algorithms for solving both problems.

Proposition 1 *There exist feasible (polynomial-time) algorithms for checking whether the tolerance solution set* X *is non-empty and, in case it is non-empty, for producing an element* $x \in X$ *of this set.*

Proof. Let us show that our tolerance-related problem can be reduced to linear programming (i.e., to solving systems of linear inequalities). We want to find the values x_i for which $\underline{y}_j \le y_j^{(0)} + \sum a_{ji} \cdot x_i \le \overline{y}_j$ for all $y_j^{(0)} \in [\underline{y}_j^{(0)}, \overline{y}_j^{(0)}]$ and $a_{ij} \in [\underline{a}_{ij}, \overline{a}_{ij}]$. For every j, the largest possible value $\overline{t}_{ji}$ of each linear term $a_{ji} \cdot x_i$ is attained on one of the endpoints of the interval $\mathbf{a}_{ji}$, and a similar property is true for the smallest possible values $\underline{t}_{ji}$ of these terms. Therefore, the above inequality is equivalent to the following system of linear inequalities, with new variables $\underline{t}_{ji}$ and $\overline{t}_{ji}$: $\underline{t}_{ji} \le \underline{a}_{ji} \cdot x_i \le \overline{t}_{ji}$, $\underline{t}_{ji} \le \overline{a}_{ji} \cdot x_i \le \overline{t}_{ji}$, $\underline{y}_j \le \underline{y}_j^{(0)} + \sum\limits_i \underline{t}_{ji}$, and $\overline{y}_j^{(0)} + \sum\limits_i \overline{t}_{ji} \le \overline{y}_j$. Thus, we can use known polynomial-time algorithms for solving linear programming problems; see, e.g., [2, 23]. The proposition is proven.

Comment. As a corollary of this proof, we conclude that the tolerance solution set X is a solution set to a linear programming problem and is, thus, a convex polyhedron.

Most problem have many different tolerance solutions. We have already mentioned that from the geometrical viewpoint, finding a tolerance solution $\mathbf{x}_i$ means finding a box $\mathbf{x}_1 \times \ldots \times \mathbf{x}_n$ that is contained in the tolerance solution set X. We have also mentioned that the tolerance solution set X is a convex polyhedron determined by the system of linear inequalities.

For every convex polyhedron X, there are many boxes contained in X and thus, many possible tolerance solutions $\mathbf{x}_1, \ldots, \mathbf{x}_n$.

For example, we can narrow each interval; since the tolerance solution consists of all the values x_i that satisfy the given constraint, subintervals also form a tolerance solution. Of course, selecting such subintervals makes no practical sense: the narrower the tolerance intervals, the costlier it is to maintain them, so we must make our tolerance intervals as wide as possible.

Even if we ignore subintervals, and consider only tolerance solutions which are *Pareto optimal* – in the sense that none of the intervals $\mathbf{x}_i$ can be enlarged without going outside X – there are still many such tolerance solutions.

We must select an optimal tolerance solution. Theoretically, there exist many possible tolerance solutions; in practice, we must select one of them. How can we make this selection? Strictly speaking, tolerances are part of the design, so the problem of selecting tolerances can be viewed as a particular case of the general problem of selecting a design.

In general, when we select a design, we must guarantee the desired functionality; since there are usually several designs with a given functionality, we usually select the least expensive of these designs. When we select tolerances, all functionality requirements are already incorporated in the definition of a tolerance solution, so the only remaining criterion is cost. In other words, we must select the least expensive tolerance solution.

To formulate this problem in precise terms, we must analyze how the design cost depends on the selected tolerances.

How expensive are the tolerances? To answer this question, we will consider two reasonable practical design situations that represent two extreme cases of design.

On the one hand, we have design situations in which manufacturing is reasonably cheap, and checking the constraints is also reasonably cheap. As example of such situations is the design and manufacturing of computer chips. In such situations, since manufacturing is cheap, if one of the manufactured objects does not satisfy one of the constraints, it is cheaper to simply throw it away and manufacture a new one.

Manufacturing of computer chips is an extreme case of such situation, in which – at least in the beginning of the chip's lifetime – the throughput (percentage of satisfactory objects) can be as low as a few percents. In such situations, we can estimate the effect of tolerance on cost as follows. Let us assume that the manufacturing process produces designs for which the values x_i take values from the intervals $\mathbf{X}_i = [\underline{X}_i, \overline{X}_i]$. Since we have no information about the relative frequency of different values within the resulting box $\mathbf{X}_1 \times \ldots \times \mathbf{X}_n$, it is natural to assume that all these values are, in some reasonable sense, equally probable – i.e., that the probabilities of different values x within this box are described by a uniform distribution, for which the probability of x being within an arbitrary subbox $\mathbf{x} = \mathbf{x}_1 \times \ldots \times \mathbf{x}_n$ is proportional to the volume $V(\mathbf{x})$ of this subbox.

Thus, for each tolerance solution $\mathbf{x}$, the probability p that the manufacturing

process produces a satisfactory object is equal to $p = \text{const} \cdot V(\mathbf{x})$. This means that to produce a single satisfactory object, we must, on average, manufacture $1/p$ such objects. Thus, in such situations, the cost of producing a satisfactory object is proportional to $1/V(\mathbf{x})$. So, minimizing the cost ($\sim 1/V(\mathbf{x})$) means producing a solution set with the largest possible volume $V(\mathbf{x})$.

On the other hand, we have situations like car manufacturing, buildings design, etc., where manufacturing is very expensive. So, once we manufactured an object and it turned out to be somewhat outside the desired range, we should try to repair and/or fine-tune it.

This distinction occurs not only in engineering design and manufacturing, it is reflected in real life. Some objects are cheap so we simply throw them away if something is wrong with them. For example, if a sheet of paper from a packet has a flaw (e.g., a hole), it is natural to throw it away and to use the next one. On the other hand, other objects are expensive: e.g., if a computer breaks down we try to repair it first.

How does the cost of such repairable objects depend on tolerances? Every time we perform a repair or tuning, we need to measure the corresponding design parameter. So, the resulting cost is roughly proportional to the number of measurements performed during this process. According to statistics, in general, after we perform N independent measurements of the same quantity, we can estimate its value with an accuracy $\sim 1/\sqrt{N}$. Our objective is to achieve accuracy proportional to the widths $w_i = \overline{x}_i - \underline{x}_i$ of the corresponding tolerance intervals. To achieve this accuracy, we need to perform N_i measurements, where $1/\sqrt{N_i} \approx w_i$ – i.e., $N_i \sim 1/w_i^2$ measurements. To measure all the desired parameters $x_1, \ldots, x_n$, we must therefore perform $\sum\limits_{i=1}^{n} N_i \sim \sum\limits_{i=1}^{n} w_i^{-2}$ measurements. So, in this repairable case, the cost of a tolerance solution is proportional to $S(\mathbf{x}) = \sum\limits_{i=1}^{n} w_i^{-2}$. Thus, the least expensive tolerance solution is the one for which this sum $S(\mathbf{x})$ is the smallest possible.

How can we solve the corresponding optimization problems?

Analysis of the problem. Both cost characteristics – the (maximized) volume $V(\mathbf{x})$ and the (minimized) sum $S(\mathbf{x})$ – have a common feature: the volume is a concave function of its variables, while the sum $S(\mathbf{x})$ (as one can easily check) is convex.

Resulting algorithm for computing the least expensive tolerances.

Theorem 1 *There exist a feasible (polynomial-time) algorithm that, given a tolerance problem and a convex objective function $F(\mathbf{x}_1, \ldots, \mathbf{x}_n)$, produces an F-optimal tolerance solution to the given tolerance problem.*

Proof. We know that the tolerance solution set X is convex. Thus, if we have two tolerance solutions $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and $\mathbf{x}' = (\mathbf{x}'_1, \ldots, \mathbf{x}'_n)$, then their

convex combination

$$(\alpha \cdot \mathbf{x}_1 + (1-\alpha) \cdot \mathbf{x}'_1, \ldots, \alpha \cdot \mathbf{x}_n + (1-\alpha) \cdot \mathbf{x}'_n)$$

is also a tolerance solution. In other words, the class of all tolerance solutions is also convex.

There exist efficient algorithms for minimizing a convex function (or maximizing a concave function) on a convex set; see, e.g., [28]. Thus, in both situations, we can efficiently find the c-optimal tolerance set. The theorem is proven.

Problem of optimal tolerance revision: formulation. Suppose that we already have found a tolerance solution that satisfies given design constraints. In general, if we have a different design-related problem, we must solve it anew.

Often, the new problem is largely similar to the already solved one – e.g., we want to design a similar building or a similar chip, but for a slightly different environment and thus, for slightly different constraints. In such situations, instead of designing "from scratch", it is desirable to modify (revise) the previous tolerance solution. The fewer modifications we need to make, the less expensive the resulting implementation is. Thus, it is desirable, given a combination of intervals and a design problem, among all possible revisions of the given combination of intervals that turn this collection into a tolerance solution, to find a one which is minimal (in some reasonable sense).

A natural description of minimality comes from the fact that in many practical situations, an implementation of each of the tolerances $\underline{x}_i \le x_i$ and $x_i \le \overline{x}_i$ requires that we implement objects with the values $\underline{x}_i$ and $\overline{x}_i$. For example:

- When x_i is weight, we may implement two objects with weights $\underline{x}_i$ and $\overline{x}_i$ and compare the weight x_i of the object with the weights of these standard objects.
- When x_i is an electric current, we may implement standard currents of size $\underline{x}_i$ and $\overline{x}_i$ and compare the current x_i with these standard currents.
- When x_i is length, we may implement two bodies with lengths $\underline{x}_i$ and $\overline{x}_i$ and compare the length x_i of an object with these two standard lengths.

One may ask: why not simply measure x_i and compare the measurement result with the numbers $\underline{x}_i$ and $\overline{x}_i$? The reason is that for most quantities, measurement consists of repeated comparisons with standard values; see, e.g., [19]. If we simply compare the value x_i with two standard signals $\underline{x}_i$ and $\overline{x}_i$, then we only need two comparisons. However, if we actually measure x_i, this measurement involves many different comparisons. It is therefore less expensive to implement pure comparisons.

From this viewpoint, changing an endpoint $\underline{x}_i$ or $\overline{x}_i$ requires re-designing the corresponding standard object. So, the fewer endpoints we must change, the least expensive is the design revision.

Thus, we arrive at the following problem:

Definition 3 *Let $\mathcal{P}$ be a given tolerance problem, let $\mathbf{x}_1 = [\underline{x}_1, \overline{x}_1]$, ..., $\mathbf{x}_n = [\underline{x}_n, \overline{x}_n]$ be intervals, and let c be a positive integer. We say that a tolerance set $\mathbf{x}'_1, \ldots, \mathbf{x}'_n$ is obtained from $\mathbf{x}_i$ by c revisions, if we can get this set from the given intervals by $\le c$ changes of endpoints.*

Problem of optimal tolerance revision: computational complexity. Let us prove that this problem is computationally intractable (to be more precise, NP-hard):

Theorem 2 *The problem of checking whether a given tolerance problem has a tolerance solution which is obtained form a given collection of intervals by a given number of revisions is NP-hard.*

Proof. To prove NP-hardness of our problem, we will reduce, to this new problem, a known NP-complete *partition problem* (see, e.g., [4, 7]). The partition problem is as follows: given n positive integers $s_1, \ldots, s_n$, check whether there exist values $\varepsilon_i \in \{-1, 1\}$ such that $\sum\limits_{i=1}^{n} s_i \cdot \varepsilon_i = 0$.

For every instance of the subset problem, we take:

- the initial intervals $\mathbf{x}_1 = \ldots = \mathbf{x}_n = [-1, 1]$,
- the following tolerance problem: $m = 1$, $a_{1i} = s_i$, and $\mathbf{y}_1^{(0)} = \mathbf{y}_1 = [0, 0]$, and
- $c = n$.

For this problem, the tolerance set X consists of all the vectors $x = (x_1, \ldots, x_n)$ for which $\sum\limits_{i=1}^{n} s_i \cdot x_i = 0$. This set is defined by a single linear equation and thus, it forms a hyperplane in n-dimensional space. Therefore, no non-degenerate box $\mathbf{x}_1 \times \ldots \times \mathbf{x}_n$ can be a subset of this set X. Thus, for this problem, the only possible tolerance solutions are fully degenerate ones, for which, for every i, we have $\mathbf{x}_i = [x_i, x_i] = \{x_i\}$ for some real number x_i. So, finding a tolerance set means finding the values x_i (or, equivalently, intervals $[x_i, x_i]$) for which $\sum\limits_{i=1}^{n} s_i \cdot x_i = 0$.

For each i, we go from the original non-degenerate interval $[-1, 1]$ to a degenerate one. Thus, for each i, at least one endpoint has to be revised. So, overall, at least n endpoints have to be revised. The only way to get a tolerance solution after $\le c = n$ revisions is to make sure that for every i, there is exactly one revision. In other words, the only way to get a tolerance solution in a given number of revisions is to make sure that for every i, x_i coincides with one of the endpoints of the original interval $[-1, 1]$, i.e., that $x_i = 1$ or $x_i = -1$.

Thus, we have $\sum\limits_{i=1}^{n} s_i \cdot x_i = 0$ for some $x_i \in \{-1, 1\}$. So, if the tolerance revision problems has a solution in $\le c$ revisions, then the original instance of

the partition problem has a solution. Conversely, if this instance has a solution $\varepsilon_i \in \{-1, 1\}$, then the intervals $[\varepsilon_i, \varepsilon_i]$ form a tolerance solution that can be obtained from the original intervals by $c = n$ revisions.

The reduction is proven. This completes the proof of the theorem.

Problem of optimal tolerance revision under constraints: formulation. In general, revising a tolerance set $[\underline{x}_1, \overline{x}_1]$, ..., $[\underline{x}_n, \overline{x}_n]$ means changing some of the $2n$ thresholds $\underline{x}_1, \overline{x}_1, \ldots, \underline{x}_n, \overline{x}_n$.

The fact that we need changes means that we cannot keep all the thresholds, we must change some of them. For example, for some quantity x_i, we may know that we cannot keep both original thresholds $\underline{x}_i$ and $\overline{x}_i$, at least one of these thresholds must be changed. In other words:

- if we do not change the lower threshold $\underline{x}_i$, then we must change the upper threshold $\overline{x}_i$, and
- if we do not change the upper threshold $\overline{x}_i$, then we must change the lower threshold $\underline{x}_i$.

In some cases, for two quantities x_i and x_j, we can keep thresholds for one of these quantities at the expense of tightening the thresholds for the other quantity. In other words, out of the four thresholds $\underline{x}_i$, $\overline{x}_i$, $\underline{x}_j$, and $\overline{x}_j$, at least one must change:

- if we do not change the thresholds $\underline{x}_i$, $\overline{x}_i$, and $\underline{x}_j$, then we must change the threshold $\overline{x}_j$;
- if we do not change the thresholds $\underline{x}_i$, $\overline{x}_i$, and $\overline{x}_j$, then we must change the threshold $\underline{x}_j$;
- if we do not change the thresholds $\underline{x}_i$, $\underline{x}_j$, and $\overline{x}_j$, then we must change the threshold $\overline{x}_i$;
- if we do not change the thresholds $\overline{x}_i$, $\underline{x}_j$, and $\overline{x}_j$, then we must change the threshold $\underline{x}_i$.

In the previous section, we implicitly assumed that all the thresholds are implemented independently of each other. This is indeed true in some practical situations. In such situations, threshold changes can be also done independently for different thresholds. In these cases, any combination of threshold changes is possible, and to find the optimal tolerance revision, we can simply minimize the number of such changes.

In many other practical situations, however, the implementations of the threshold correlations are dependent on each other. In such situations, not all combinations of threshold changes make sense, we may have additional constraints on the corresponding changes.

For example, for some quantities x_i, changing the value of the corresponding threshold can be very expensive. In such situations, we would like to avoid changing both thresholds, i.e.,

- if we change the lower threshold $\underline{x}_i$, we should not change the upper threshold $\overline{x}_i$, and
- if we change the upper threshold $\overline{x}_i$, we should not change the lower threshold $\underline{x}_i$.

If we have several expensive-to-change quantities, then we may have constraints that restrict how many of the corresponding thresholds we can change. For example, if we have two expensive-to-change quantities x_i and x_j, with a total of four thresholds, and we can only afford to change no more than three of them, then we end up with the following four constraints:

- if we change $\underline{x}_i$, $\overline{x}_i$, and $\underline{x}_j$, then we should not change $\overline{x}_j$;
- if we change $\underline{x}_i$, $\overline{x}_i$, and $\overline{x}_j$, then we should not change $\underline{x}_j$;
- if we change $\underline{x}_i$, $\underline{x}_j$, and $\overline{x}_j$, then we should not change $\overline{x}_i$;
- if we change $\overline{x}_i$, $\underline{x}_j$, and $\overline{x}_j$, then we should not change $\underline{x}_i$.

In the above examples, we implicitly assumed that for each quantity x_i, the thresholds $\underline{x}_i$ and $\overline{x}_i$ are implemented independently. For some quantities x_i, when the thresholds $\underline{x}_i$ and $\overline{x}_i$ are very close, an implementation of $\overline{x}_i$ can be obtained as a simple (and thus, reasonably inexpensive) modification of the implementation of $\underline{x}_i$, and vice versa. In such situations, once we change one of these thresholds, it may be beneficial to change the other one so that the resulting combination $[\underline{x}_i, \overline{x}_i]$ remains optimal: we may pay a little bit for this change, but we may also gain a lot in terms of the optimality. In such situations, we have the following two constraints:

- if we change the lower threshold $\underline{x}_i$, we should also change the upper threshold $\overline{x}_i$, and
- if we change the upper threshold $\overline{x}_i$, we should also change the lower threshold $\underline{x}_i$.

In general, we may have a lot of constraints of this type. In such situation, it is not immediately clear how to find a combination of threshold changes that satisfies all these constraints.

In general, there are usually several different combinations of threshold changes that satisfy given constraints. Since there is an expense associated with each change, we should select, among such combinations, the optimal one – i.e., the one in which the changes are minimal (in some reasonable sense).

Often, specialists have already planned some changes and prepared their implementation. Their expertise may not be perfect, hence the prepared set of changes may not satisfy all the required constraints. In this case, since we have already prepared the implementations of the planned thresholds, it is reasonable to look for a plan that represents the minimal change from the prepared imperfect plan of changes.

A natural interpretation of "minimal change" is when every change is justified – i.e., if without this change, we will not be able to satisfy all the constraints.

Revision programming: a brief reminder. To design an efficient algorithm for producing the desired combination of threshold changes, we will use revision programming [12, 13, 14, 17].

Let us start by briefly describing the main notions of revision programming. The knowledge forming a general database or knowledge base can be described as a finite collection of atomic statements ("atoms").

For example, in our case, it is natural to consider atomic statements of two types:

- statements of the type "the threshold $\underline{x}_i$ is changed"; we will denote these atomic statements by $\underline{c}_i$; and
- statements of the type "the threshold $\overline{x}_i$ is changed"; we will denote these atomic statements by $\overline{c}_i$.

For most databases, the set U of all possible atomic statements is usually (large but) finite. This finite set is called a *Universe of discourse*, or *Universe*, for short. Formally, we can simply say that we have a finite set U; its elements are called *atomic statements*, or simply *atoms*. Subsets of U are called *databases*.

To describe changes in databases, we will use the following notations: $\mathbf{in}(a)$ means that the atom a is a part of the database, and $\mathbf{out}(a)$ means that the atom a is not a part of the database. Expressions of the form $\mathbf{in}(a)$ or $\mathbf{out}(a)$ are called *revision literals*.

For a revision literal α of the type $\mathbf{in}(a)$, its *dual* α^D is the revision literal $\mathbf{out}(a)$. Similarly, the *dual* of $\mathbf{out}(a)$ is $\mathbf{in}(a)$. A set of revision literals L is *coherent* if it does not contain a pair of dual literals.

For any set of atoms $B \subseteq U$, we denote the corresponding set of revision literals by $B^c = \{\mathbf{in}(a) : a \in B\} \cup \{\mathbf{out}(a) : a \notin B\}$.

A *revision rule* is an expression of one of the following two types:

$$\mathbf{in}(a) \leftarrow \mathbf{in}(a_1), \ldots, \mathbf{in}(a_m), \mathbf{out}(b_1), \ldots, \mathbf{out}(b_n) \tag{2}$$

or

$$\mathbf{out}(a) \leftarrow \mathbf{in}(a_1), \ldots, \mathbf{in}(a_m), \mathbf{out}(b_1), \ldots, \mathbf{out}(b_n), \tag{3}$$

where a, a_i, and b_j are atoms. The revision literal on the left hand side of $\leftarrow$ is called the *head* of the rule r and denoted by $head(r)$. The set of all the literals on the right hand side of $\leftarrow$ is called the *body* of the rule r and denoted by $body(r)$.

Each rule has two possible interpretations.

In *declarative* interpretation, a revision rule is viewed as a constraint on the database. For instance, rule (2) imposes the following condition: if the database contains all m atoms $a_1, \ldots, a_m$, and does not contain any of the n elements $b_1, b_2, \ldots, b_n$, then the database must contain a.

In *computational* (*imperative*) interpretation, a revision rule expresses a way to enforce a constraint. Assume that all atoms a_i, $1 \leq i \leq m$, belong to the current database B, and none of the atoms b_j, $1 \leq j \leq n$ belongs to B. Then, to enforce the constraint (2), the atom a must be added to the database.

Similarly, in the case of the constraint (3), the atom a must be removed from the database. A collection of revision rules is also called a *revision program*.

It is worth mentioning that in the declarative interpretation, when we formulate a rule of the type (2), what we are saying, in effect, is that either a is in the database B, or at least one of the atoms a_i, $1 \leq i \leq m$, is *not* in the database, or at least one b_j, $1 \leq j \leq n$, is *in* the database. From the declarative viewpoint, there is no difference between the rule (2) and, e.g., the rule

$$\mathbf{out}(a_1) \leftarrow \mathbf{in}(a_2), \ldots, \mathbf{in}(a_m), \mathbf{out}(a), \mathbf{out}(b_1), \ldots, \mathbf{out}(b_n) \tag{4}$$

However, from the computational viewpoint, there is a difference between the rules (2) and (4).

Indeed, in principle, if the original database B does not satisfy the rule, then we make this rule true in two possible ways: we can force the conclusion $head(r)$ of the implication r to be true (as we did), or we can make the premise $body(r)$ of the rule r to be false, by removing one of the atoms a_i or by adding one of the atoms b_j. In revision programming, the rule specifies not only what constraint we want to be satisfied, but also how exactly we want this rule to be enforced. The rule (2) says that if the implication expressed by the rule is not true, i.e., if the premise is true and a is not in the database, we should add a to the database. If, in a similar situation, we want to delete a_1 from the databases, then we should formulate the corresponding rule in the form (4).

Revision programming also enables us to describe the situations in which we allow several different ways of revising the database. For example, in the above case, if we want to allow both adding the atom a and deleting the atom a_1, then we can describe this by adding both rules (2) and (4) to the corresponding collection of revision rules.

Let us now describe this formally. For a database $B \subseteq U$ and an atom $a \in U$, the revision literal $\mathbf{in}(a)$ is *true* if $a \in B$ and false if $a \notin B$. Similarly, the revision literal $\mathbf{out}(a)$ is *true* if $a \notin B$ and false if $a \in B$. We say that a collection P of revision rules is *satisfied* by B (or, alternatively, that B is a *model* of P) if all the implications $r \in P$ are true in this interpretation.

Revision programming formalizes the notion of a "minimal" revision as a revision in which every change must be justified, so that there are no unnecessary, unjustified changes. The main idea behind the corresponding definition of a *justified revision* (see, e.g., [12, 14]) is as follows.

- Let $I \subseteq U$ be an *initial database*, i.e., the initial set of atoms.
- Let P be a *collection of revision rules*, i.e., a set of rules (of the type (2) or (3)) that the revised database must satisfy.
- Let $R \subseteq U$ be a *revised database* that satisfies all all the rules from P.

When is R a "justified" revision of I?

As we have mentioned, from the viewpoint of revision literals, the initial database I can be described as a collection I^c of statements $\mathbf{in}(a)$ corresponding to all $a \in I$ and statements $\mathbf{out}(a)$ corresponding to all atoms $a \notin I$. Similarly,

the revised database R can be described as a collection R^c of statements $\mathbf{in}(a)$ corresponding to all $a \in R$ and statements $\mathbf{out}(a)$ corresponding to all atoms $a \notin R$. We want to define a reasonable notion of the *minimal* update. Intuitively, the word "minimal" means that many of the statements $\mathbf{in}(a)$ and $\mathbf{out}(a)$ that were initially true remain true in the revised database, in other words, that the set R^c should be "close" to the set I^c.

Specifically, we would like to require that every difference between these sets I^c and R^c is justified by the rules from P. Once we fix the literals $\mathbf{in}(a)$ and $\mathbf{out}(a)$ that do not change – they form the set $I^c \cap R^c$ – we can then apply the rules from P and deduce other revision literals. Let us describe this deduction is detail. We want to describe the set S of all the revision literals that can be deduced from $I^c \cap R^c$ by using the rules P.

This set can be obtained by using the following natural algorithm. At each stage k of this algorithm, we have a set S_k that contains both the original revision literals from $I^c \cap R^c$ and some literals that we deduced from the original ones by using the rules. At each stage of the algorithm, this set will grow until we get all the literals that can be thus deduced. We start with the set $S_0 = I^c \cap R^c$. Once we have S_k, we proceed as follows: For each rule r, we check that all the literals from the body (premise) of this rule are already derived, i.e., are already in the set S_k. If they are, we check that the head $head(r)$ (conclusion) of this rule is in S_k; if $head(r)$ is not in S_k, we mark this literal $head(r)$ as deducible.

- If after reviewing all the rules, it turns out that we did not mark anything, we stop and return S_k as the desired set S of all literals that can be deduced from the original ones by using the rules from P.
- Otherwise, if some new literals were marked, we add the marked (deducible) literals to the set S_k. The resulting enlarged set will be the set S_{k+1}.

We want every literal from R^c to be thus justified. In other words, we want the resulting set S to coincide with R^c.

The above algorithm for computing what can be deduced is well known in logic programming (see, e.g., [10, 11]); the resulting set S is called the *least model* of the collection consisting of the rules P and of the facts $I^c \cap R^c$. If we denote the least model of a collection P of rules and facts by $\mathcal{L}(P)$, then we arrive at the following definition:

Definition 4 *Let $I, R \subseteq U$ be databases, and let P be the set of rules of type (2) and (3). We say that R is a P-*justified revision *of I (or simply* justified revision*, for short) if $R^c = \mathcal{L}(P \cup (I^c \cap R^c))$.*

By definition of the least model $\mathcal{L}$, one can easily see that if R^c is a justified revision, then it satisfies all the rules from the original collection P.

An efficient algorithm for checking whether a given revision is justified. It is known that the above algorithm requires polynomial (actually

quadratic) time. Indeed, at each stage, we either stop, or add at least one new literal from the head (conclusion) of one of the rules. Thus, the overall number of stages cannot exceed the number of rules in the collection P, hence it cannot exceed the length n of the original description of the problem.

On each stage, we check each literal from each rule. This checking cannot take longer than the the size of the rules; thus, each stage requires $O(n)$ computational steps. Therefore, the algorithm consists of $O(n)$ stages with $O(n)$ steps each, to the total of $O(n) \cdot O(n) = O(n^2)$ computational steps.

Thus, with this new definition of update minimality, we not only get a reasonable definition of a justified revision, but we also get a polynomial-time algorithm for checking whether a given revision is indeed justified.

Comments.

- It is worth mentioning that by using a more sophisticated algorithm, we can actually compute the least model in linear time [3, 6, 24].
- It is known [12, 14] that every justified revision R is also minimal in the sense of the above straightforward definition: there exists no set $R' \neq R$ that satisfies all the rules from P and for which $R' - I \subseteq R - I$ and $I - R' \subseteq I - R$.

An efficient algorithm for checking whether a given combination of tolerance revisions is optimal under constraints. Let us show that the problem of finding the optimal tolerance revision under constraints can be naturally reformulated in terms of revision programming. To describe this reformulation, we will use the above atomic statements $\underline{c}_i$ (meaning "the threshold $\underline{x}_i$ is changed") and $\overline{c}_i$ (meaning "the threshold $\overline{x}_i$ is changed").

The initial database I is the description of all the thresholds which are changed according to the original (imperfect) plan. The above constraints can then be naturally reformulated as rules of revision programming.

For example, the constraints that at least one of the two thresholds $\underline{x}_i$ and $\overline{x}_i$ must be changed take the following form:

$$\mathbf{in}(\overline{c}_i) \leftarrow \mathbf{out}(\underline{c}_i);$$

$$\mathbf{in}(\underline{c}_i) \leftarrow \mathbf{out}(\overline{c}_i).$$

The constraints that out of the four thresholds $\underline{x}_i$, $\overline{x}_i$, $\underline{x}_j$, and $\overline{x}_j$, at least one must change, take the following form:

$$\mathbf{in}(\overline{c}_j) \leftarrow \mathbf{out}(\underline{c}_i), \mathbf{out}(\overline{c}_i), \mathbf{out}(\underline{c}_j);$$

$$\mathbf{in}(\underline{c}_j) \leftarrow \mathbf{out}(\underline{c}_i), \mathbf{out}(\overline{c}_i), \mathbf{out}(\overline{c}_j);$$

$$\mathbf{in}(\overline{c}_i) \leftarrow \mathbf{out}(\underline{c}_i), \mathbf{out}(\underline{c}_j), \mathbf{out}(\overline{c}_j);$$

$$\mathbf{in}(\underline{c}_i) \leftarrow \mathbf{out}(\overline{c}_i), \mathbf{out}(\underline{c}_j), \mathbf{out}(\overline{c}_j).$$

The constraint that no more than one of the two thresholds $\underline{x}_i$ and $\overline{x}_i$ should be updated takes the form:

$$\mathbf{out}(\overline{c}_i) \leftarrow \mathbf{in}(\underline{c}_i);$$

$$\mathbf{out}(\underline{c}_i) \leftarrow \mathbf{in}(\overline{c}_i).$$

The constraints that we should change at most three of four thresholds $\underline{x}_i$, $\overline{x}_i$, $\underline{x}_j$, and $\overline{x}_j$ take the following form:

$$\mathbf{out}(\overline{c}_j) \leftarrow \mathbf{in}(\underline{c}_i), \mathbf{in}(\overline{c}_i), \mathbf{in}(\underline{c}_j);$$

$$\mathbf{out}(\underline{c}_j) \leftarrow \mathbf{in}(\underline{c}_i), \mathbf{in}(\overline{c}_i), \mathbf{in}(\overline{c}_j);$$

$$\mathbf{out}(\overline{c}_i) \leftarrow \mathbf{in}(\underline{c}_i), \mathbf{in}(\underline{c}_j), \mathbf{in}(\overline{c}_j);$$

$$\mathbf{out}(\underline{c}_i) \leftarrow \mathbf{in}(\overline{c}_i), \mathbf{in}(\underline{c}_j), \mathbf{in}(\overline{c}_j).$$

The constraint that once we change one of the threshold $\underline{x}_i$ and $\overline{x}_i$, we should change the other one as well, is described as follows:

$$\mathbf{in}(\overline{c}_i) \leftarrow \mathbf{in}(\underline{c}_i);$$

$$\mathbf{in}(\underline{c}_i) \leftarrow \mathbf{in}(\overline{c}_i).$$

We have already described an efficient revision programming algorithm for checking whether a given revision is justified. Since we have reformulated our tolerance-related problems in terms of revision programming, we can thus use this algorithm to check whether a given combination of threshold changes is optimal under given constraints.

Therefore, we have an efficient algorithm for checking whether a given combination of tolerance revisions is optimal under constraints.

Revision programming can be also used to find a combination of tolerance revisions which is optimal under given constraints. In the previous section, we described how revision programming can be used to *check* whether a given combination of tolerance revisions is indeed optimal under given constraints.

It is also possible to use revision programming to also *find* such optimal combinations of tolerance revisions. Of course, as we have mentioned, the problem of finding an optimal combination of tolerance revisions is, in general, NP-hard, so we cannot hope that the resulting algorithm will always work fast; however, for small size problems, this algorithm is usually reasonable efficient.

This algorithm is based on the fact that certain transformations (called *shifts*) preserve justified revisions [12]. For each set $W \subseteq U$, a *W-transformation* is defined as follows:

- For every revision literal α (i.e., a literal of the form $\mathbf{in}(a)$ or $\mathbf{out}(a)$), we define
$$T_W(\alpha) \stackrel{\text{def}}{=} \begin{cases} \alpha^D & \text{when } a \in W \\ \alpha & \text{when } a \notin W. \end{cases}$$

- For every set L of revision literals, we define $T_W(L)$ as the result of applying the transformation T_W to all the literals from this set L, i.e., as

$$T_W(L) \stackrel{\text{def}}{=} \{T_W(\alpha) \,|\, \alpha \in L\}.$$

- Similarly, for every set A of atoms, we define $T_W(A)$ as follows:

$$T_W(A) = \{a \,|\, \mathbf{in}(a) \in T_W(A^c)\}.$$

- Finally, for a collection P of revision rules, we define $T_W(P)$ as the result of applying T_W to every literal in P.

The Shifting Theorem [12] states that for any two databases I and J, database R is a P-justified revision of I if and only if $T_{I\Delta J}(R)$ is a $T_{I\Delta J}(P)$-justified revision of J, where $I\Delta J \stackrel{\text{def}}{=} (I - J) \cup (J - I)$.

Let I be the initial database, and let P be a given collection of revision rules. Then, the resulting algorithm for finding a justified revision of I is as follows:

1. Apply the transformation T_I to P to obtain $T_I(P)$ (corresponding to the empty initial database $J = \emptyset$).

2. Convert $T_I(P)$ into a logic program with constraints by replacing revision rules of the type (2) by

$$a \leftarrow a_1, \ldots, a_m, \text{not } b_1, \ldots, \text{not } b_n \tag{5}$$

   and replacing revision rules of the type (3) by constraints

$$\leftarrow a, a_1, \ldots, a_m, \text{not } b_1, \ldots, \text{not } b_n. \tag{6}$$

   We will denote the resulting logic program with constraints by $lp(T_I(P))$.

3. Apply an "answer set" programming engine (e.g., `smodels` [16]) to the program $lp(T_I(P))$. These engines return a collection of sets of atoms called *answer sets*.

4. Finally, apply the transformation T_I to the answer sets produced by the engine.

As shown in [12], the resulting sets will be exactly P-justified revisions of I.

Since we have shown that our problem – finding a combination of tolerance revisions which is optimal under given constraints – can be described in terms of revision rules, we can therefore use this algorithm to find a combination of tolerance revisions which is optimal under given constraints.

Comment. In the above text, we considered only constraints that have to be satisfied. In addition to such constraints, there are usually constraints that are desirable but not necessary. There can be several different levels of desirability, e.g., the user may consider some constraints more important than the others. In other words, instead of a set of equally important constraints, we have a set of constraints with user-defined preferences between these constraints.

An extension of revision programming to such situations is presented in [18]; by using algorithms presented in [18], we can therefore incorporate such preferences into our tolerance-related problems.

Acknowledgments. I.P. was supported by the NSF-funded ADVANCE Institutional Transformation Program at New Mexico State University, Grant No. 0123690. V.K. was supported in part by NSF grants EAR-0225670 DMS-0532645 and by Texas Department of Transportation grant No. 0-5453.

References

[1] O. Beaumont and B. Philippe, "Linear Interval Tolerance Problem and Linear Programming Techniques", *Reliable Computing*, 2001, Vol. 7, No. 6, pp. 433–447.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 2001.

[3] W. Dowling and J. Gallier, "Linear Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae", *Journal of Logic Programming*, 1984, Vol. 3, pp. 267–284.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freemand and Company, San Francisco, California, 1979.

[5] G. Heindl, "A Representation of the Interval Hull of a Tolerance Polyhedron Describing Inclusions of Function Values and Slopes", *Reliable Computing*, 1999, Vol. 5, No. 3, pp. 269–278.

[6] A. Itai and J. Makowsky, *On the complexity of Herbrand's theorem*, Technical Report No. 243, Department of Computer Science, Israel Institute of Technology, Haifa, 1982.

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

[8] A. V. Lakeyev and S. I. Noskov, "A description of the set of solutions of a linear equation with interval defined operator and right-hand side" *Russian Academy of Sciences, Doklady, Mathematics*, 1993, Vol. 47, No. 3, pp. 518–523.

[9] A. V. Lakeyev and S. I. Noskov, "On the solution set of a linear equation with the right-hand side and operator given by intervals", *Siberian Math. J.*, 1994, Vol. 35, No. 5, pp. 957-966.

[10] V. Lifschitz, "Foundations of logic programming", In: *Principles of Knowledge Representation*, CSLI Publications, Stanford, CA, 1996, pp. 69–127.

[11] J. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1984.

[12] V. W. Marek, I. Pivkina, and M. Truszczyński, "Revision programming = logic programming + constraints", *Proceedings of the 12th International Workshop on Computer Science Logic CSL'98*, Springer-Verlag Lecture Notes in Computer Science, 1999, Vol. 1584, pp. 73–89.

[13] V. W. Marek, I. Pivkina, and M. Truszczyński, "Annotated revision programming", *Artificial Intelligence*, 2002, Vol. 138, pp. 149–180.

[14] V. W. Marek and M. Truszczyński, "Revision programming", *Theoretical Computer Science*, 1998, Vol. 190, pp. 241–277.

[15] A. Neumaier, *Interval Methods for Systems of Equations*, Cambridge University Press, Cambridge, 1990.

[16] I. Niemelä and P. Simons, "Efficient implementation of the well-founded and stable model semantics", *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96, Bonn, Germany, September 2–6, 1996*, MIT Press, Cambridge, Massachusetts, 1996, pp. 289–303.

[17] I. Pivkina and V. Kreinovich, "Minimality of Solution Update in Conflict Resolution: An Application of Revision Programming to von Neumann-Morgenstern Approach", *International Journal of Intelligent Systems*, 2005, Vol. 20, No. 9, pp. 939–956.

[18] I. Pivkina, E. Pontelli, and T. C. Son, "Revising Knowledge in Multi-Agent Systems Using Revision Programming with Preferences, in *Computational Logic in Multi-Agent Systems, 4th International Workshop CLIMA'2004, Fort Lauderdale, FL, USA, January 6–7, 2004, Revised Selected and Invited Papers*, Springer Lecture Notes in Artificial Intelligence, 2004, Vol. 3259, pp. 134–158.

[19] S. Rabinovich, *Measurement Errors and Uncertainties: Theory and Practice*, Springer-Verlag, New York, 2005.

[20] J. Rohn, *Input-output planning with inexact data*, Freiburger Intervall-Berichte, 1978, Vol. 9.

[21] J. Rohn, "Input-output model with interval data", *Econometrica*, 1980, Vol. 48, No. 3, pp. 767–769.

[22] J. Rohn, "Inner solutions of linear interval systems", In: K. Nickel (ed.), *Interval Mathematics 1985*, Lecture Notes in Computer Science, Vol. 212, Springer-Verlag, Berlin, 1986, pp. 157–158.

[23] A. Schrijver, *Theory of Linear and Integer Programming*, Wiley, New York, 1998.

[24] M. G. Scutellá, "A note on Dowling and Gallier's top-down algorithm for propositional Horn satisfiability", *Journal of Logic Programming*, 1990, Vol. 8, pp. 265–273.

[25] V. V. Shaidurov and S. P. Shary, *Solution of interval algebraic problem on tolerances*, Krasnoyarsk, Academy of Sciences Computing Center, Technical Report No. 5, 1988 (in Russian).

[26] S. P. Shary, "On computibility of linear tolerance problem", *Interval Computations*, 1991, No. 1, pp. 92–98 (in Russian).

[27] S. P. Shary, "Algebraic approach to the interval linear static identification, tolerance, and control problems, or One more application of Kaucher arithmetic", *Reliable Computing*, 1996, Vol. 2, No. 1, pp. 3–34.

[28] S. A. Vavasis, *Nonlinear optimization: complexity issues*, Oxford University Press, N.Y., 1991.