Constraint Satisfaction Methods for Applications in Engineering

Esther Gelle[⋆], Boi V. Faltings[⋆], Denis E. Clément[⋄] and Ian F.C. Smith[⋄]
[⋆] Artificial Intelligence Laboratory (LIA)
[⋄] Institute of Structural Engineering and Mechanics (ISS-IMAC)
EPFL - Swiss Federal Institute of Technology
1015 Lausanne, Switzerland

Abstract

Constraints provide declarative descriptions of important requirements related to engineering projects. Most existing algorithms for constraint satisfaction require input consisting of binary constraints on variables that have discrete values. Such restrictions limit their use in engineering since complex constraints, involving several variables that have discrete and numeric values, are common. This paper provides an approach for decision support through approximating solution spaces that are defined by constraints. Our algorithm is not limited to a specific type of constraint but handles numeric and discrete variables in the same framework. Since a new type of local consistency narrows down the search space effectively, full-scale engineering tasks, such as designs involving hundreds of variables, are accommodated without excessive computational complexity. The approach is demonstrated for selection of appropriate wind bracing for single story steel-framed buildings. Results may be used for input into other tools containing algorithms such as those offering i) higher levels of consistency, ii) optimally directed point-solution search and iii) simulation behavior. Finally, extension to dynamic constraint satisfaction using different combinations of activation conditions is straightforward. It is expected that this approach will improve the performance of many existing and future computer-aided engineering tools.

1 Introduction

Key engineering tasks have always been decomposed into constraints. A structural design task, for example, is decomposed into functional criteria such as structural safety and serviceability and these criteria are represented in terms of inequalities. Examples of inequalities are : i) the stresses due to loads must be *less than* the resistance provided by the structural system and ii) the deflection under loading must be *less than* the maximum allowable deflection. Some criteria may also be geometric; for example, the available clearance beneath a bridge must be *greater than* the minimum required clearance. Since application of relevant geometrical and engineering principles is always carried out within the scope of such functional criteria, most important engineering decision making involves judgements regarding inequalities.

Inequality constraints define sets of solutions called solutions spaces. Currently, single point solutions are employed in engineering because solution spaces are difficult to calculate and to manage and because traditional media, such as engineering drawings, require fixed-value assignments for variables. Nevertheless, solution spaces have many advantages. For example, designers are free to consider other criteria that is difficult to express formally (such as aesthetics and socio-economic factors) in their final choice of design when they are able to explore solution spaces. Also, least commitment decision strategies are supported more effectively and this leads to less complicated revisions when conditions change. Other advantages include fewer artificial conflicts, opportunities for more effective negotiation between partners and efficient bounding of optimally directed decision strategies [Lottaz et al., 1999].

Engineering tasks can be represented as constraint satisfaction problems (CSP) in a natural way. A CSP is defined by a set of variables, each of which has a domain of possible values, and relations called constraints which restrict the variable values. The variables correspond to the relevant parameters of the engineering task and the constraints to (in)equalities expressing design criteria. The CSP approach provides search methods which find 1) single variable assignments that satisfy all the constraints and 2) descriptions of solution spaces, i.e. the set of all solutions. Early applications of CSP research emerged from the field of image interpretation [Waltz, 1975]. [Mackworth, 1977] and [Montanari, 1974] generalized this concept to any kind of discrete data related by binary constraints (so called discrete CSPs). CSP research is rapidly becoming a well established field and several introductory texts exist, for example [Tsang, 1993] and [Kumar, 1992].

Consistency is one of the techniques employed in the resolution of a CSP. Consistency techniques provide filters which remove inconsistent values, i.e. values that cannot be part of a solution, from the search space and thus make search more efficient. It has been shown in [Freuder, 1982b, Freuder, 1982a] that the degree of consistency reached in a CSP is related to the level of backtracking needed to solve it. Consistency techniques can be used as preprocessing in order to transform the given CSP in a CSP which is simpler to solve. This approach has led to the identification of restrictions on the constraint syntax ([Van Hentenryck et al., 1992]), the topology of the constraint network [Dechter, 1990] and the solution space described by constraints ([van Beek and Dechter, 1995]). Other work has concentrated on applying these techniques during search. In addition to the degree of consistency used, heuristics on the order in which the variables are instantiated determine the execution time ([Haralick and Elliott, 1980],[McGregor, 1979],[Prosser, 1993]).

Initially, most research concentrated on discrete binary constraint problems. Only recently, search techniques developed for discrete problems have been successfully applied to numeric domains (numeric CSPs). Since in a numeric CSP value combinations are not enumerable, interval analysis is employed to achieve partial degrees of consistency ([Van Hentenryck et al., 1995],[Lhomme, 1993],[Hyvönen, 1992],[Davis E., 1987]). [Sam-Haroud and Faltings, 1996] has extended the results of [van Beek and Dechter, 1995] to numeric CSPs and presents convexity conditions under which a numeric CSP becomes backtrack free. Since this algorithm relies on an explicit representation of the solution spaces, time complexity is high. In practical situations, it is typically applicable to problems involving less than 20 variables.

A generalization of the standard CSP has been defined in [Mittal and Falkenhainer, 1990] where the static CSP formalism, in which the variables and constraints of a CSP are given in the problem definition, is extended to the definition of conditional CSP (CCSP)[1]. In a CCSP, new variables and constraints may be activated during search. However, only discrete CCSPs are treated in [Mittal and Falkenhainer, 1990] and the resolution algorithm relies on the explicit enumeration of values. No proposals exist for a conditional approach involving mixed constraints (those that involve discrete and continuous variables) even though most engineering tasks are constrained by discrete and numeric CSPs.

This paper addresses the problem of solving engineering tasks represented as CSPs. The advantage of using CSPs in this field is that consistency techniques provide a means of representing an approximation of solution spaces instead of single point solutions. This provides the engineer with a better basis for decision making and even improves the efficiency of optimization algorithms used to derive point solutions since it helps the algorithm focus onto those parts of the search space which are likely to contain solutions. In this context, we propose the following new methods:

- The approximation of solution spaces is achieved by a new local consistency method for discrete and numeric variables providing good results in times of pruning and execution time. This is mainly due to the local consistency operator for numeric constraints which is superior in pruning power to existing methods [Faltings, 1994], [Gelle, 1998].

- A new search method embedding local consistency for discrete and numeric variables is proposed. In contrast to most existing approaches for solving mixed CSPs, which are based on a cooperation between constraint solvers [Tinelli and Harandi, 1996], it integrates the local consistency methods for discrete and numeric variables into the search process and also makes use of the mixed constraints to prune the search space.

- Finally, a new algorithm is introduced for solving conditional constraint problems with numeric constraints. From the conditional problem formulation a tree of static CSPs is generated to each of which the proposed local consistency methods are applied in order to eliminate inconsistent branches in the tree.

The paper is organized as follows. In section 2, constraint satisfaction problems are introduced. New local consistency and search methods are presented. The new search algorithm is presented in section 3. The performance of the algorithm is demonstrated using a design task in structural engineering in section 4 and finally, its application to conditional CSPs is described using the same example in section 5.

[1]We do not use the original name of dynamic CSP as it is used in [Mittal and Falkenhainer, 1990] since the word dynamic is used in different contexts even in the field of constraint satisfaction.

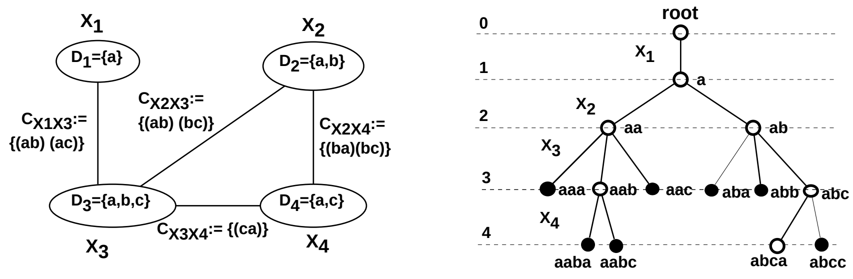

Figure 1: *Search tree solving a small CSP by backtracking. The black nodes have been pruned from the search space. Example taken from [Kondrak, 1994]*

2 Constraint Satisfaction Problems (CSPs)

A CSP $\langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$ is defined by the variables of interest $\mathcal{V}$, each variable $X_i \in \mathcal{V}$ with a domain $D_i \in \mathcal{D}$ of possible values, and a set of constraints $\mathcal{C}$ defining allowed value combinations over subsets of variables. The goal is to find one or all *solutions* to the CSP. A solution is a consistent assignment of values to all variables such that all constraints are satisfied. A constraint can be represented either extensionally by specifying all valid value combinations or intensionally by a logical predicate. We distinguish between:

- *discrete* constraints where the variables have domains consisting of discrete values; i.e. $D_i = \{d_{i_1}, \ldots, d_{i_m}\}$ for the variables X_i and the constraint is defined by a set of tuples, each tuple specifying one allowed value combination in $D_1 \times \ldots \times D_k$.
- *numeric* constraints where the variables take their value in a real interval and the constraint is a relation $E \circledcirc 0$ with $\circledcirc \in \{=, \geq, \leq\}$ and E, an expression built from constants, variables and operations $\{+, -, /, *, exp, ..\}$ over the reals.
- *mixed* constraints defined over a set of discrete $\mathcal{V}_{Dis}$ and a set of numeric variables $\mathcal{V}_{Con}$ where the constraint is a relation defined on $\mathcal{V}_{Dis} \cup \mathcal{V}_{Con}$.

In general, solving a CSP, that is deciding if there is at least one solution, is NP-complete since, in the worst case, all combinations of values for the variables have to be tested [Mackworth, 1977]. Lower levels of complexity are possible for specific cases. An example of a binary discrete CSP is given in Figure 1. Each node corresponds to a variable with its domain indicated in brackets and edges between the nodes represent constraints between pairs of variables with the allowed value tuples. On the right-hand side, the search tree for this problem is shown. The variables are instantiated sequentially in the order X_1, X_2, X_3, X_4. The depth of the search tree at a given node corresponds to the number of variables already instantiated; this is marked by dotted lines. The black nodes show inconsistent value combinations which have been pruned from the search space. There is only one solution to this problem: $\{X_1 = a, X_2 = b, X_3 = c, X_4 = a\}$.

2.1 Local consistency

Local consistency techniques transform the given CSP by removing inconsistent value combinations from subsets of variables and result in a combination of refined variable domains called labels. This transformation step is complete, i.e. it does not loose solutions. The set of refined labels corresponds to an approximation of the solution space of the given problem. In engineering tasks, it can be important to know at least an approximation of the solution space:

- In the traditional approach, engineering tasks are solved using some optimization method which provides single point results. This may lead to artificial conflicts when different engineers are involved in a project [Lottaz et al., 1999]. In this case, engineers will deliver a single solution to the subproblem they are involved in without knowing alternative solutions. When the subproblems are combined in the context of the overall project, a conflict might appear. Consistency methods can help to avoid this type of conflict.
- The result of local consistency is not only a valuable input to a search algorithm, e.g. an optimization method, but it can also help to visualize solution spaces. A visualization helps engineers to find a compromise in case of artificial conflicts and supports them during decision making. Local consistency thus provides a basis for least-commitment strategy when solving engineering problems.
- Although locally consistent solution spaces may still contain infeasible value combinations, they provide useful input into further search algorithms since they have already narrowed down the search space. Furthermore, local consistency with search is often the only alternative for solving engineering problems as it can be very difficult to provide a good objective function for this type of problem. Often, these problems involve several design criteria, which are at least partially contradictory. In this case, it is beneficial to visualize the solution space in order to find a good optimum.

We are interested here in the consistency between pairs of discrete values, also termed arc-consistency [Mackworth, 1977]. More precisely, arc-consistency takes each consistent value of a variable and checks if it is possible to instantiate a second variable such that all constraints between both variables are satisfied. Consistency techniques employ a compact representation of the effective combinations of consistent values. Such combinations of values are called *labels*. Initially, labels contain the same values as the domains. Local consistency removes values from the labels thus reducing the size of the search space (the number of value combinations). In Figure 1, the labels L_1 and L_3 are initially set to $\{a\}$ and $\{a, b, c\}$. One step of local consistency called `refine`, making L_3 locally consistent with label L_1 consists of removing the value a from L_3 because this value does not participate in any tuple of the constraint $C_{X_1X_3}$. Local consistency on a CSP is thus achieved by examining all ordered pairs of variables, narrowing down the variable labels and adding the variable pairs that are dependent on the changed labels to the queue. Changes are so propagated through the CSP (update of Q in Figure 2). In the example, the unique solution is obtained simply by enforcing local consistency.

Local consistency for discrete domains can be achieved in polynomial time in the size of the variable domains since the maximal number of value combinations to be explored for a variable pair is D^2 with D the maximal domain size. The goal of local consistency is thus to find the projection of a constraint onto each variable. The **refine** operator in

```
procedure propagate
begin
  $Q \leftarrow \{(X_i, X_j, C_{X_iX_j}) | i \neq j\}$
  while $Q \neq \emptyset$ do
    remove element $(X_i, X_j, C_{X_iX_j})$ from $Q$
    $L_{new} \leftarrow L_{X_j} \cap$ refine$(X_i, X_j, C_{X_iX_j})$
    if $L_{new} = \emptyset$ then
      return inconsistent !
    fi
    if $L_{X_j} \neq L_{new}$ then
      $L_{X_j} \leftarrow L_{new}$
      $Q \leftarrow Q \cup \{(X_j, X_k, C_{X_jX_k}) | k \neq i, k \neq j\}$
    fi
  od
end
```

```
function refine$(X, Y, C_{XY})$
begin
  $L \leftarrow$ all values $v$ of $L_Y$ such that
    there exists a value $w$ in $L_X$ and
    $C_{XY}$ is satisfied by $v, w$
  return $L$
end
```

Figure 2: *General algorithm for ensuring local consistency*

Figure 2 computes one projection. In order to simplify the discussion of local consistency over numeric constraints, we consider only binary numeric constraints in the following example, i.e. numeric constraints defined over two variables. Such a constraint defines a feasible two-dimensional region. Most existing refine operators for numeric constraints approximate the projection of this region onto the variable axes. They compute an approximation of the projection for each constraint through accounting for intersections of interval bounds with the constraints (2B-consistency [Lhomme, 1993]) or through using interval analysis (box-consistency [Benhamou et al., 1994], [Van Hentenryck et al., 1995]). Consider the projection onto the Y-axis of the feasible region defined by the constraints in Figure 3 such that all values of X are in I_X. Any algorithm propagating such constraints individually will result in the single interval I_Y for the label of Y.

The true projection of a feasible constraint region is computed using local extrema of the region [Faltings, 1994]. A refine operator combines all constraints defined on the same pair of variables simultaneously in a *total constraint.* The local extrema are defined as local extrema of individual constraints, intersections between constraints and intersections of an interval boundary with a constraint. When this algorithm is applied to the example in Figure 3, the true projection, consisting of two intervals I_1 and I_2, is obtained. The crosses are intersections between constraints; these points are calculated once and then used to compute the projection.

For constraints with higher arity, this approach can directly be applied as described above during search when all but two variables of the constraint network have been instantiated. An other possibility is to use an extension of the algorithm to ternary constraints given in [Faltings and Gelle, 1997, Gelle, 1998]. This generalized algorithm is valid for topologically simply connected feasible regions[2]. This means that three-dimensional regions with channels, i.e. holes that connect to the surface of a region, are excluded. Since

[2]A region is simply connected if any closed curve inside the region can be contracted to a single point.

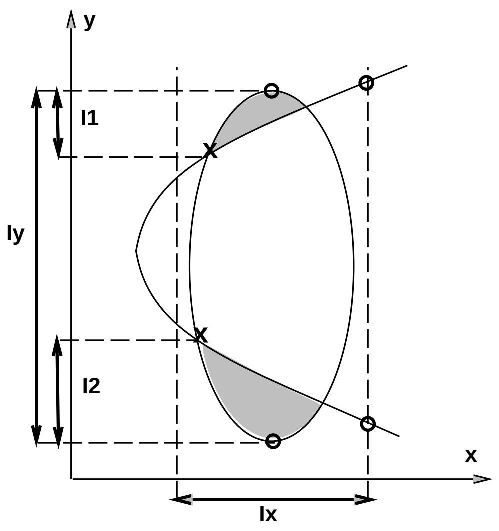

Figure 3: *Propagating constraints individually results in the single interval I_Y whereas the exact projection of the feasible region onto the Y-axis are the intervals I_1 and I_2.*

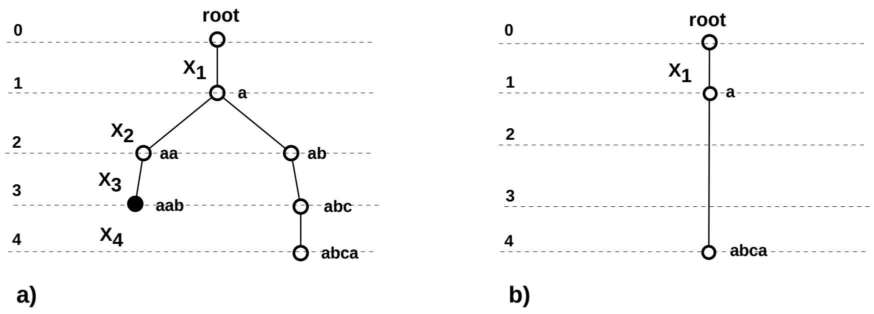

Figure 4: *Search tree of the CSP in Figure 1 solved by a) forward checking and b) maintaining arc-consistency.*

any constraint with an arbitrary number of variables can be decomposed into an equivalent network of ternary constraints, local consistency for a wide range of n-ary constraint networks can be computed.

In addition, some constraints are mixed in that they involve both discrete and numeric variables. Mixed constraints can be divided into the following classes:

1. Discrete constraints with interval values or real values, for example

$$if\ beam\_type = rolled \quad then \quad beam\_depth \in [0.1, 1]$$
$$if\ beam\_type = plate\_girder \quad then \quad beam\_depth \in [1, 15]$$

   This constraint describes the expected depth of the beam according to its type.

2. Numeric constraints that use discretization operators such as trunc and round or those that contain integer variables such as the variable Nft in

$$B = (Nft - 1) * Eft$$

   This constraint describes the width of the structure as a function of the number of joists and the joist spacing.

3. Mixed constraints that link discrete value tuples or interval values to numeric constraints such as discontinuous functions:

$$\begin{aligned} Q_n = 320 * (1 + \frac{H_0}{350}^2) \quad & Q_n \geq 900 \\ Q_n = 900 \quad & Q_n < 900 \end{aligned}$$

   This constraint describes the snow load (Q_n) on a flat roof as a function of the altitude (H_0) at which a structure is built in Switzerland in N/m^2. The minimum value for Q_n is 900 N/m^2.

Discrete constraints with interval values can be treated by the discrete refine operator presented in Figure 2 if a discretization of all numeric variables involved is also defined. A discretization of a numeric domain involves defining landmarks. The intervals between landmarks are taken as discrete values. For variable *beam_depth* in the constraint given above, the intervals $[0.1, 1]$ and $[1, 15]$ are two discrete values covering the variable domain $[0.1, 15]$. The constraint can thus be represented internally as a discrete constraint and treated by a discrete refine operator. Before activating refine operators, *transformation functions* convert numeric variable labels into discretized sets of values. After the call to refine, resulting labels are translated back into sets of interval values in order to be available for other numeric constraints.

Since refine operators may use different value representations and some constraints of different type might share variables, the notion of an *approximate domain* becomes important. An approximate domain over the variable domain D allows the approximation of the original results for a variable [Benhamou, 1996]. Approximate domains are necessary, for example, to combine results for a variable coming from a floating-point operator and another one from a rational operator. Before and after calls to refine operators, transformation steps might be necessary in order to combine approximate results.

Numeric constraints with discretization operators are approximated by a set of numeric constraints according to Table 1. The numeric refine operator treats the constraint as all other numeric constraints. Additionally, if a new interval value $[a, b]$ with $a, b \in \mathbb{R}$ is derived for variable N_g, it is rounded to the next integer interval applying the transformation function $f([a, b]) = [\lceil a \rceil, \lfloor b \rfloor]$.

Mixed constraints associating discrete values with numeric constraints Constraints in which discrete tuples are related to numeric constraints, as for example given in the following piecewise defined function are propagated during search.

$$\begin{aligned} Q_n = 320 * (1 + \frac{H_0}{350}^2) \quad & Q_n \geq 900 \\ Q_n = 900 \quad & Q_n < 900 \end{aligned}$$

The strategy is to enumerate first discrete values of variables in such constraints in order to define those subspaces where the numeric part of the constraint is valid.

Table 1: *A list of operators and their approximations. The operand r is a real, i,p, and q are integers.*

Operator	**Name**	**Approximation**
$i = \lceil r \rceil$	ceiling	$r \leq i < r + 1$
$i = \lfloor r \rfloor$	floor	$r - 1 < i \leq r$
$i = round(r)$	round	$r - 1/2 \leq i < r + 1/2$
$i = trunc(r)$	trunc	$r - 1 < i \leq r$ if $r \geq 0$
		$r \leq i < r + 1$ if $r < 0$
$i = mod(p, q)$	mod	$r/q = div(r, q) + i$
$i = div(p, q)$	div	$i = trunc(r/q)$

2.2 Search techniques for CSPs

Many search techniques are available for discrete CSPs. Most of them are based on *backtrack search*, a method which solves CSPs by instantiating variables and testing partial instantiations against relevant constraints (consistency checks). The search space of a CSP can be represented as a tree, in which each node represents a particular state of the search. Edges linking the nodes are transitions between two states. A node in the search tree of a CSP corresponds to a partial instantiation of the variables and an edge between two nodes represents the choice of a value for the next variable. At a given node, a next variable is selected for instantiation from a given order of the variables (order X_1, X_2, X_3, X_4 in Figure 1) and a new node is created as successor for each value in the domain of this variable. If a node is reached where the tuple of values becomes inconsistent, no future choices will lead to a solution and therefore, the node is pruned. In Figure 1, node (aaa) is pruned and its subbranch neglected because it does not satisfy the constraint between variables X_1 and X_3. Backtracking still suffers from *thrashing* [Mackworth, 1977]. For example in Figure 1, the inconsistency between $X_1 = a$ and $X_3 = a$ is rediscovered during search.

Search is improved through removing inconsistent combinations of values either prior to or during search. During backtrack search, a refine step that achieves local consistency for one variable in a pair is applied to the future variables. Each instantiation is propagated further to not-yet-assigned variables. The amount of local consistency performed at each node determines the type of search algorithm.*Forward checking* [Haralick and Elliott, 1980], for example, applies a refine step to all the neighbors of an instantiated variable, that is, to all those future variables which are connected to the current variable by a constraint. Another algorithm, MAC, *maintaining arc-consistency* [Sabin and Freuder, 1994], [Bessière and Régin, 1996] applies local consistency to the entire CSP after each domain reduction (Figure 4).

In addition to local consistency checks, heuristics such as variable ordering can be applied. Branches in the search tree that are likely to fail are chosen first in order to prune as much as possible of the search tree (first-fail principle). For discrete binary CSPs, a first-fail principle based on the number of neighbors it is connected to through constraints (the degree) and minimum domain size has proven to be efficient for a number

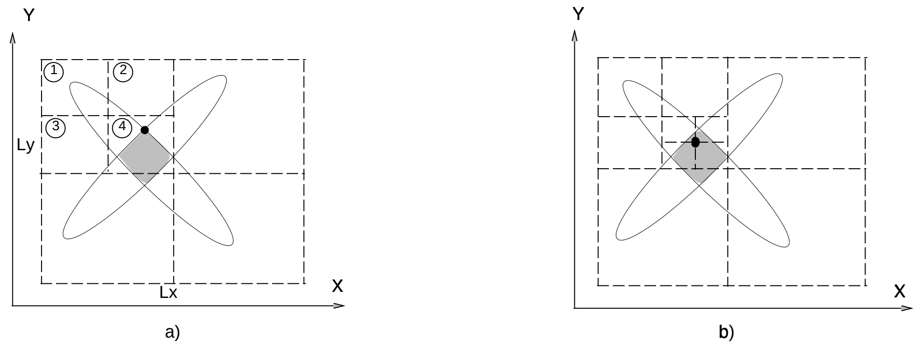

Figure 5: *Search in numeric CSPs: a) interval method b) backtrack search instantiating the mid-value of an interval.*

of applications [Haralick and Elliott, 1980]. A more sophisticated measure is a combination of domain size and maximum degree heuristic, which has been tested in different combinations in [Bessière and Régin, 1996]

While there are a great deal of research results related to solving discrete CSPs, reliable numeric CSP resolution has proven to be more difficult. This is partly due to the infinitely many possible values variables may take in their domains. Search methods using interval analysis [Moore, 1979] are often used for nonlinear optimization and for resolution of equation systems in order to produce single solutions. In a typical branch and bound approach, the constraint region is subdivided into a finite number of cubic subregions (sets of intervals) that are then tested for the optimum using interval analysis. If a test fails on a cubic region, it is guaranteed not to contain a solution and can be discarded.

In Figure 5a, interval splitting is applied to find the maximum in Y of the shaded region. Boxes 1,2, and 3 can be discarded by interval analysis. Further splits in box 4 are necessary to find a reasonable approximation of the optimum. [Van Hentenryck et al., 1995] has developed a prototype called Newton and its successor, Numerica [Van Hentenryck et al., 1997], enhancing interval methods with consistency techniques. The system consists of a branch and bound algorithm applying box-consistency to improve assignment of variable labels. [Hyvönen, 1992] uses an interval analysis approach called tolerance propagation. This algorithm refines solution sets in a top-down manner in order to create a lattice of solution sets.

3 A generic search algorithm with refine operators for mixed constraints

A generic way to solve a mixed CSP is to exploit similarities between search in discrete and numeric CSPs. In general, numeric CSPs are solved using interval methods. These methods create a tree of intervals through bisection (Figure 5b) and then intervals are pruned using interval analysis. Discrete CSPs, on the other hand, are solved by enumerating combinations of values for variables. We combine these two methods and propose a systematic enumeration of values as follows:

- bisection generates a tree of intervals from which a tree of midpoint values is obtained.
- The definition of midpoint value depends on the domain representation: discrete variable labels can be represented by a set of integer intervals, e.g. the label $\{a, b, d\}$ of a discrete variable with domain $\{a, b, c, d\}$, ordered by a:1, b:2,c:3, d:4, is represented by the list of intervals $[1, 2], [4, 4]$ The midpoint value for a discrete interval $[a, b]$, $a, b \in \mathbb{N}$ is then computed by the formula $floor(\frac{a+b}{2})$. Continuous variable labels are represented by a set of real intervals. The midpoint value is thus the midpoint of one of the intervals $[c, d]$ with $c, d \in \mathbb{R}$ given by the formula $\frac{c+d}{2}$.
- the tree of midpoint values is searched in a depth-first manner starting with the first interval in a variable label. However, the locally consistent intervals are kept for each variable. If a consistency check fails, the algorithm can directly continue with the next interval in a label.

When the solution space consists of few contiguous regions, a solution is found rapidly. The algorithm **split** instantiates discrete and numeric variables successively to a single value m, the midpoint, of the current interval I of their label L and checks the obtained assignments against the constraints in **treat-value**. When a consistency check fails, the algorithm first backtracks to the interval to the left of the failed value $[left(I), m)$ and then to the interval to the right $(m, right(I)]$ and searches them recursively. Domain splitting continues until a given distance w between adjacent values is reached. w is 1 for discrete domains and is chosen for numeric domains such that a reasonable number of values in the label are checked. An example of a simple backtrack algorithm with value instantiation is presented in Figure 5b. The dot indicates one possible midpoint solution. Here a solution is found after three levels of splitting, i.e. in the worst case the centers of sixteen cubes have to be tested.

This generic search algorithm is enhanced by consistency techniques that propagate instantiated values to the future variables. The function **check** in Figure 6 applies forward checking to prune neighbor labels using the refine operators defined in section 2.1. It takes the set of labels L, the constraint set $\mathcal{C}$ and the already obtained assignments $\mathcal{S}$ as input and returns the changed labels and an ok-status indicating if an inconsistency has been found in the assigned variables and the constraints or not. If the problem is still consistent (ok is true), the algorithm continues calling **recursive-search** on the next variable, otherwise the last assignment added is removed from the solutions and the labels, which may have been changed by **check**, are reset to the state before the last assignment (**unwind-labels**). Mixed constraints of type three are treated by adding them to the constraint set when a new variable has been assigned a value (**add-mixed-constraints**). They are removed again when the algorithm backtracks on this variable.

Additionally, this algorithm integrates dynamic ordering to improve search (function **reorder-variables**). Before the next variable is instantiated, variables are reordered taking into account the following strategies:

- Value enumeration for discrete variables may also refine numeric variable labels due to mixed constraints of type 1 and 2 presented in section 2.1.

```
**procedure** **recursive-search**$(\mathcal{V}, \mathcal{L}, \mathcal{C}, width, \mathcal{S})$
**begin**
  **if** all variables of $\mathcal{V}$ are assigned in $\mathcal{S}$ **then**
    **return** on first solution: $\mathcal{S}$
  **else**
    $\mathcal{V} \leftarrow$ **reorder-variables**$(\mathcal{V})$
    $X \leftarrow$ first variable in $\mathcal{V}$ unassigned in $\mathcal{S}$
    $I_X \leftarrow$ next interval from $L_X \in \mathcal{L}$
    **split**$(I_X, X, \mathcal{V}, \mathcal{L}, \mathcal{C}, width, \mathcal{S})$
**end**

**procedure** **split**$(I_X, X, \mathcal{V}, \mathcal{L}, \mathcal{C}, width, \mathcal{S})$
**begin**
  $m \leftarrow$ **midpoint**$(I_X)$
  **treat-value**$(m, X, \mathcal{V}, \mathcal{L}, \mathcal{C}, \mathcal{S}, width)$
  **if** $right(I) - left(I) > width$ **then**
    **split**$(X, [left(I_X), m), \mathcal{V}, \mathcal{L}, \mathcal{C}, w)$
    **split**$(X, (m, right(I_X)], \mathcal{V}, \mathcal{L}, \mathcal{C}, w)$
  **fi**
**end**

**procedure** **treat-value**$(m, X, \mathcal{V}, \mathcal{L}, \mathcal{C}, \mathcal{S}, width)$
**begin**
  $oldLabels \leftarrow \mathcal{L}$
  $L_X \leftarrow m,\ L_X \in \mathcal{L}$
  $\mathcal{S} \leftarrow$ add $X = m$ to $\mathcal{S}$
  $\mathcal{C} \leftarrow$ **add-mixed-constraints**$(\mathcal{C}, X = m)$
  $newLabels \leftarrow$ **check**$(\mathcal{L}, \mathcal{C}, \mathcal{S})$
  **if** $newLabels \neq \emptyset$ **then**
    **recursive-search**$(\mathcal{V}, newLabels, \mathcal{C}, width, \mathcal{S})$
  **fi**
  $\mathcal{S} \leftarrow$ remove $X = m$ from $\mathcal{S}$
  $\mathcal{C} \leftarrow$ **remove-mixed-constraints**$(\mathcal{C}, X = m)$
  **unwind-labels**$(newLabels, oldLabels)$
**end**
```

Figure 6: *Search algorithm for a mixed CSP with interval splitting on backtracking.*

- Discrete value combinations occurring in a discontinuous function impose further numeric constraints on the problem.

Discrete Variables occurring in discontinuous constraints should be enumerated first since they add numeric constraints to the problem. Then, other discrete variables are enumerated subsequently constraining numeric variables through mixed constraints. Variables are ordered according to the max degree criterion (section 2.2) first and the secondary criterion of min domain size. For discrete variables, the ordering max degree + min domain (first fail principle) and for the continuous variables, the ordering max degree + max domain performs well on the example presented in the following section. In this example, we choose to enumerate all values for discrete variables and to find only one consistent value for numeric variables. The reason is that this example is underconstrained and an enumeration of values in the numeric labels results in very similar solutions at a distance w of the previous solution. The algorithm in Figure 6 shows the general principle of value enumeration over discrete and numeric variables and can be changed easily to avoid a complete enumeration of the numeric variables.

In numeric domains, pruning the intervals from inconsistent values using local consistency is important since interval size and the number of splitting iterations can be reduced drastically. Furthermore, regions become constrained during instantiation; local consistency algorithms for numeric domains, such as the one described in Section 2.1 have improved computation times. This is especially true if many variables are already instantiated. At that point, inconsistencies are easily detected by forward checking since the search space has become highly constrained. Our approach is very generic in that the search engine is the same for discrete and numeric variables and that special *refine operators* can be associated to each constraint type (discrete, numeric, and mixed) to prune variable labels. When a variable has been instantiated, the associated refine operator propagates the choice to its neighbor variables (implemented in the call to **check**).

4 A Full-Scale Example

In this section, the advantages of using local consistency techniques in engineering applications are demonstrated. The algorithm described in this paper

- is capable of providing support for full-size tasks,
- gives good approximations of solution spaces found in typical engineering tasks and can thus be used to show, for example, the influence of changes in constraints
- enhances other search algorithms used in computer-aided engineering through removing local inconsistencies early in the search process.

This algorithm provides a different kind of support than those offering global consistency, for example [Sam-Haroud and Faltings, 1996]. Global consistency ensures that each value in the domain of a variable can be found in at least one solution and is thus, a much stronger condition. Solution spaces calculated in this way provide greater engineering support. However, global consistency algorithms have high polynomial time complexity and as a result, they can only be applied to examples having a small number (usually less than twenty) variables.

We present a simple example on the design of a single story steel building (Figure 7). Non-rigid connections (pin connections) are envisaged. The building has to resist to external loads such as wind and snow as well as internal loads caused by an overhead crane. In order to resist horizontal loads, a longitudinal and transversal bracing system is necessary (Figure 8). Constraints from the Swiss Code (SIA 161) that have to be respected are: bending of joist (Ft) and girder (T), biaxial bending of girt (Fl and Ff), buckling of columns (Cl and Cf) and axial forces in diagonals (Dlt, Dll, Dtt, Dtl) of the bracing systems. The values of snow and wind load are provided by the Swiss Code SIA 160. Characteristics of steel sections available in Switzerland are introduced in form of tables. The cross-sectional shape of each element is chosen by the user. In this example, girders, columns, and joists are I-shaped , diagonals in the bracing system are angles, and the girts are channels. Geometric constraints on volume, surface, width, length, height, width-length ratio, spacing of the elements (E, Eft, Ecf, Efl), frontal (Lpf) and lateral door (Lpl) width are also considered. Unary constraints are introduced in order to restrict value domains of certain variables. The complete list of variables and constraints is given in the Appendix.

Typical input parameters are snow and wind load, the altitude at which the building is situated and the minimal surface it occupies. Applying local consistency methods at this point results in reduced domains for the numeric variables. However, there is no effect on the choice of the cross sections since there are too many degrees of freedom. When additional constraints are imposed on variables, their effect on other variables is observed through different results for solution spaces. Typical user constraints are those on the dimensions of the building (length, width, and height) and the spacing between elements. These parameters influence the choice of the cross section for the girts, joists, columns, and frames. Although a large number of elements guarantee the stability of the building, the resulting design might cost more and be less aesthetic. We present two examples of additional user constraints and solution spaces in Table 2. Constraint set 1 restricts the building dimensions and the spacings E, Eft, Efl, and Ecf, and constraint set 2, imposes constraints on the number of elements (N, Nft, Ncf, and Nfl). Both examples were executed with the lateral and frontal wind loads set to 800 N/m^2 and 250 N/m^2 and with a wall cladding self weight of 500 N/m^2. The load of the overhead crane was set to 2000 kN.

The average execution time was four minutes on a two-processor Sun Sparc for local consistency with additional constraints posted by the user. These examples involve over 100 variables. The prototype implementation is written mostly in Lisp with some code in Maple (implementation of the refine operators) and has not been optimized.

In both solution spaces, the choices of cross-sections for the girder and the column were reduced from around ten possibilities to one and two respectively. The spacings and the number of elements are also reduced. Solutions are at the top of the allowable sections listed in the Appendix. Typically solution spaces are not bounded from above since heavier sections are almost always feasible. Engineers naturally select lighter member from the solution set. Although some values within these intervals may not be part of the exact

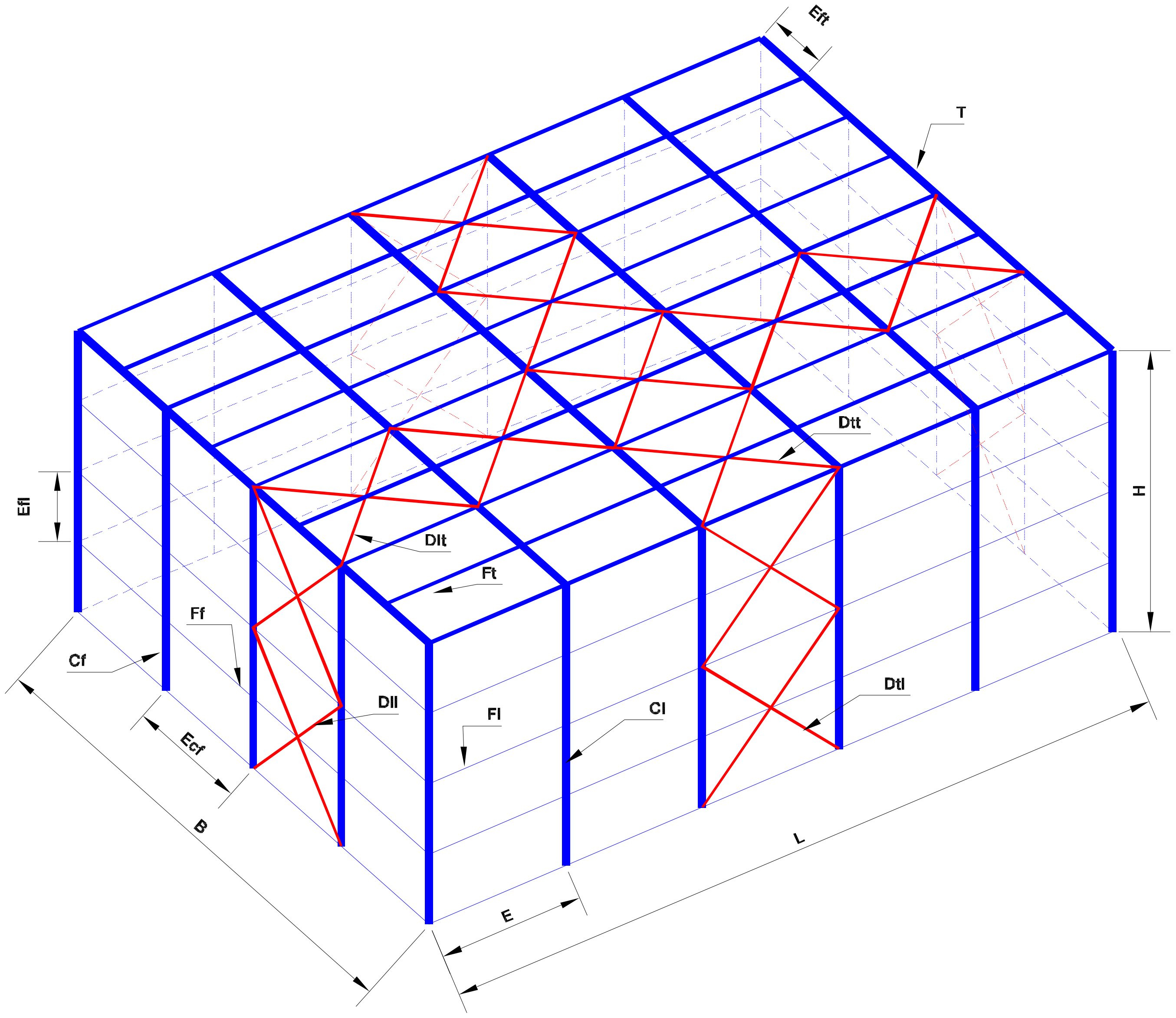

Figure 7: *Design of a single story steel building.*

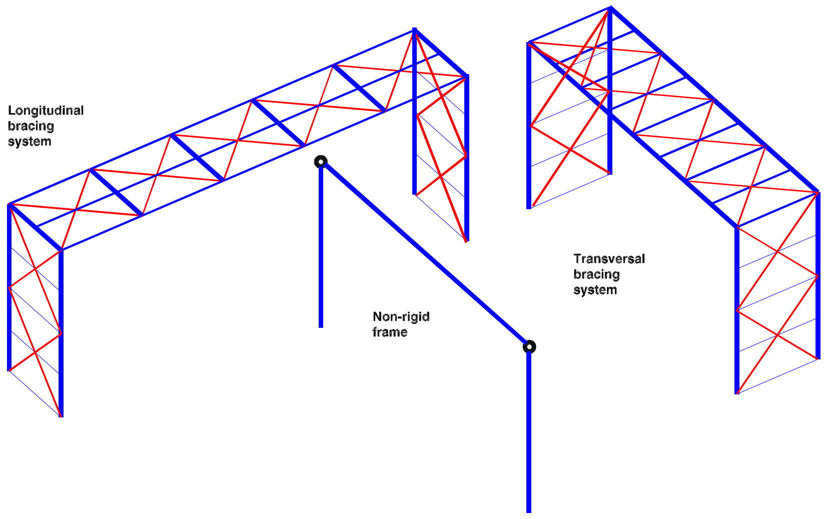

Figure 8: *An example of a longitudinal and transversal bracing system for resisting horizontal loads.*

solution space, the presented spaces provide good approximations. Also variations in the solution spaces with changes in user-posted constraints provide indications for sensitivity studies.

Such solution spaces can also be the input to algorithms that provide higher levels of consistency. In any case, when a solution is chosen it needs to be checked against all constraints before it is adopted. A solution can for example be derived for a structure with $L = 36$, $B = 24$, $H = 10$ at an altitude of 900 meters using the following cross sections: $T = HEB300$, $Cl = HEB300$, $Cf = HEB300$, $Ft = IPE200$, $Fl = UNP280$, $Ff = UNP280$, and $LNP100{*}10$ for all diagonals. The number of girts in the longitudinal bracing system is three (Cvl) and one in the transversal bracing system. This corresponds to the third possibility shown in Figure 9.

5 Conditional CSPs

Support for structural design of, for example, this industrial building can be enhanced by considering different configuration possibilities of the bracing system (Figure 8). Some designs include only a transversal bracing system when girders are attached with moment-connections to the columns and when the frames are rigid enough to resist horizontal loads. Under these circumstances, the constraints related to the longitudinal bracing system are not relevant. This means that the constraints in A3.2.7, A3.2.9 and A3.2.11 (see appendix) are not needed and that A3.2.1 and A3.2.2 are replaced by another set of constraints. The feasible height might fall into a range of values that allows for a simpler system without longitudinal bracing. When the values for height are refined during calculation of the solution space, such a simplification may be revealed only during execution of the

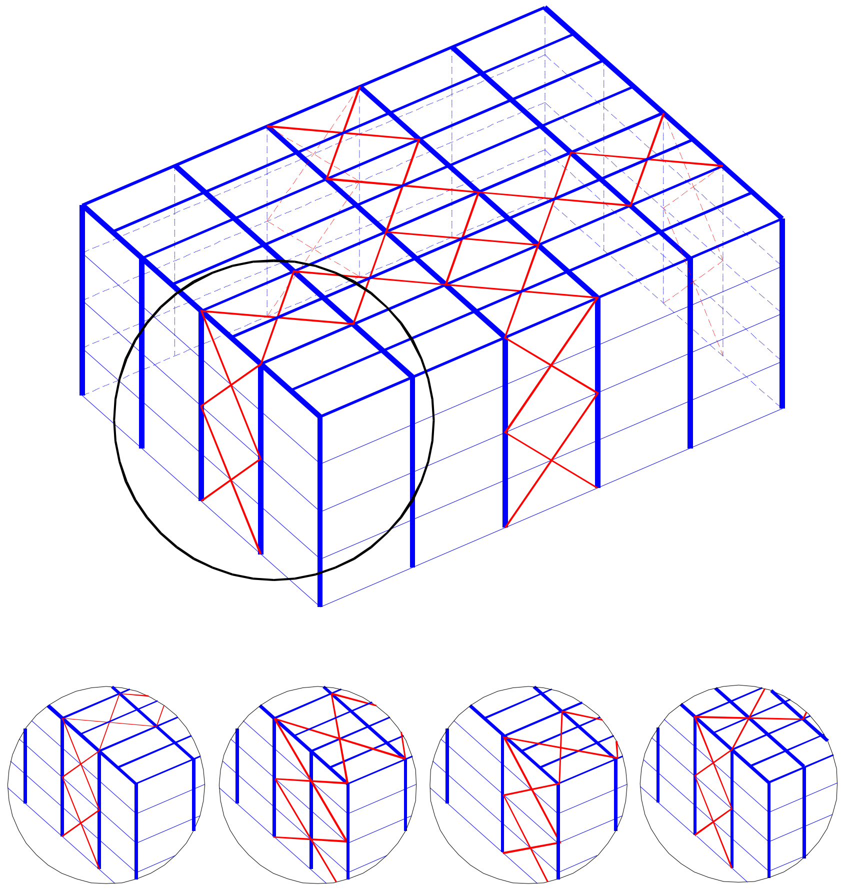

Figure 9: *Different solutions for the longitudinal and transversal bracing system.*

Table 2: *Users may post additional constraints to reduce the size of the solution space. Where applicable, units are meters.*

Additional User Constraint Set 1	
altitude H_0	800
minimal surface S_{min}	200
width B	[10,40]
height H	[8,10]
frame spacing E	[4,5]
joist spacing Eft	[4,5]
joist spacing Efl	[4,5]
joist spacing Ecf	[4,5]
Solution space 1	
Nb of joists Nft	[3,4]
Nb of windcolumns Ncf	[3,4]
Nb of girts Nfl	3
Girder T	HEB300
Longitudinal girt Fl	UNP240/280/320
Transversal girt Ff	UNP240/280/320
Column Cl	HEA240,HEB300

Additional User Constraint Set 2	
altitude H_0	1000
minimal surface S_{min}	400
length L	[40,200]
height H	[10,20]
Nb of frames N	[1,9]
Nb of joists Nft	[1,7]
Nb of windcolumns Ncf	[1,10]
Nb of girts Nfl	[1,7]
Solution space 2	
frame spacing E	[5,7.19]
joist spacing Eft	[1.16,5]
joist spacing Efl	[1.67,3.45]
joist spacing Ecf	[3.45,10.19]
Nb of frames N	[7,9]
Nb of joists Nft	[3,7]
Nb of windcolumns Ncf	[2,4]
Nb of girts Nfl	[4,7]
Girder T	HEB300
Column Cl	HEA240
	HEB300

algorithm. This phenomenon is called conditional variable activation; complex engineering tasks often demonstrate such behavior.

A model of *conditional constraint satisfaction* (CCSP) [Mittal and Falkenhainer, 1990] has been introduced to adapt CSPs to changing environments as illustrated by the constraints on the bracing system. The standard CSP model is extended to reason on the *presence* of variables in a solution. A variable in a CCSP can be active or not in a solution; i.e. $\forall X_i \in \mathcal{V} : \mathit{active}(X_i) \leftrightarrow X_i = x$ with $x \in D_i$. Typical constraints restricting value combinations of variables are called *compatibility constraints* in this model. We extend the original definition of CCSP by allowing numeric and discrete compatibility constraints. In contrast to a static CSP, a compatibility constraint in a CCSP is only *relevant* to a problem if all the variables of this constraint are active. It follows that a compatibility constraint is trivially satisfied if at least one of its variables is not active. This is made explicit by considering a compatibility constraint only in those parts of the search space in which all variables of the constraint are active. Additionally, a CCSP can post constraints on a variable's activity in a given context of value assignments. Such an activity constraint has the form:

$$C_{X_1,\ldots,X_j} \rightarrow \mathit{active}(X_k)$$

where $C_{X_1,\ldots,X_j}$ is a single constraint, which expresses an *activation condition* under which the variable X_k becomes active. An activity constraint is *satisfied* by a set of values $\{X_1 = x_1, \ldots, X_j = x_j\}$ if $\neg C_{X_1,\ldots,X_j}(x_1,\ldots,x_j) \vee \mathit{active}(X_k)$ is true.

In our model, we allow the formulation of an activation condition as a set of value assignments (a discrete constraint) and as a numeric constraint. A CCSP thus consists of

- A set of variables $\mathcal{V}$ representing all variables that may potentially become active and be part of a solution. Each variable $X_i \in \mathcal{V}$ has associated a domain $D_i \in \mathcal{D}$ representing the set of possible values for the variable.
- A non-empty set of *initial variables* $V_I \subseteq \mathcal{V}$. These variables have to be part of every solution.
- A set of *compatibility constraints* on subsets of $\mathcal{V}$ representing allowed value combinations for these variables.
- A set of *activity constraints* on subsets of $\mathcal{V}$ specifying constraints between the activity of a variable and possible values of problem variables.

The goal of a CCSP is to find all solutions, where a solution S is

1. an assignment of values to a set of variables such that S satisfies all constraints in $C^C \cup C^A$.
2. minimal – there is no solution S' satisfying all constraints such that $S' \subset S$.
3. complete – all variables of V_I are assigned in S

Minimal solutions of a CCSP contain only those assignments for which there exist no other assignment that has fewer identical variable-value pairs satisfying the same constraints. Starting from the set of initially active variables V_I, a CCSP incrementally defines spaces where different variables are active and values are only assigned to the active variables.

In order to illustrate the formulation of a CCSP, the constraints associated with designing a steel structure are extended as follows. A new discrete variable *cgt* standing for column-girt connection is introduced with the domain $[moment\_connection, pin\_connection]$ as well as a numeric variable Hd representing the horizontal displacement of the structure. The following activity constraints define different bracing systems:

$$cgt = pin\_connection \overset{ACT}{\rightarrow} Dlt \tag{1}$$
$$cgt = pin\_connection \overset{ACT}{\rightarrow} Dll \tag{2}$$
$$cgt = moment\_connection \overset{ACT}{\rightarrow} Hd \tag{3}$$
$$Hd \geq f(H) \overset{ACT}{\rightarrow} Dlt \tag{4}$$
$$Hd \geq f(H) \overset{ACT}{\rightarrow} Dll \tag{5}$$

These constraints ensure that the longitudinal bracing system represented by the variables *Dlt* and *Dll* is only computed if either *cgt* is not a moment connection or

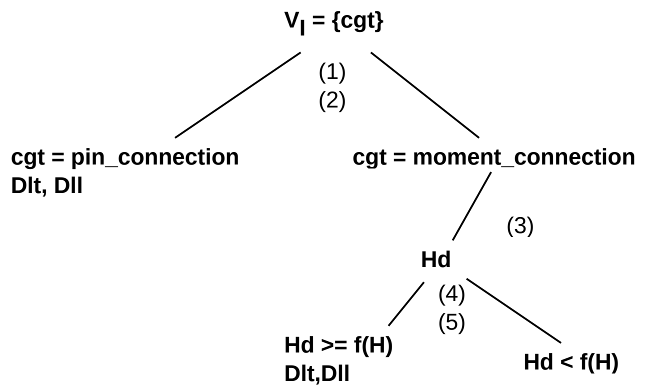

Figure 10: *The constraints and variables added dynamically are represented in a combination tree. The numbers in brackets refer to the constraints in the example of the bracing system.*

the horizontal displacement is larger than a given function on the building height. As a consequence, the compatibility constraints over the longitudinal bracing system only become relevant when the variables Dlt and Dll are active.

In the original CCSP algorithm by [Mittal and Falkenhainer, 1990], the condition of an activity constraint was restricted to a set of value assignments. This restriction does not allow treatment of problems where:

1. The activation condition is a discrete constraint with several value combinations
2. The activation condition is a numeric constraint defining infinitely many value combinations.

Activating new variables based on partial solutions is thus not an efficient method for solving a mixed CCSP. We propose the generation of subspaces where variables are active through consideration of different combinations of activation conditions. The activity constraint that introduces a longitudinal bracing system, leads to two subspaces, one where the traversal bracing system exists and the condition on the width is satisfied and one where the condition is not satisfied. Solutions to a CCSP are found in the combinations of all these subspaces. This combination can be represented in a tree where for each constraint two sub-branches are added to each node, one with the condition satisfied and the variable active and one with the complement of the condition satisfied. More detailed information on generalized CCSPs is given in [Gelle, 1998].

In Figure 10, each node shows only newly added variables and conditions. Trivially inconsistent subspaces are not shown. In the leaf nodes of this tree are represented the spaces where solution spaces for the CCSP are found. In our example, the leaf spaces P_i with additional variables $\mathcal{V}_i$ and constraints $\mathcal{C}_i$, $i = 1, 2, 3$ are:

$$\begin{array}{lll} \mathcal{P}_1: & \mathcal{V}_1 = \{Dlt, Dll\} & \mathcal{C}_1 = \{cgt = pin\_connection\} \\ \mathcal{P}_2: & \mathcal{V}_2 = \{Hd, Dlt, Dll\} & \mathcal{C}_2 = \{cgt = moment\_connection, Hd \geq f(H)\} \\ \mathcal{P}_3: & \mathcal{V}_3 = \{Hd\} & \mathcal{C}_3 = \{cgt = moment\_connection, Hd < f(H)\} \end{array}$$

Problem set three describes all solutions with a horizontal displacement smaller than the given limit. Thus, those solutions do not require a bracing system. Since each space corresponds to a standard CSP, the methods described in the preceding sections can be applied to define solution spaces.

Activity constraints may introduce new variables that are themselves used in the condition of another activity constraint. This dependency can be analyzed in a directed graph. Such an analysis helps to identify i) cycles in activity constraints ii) constraints that are never reached (and thus their variables are never activated) and finally iii) an ordering for activity constraints. The algorithm for generating the solution spaces of a CCSP includes the following steps:

1. Create a directed graph from the set of activity constraints representing the dependencies between the constraints. Each activity constraint as well as the set of initially active variables is a node (respectively a root node) and a directed edge exists between two nodes if the variable activated by the first constraint is found in the condition of the second constraint. The second constraint is dependent on the first one and the arrow points to the dependent node.

2. Eliminate cycles in the original graph by clustering all nodes in such cycles into a super-node. Instead of identifying cycles directly, strongly connected components are identified in the graph. In a strongly connected component, there is a path between each pair of nodes. In contrast to cycle identification, this can be done in time linear in the number of nodes and arcs. A reduced graph is built by collapsing each strongly connected component into one node and by linking those components which contain each node that was linked by an edge in the original graph. The reduced graph is acyclic by construction.

3. Find a partial order on the resulting acyclic graph, traverse the graph in this order, and apply the activity constraints. On the reduced graph a strict partial order can be established. A relation is a strict partial order if it is asymmetric, transitive, and irreflexive. Using a longest path algorithm, we attribute a length to each path so that an activity is only executed if all the variables in its condition are active. Ordering the nodes according to the increasing distance from the root node, gives us an order in which the activity constraints can be applied safely. If a node has been collapsed, i.e. contains several activity constraints, each of them has to be tested for applicability.

Consider the directed graph in Figure 11. In the first example, only the variables X_1, X_2 are initially active. Activity constraints are $X_2 = c \overset{ACT}{\rightarrow} X_5$,$X_5 = i \overset{ACT}{\rightarrow} X_3$,$X_1 = b \overset{ACT}{\rightarrow} X_3$ $X_3 = e \overset{ACT}{\rightarrow} X_4$. The directed graph shows dependencies between the constraints. The initially active variables form the root node. The numbers in bracket give the longest path from the root node to a node. They indicate a partial ordering on the activity constraints. This order is converted to a full order and the constraints are applied in the order $X_1 = b \overset{ACT}{\rightarrow} X_3$, $X_2 = c \overset{ACT}{\rightarrow} X_5$,$X_5 = i \overset{ACT}{\rightarrow} X_3$, $X_3 = e \overset{ACT}{\rightarrow} X_4$. The application of the first activity constraint $X_1 = b \overset{ACT}{\rightarrow} X_3$ to the space containing the active variables X_1, X_2 results in two spaces, one in which X_1, X_2, X_3 are active and X_1 is reduced to b, and a second, in which only X_1, X_2 are active but X_1 cannot take the value b. Each

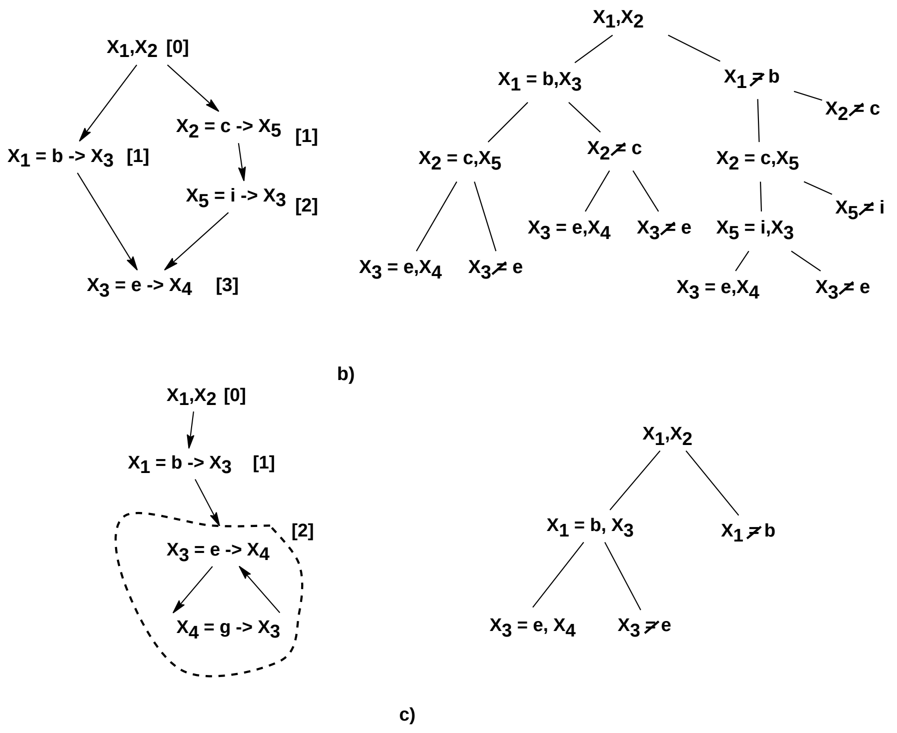

Figure 11: *Dependencies in a DCSP represented as directed acyclic graph. The right hand sides show how problem spaces are combined. At each node only the additional variables and constraints are shown.*

activity constraint is applied to the corresponding subspaces and may lead to a doubling of the number of subspaces. Local consistency can detect inconsistencies in the subspaces and thus helps reducing the number of subspaces.

6 Conclusions

Properly formulated local consistency techniques provide useful support for engineering tasks when many interdependent variables of discrete and numeric nature are involved. The local consistency techniques described in this paper are able to approximate globally consistent spaces rapidly and with a reasonable degree of accuracy for constraint networks that exceed twenty variables. However, local consistency algorithms are not completely reliable. When accuracy is essential, our approach provides an effective filter for infeasible values such that its results can be the input to subsequent search or optimization methods. Our extension to conditional CSPs allows a systematic enumeration of solution spaces, which leads to a clarification of the overall complexity of solution-space sets. As in the context of standard CSPs, local consistency is applied to remove entire spaces that are inconsistent. An example of steel-structure construction motivates the results of this paper and also demonstrates the need for conditional CSPs. The proposed methods will enhance performance and quality of future computer-aided engineering systems.

Acknowledgements

This research was funded by the Swiss National Science Foundation under contract no. 21-39379.93 and no. 20-45416.95.

References

[Benhamou, 1996] Benhamou, F. (1996). Heterogeneous constraint solving. In Hanus, M. and Rodríguez-Artalejo, M., editors, *Algebraic and Logic Programming, 5th International Conference, ALP'96*, volume 1139 of *lncs*, pages 62–76, Aachen, Germany. Springer.

[Benhamou et al., 1994] Benhamou, F., McAllester, D., and Hentenryck, P. V. (1994). CLP(intervals) revisited. In Bruynooghe, M., editor, *Logic Programming - Proceedings of the 1994 International Symposium*, pages 124–138, Massachusetts Institute of Technology. The MIT Press.

[Bessière and Régin, 1996] Bessière, C. and Régin, J.-C. (1996). Mac and combined heuristics: two reasons to forsake fc (and cbj). In Freuder, E. and Jampel, M., editors, *Principles and Practice of Constraint Programming*, volume 1118 of *Lecture Notes in Computer Science*. Springer.

[Davis E., 1987] Davis E. (1987). Constraint propagation with interval labels. In *Artificial Intelligence 32*.

[Dechter, 1990] Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312.

[Faltings, 1994] Faltings, B. (1994). Arc consistency for continuous variables. In *Artificial Intelligence 65(2)*, pages 85–118.

[Faltings and Gelle, 1997] Faltings, B. and Gelle, E. (1997). Local consistency for ternary numeric constraints. In *Proc. 11th Int. Joint Conf. on Artificial Intelligence, IJCAI-97*, pages 392–397.

[Freuder, 1982a] Freuder, E. C. (1982a). A sufficient condition for backtrack-bounded search. *A.C.M.*, 32(4):755–761.

[Freuder, 1982b] Freuder, E. C. (1982b). A sufficient condition for backtrack-free search. *A.C.M.*, 29(1):24–32.

[Gelle, 1998] Gelle, E. (1998). *On the generation of locally consistent solution spaces in mixed dynamic constraint problems*. PhD thesis, Swiss Federal Institute of Technology, EPFL.

[Haralick and Elliott, 1980] Haralick, R. M. and Elliott, G. L. (1980). Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313.

[Hyvönen, 1992] Hyvönen, E. (1992). Constraint reasoning based on interval arithmetic. the tolerance propagation approach. In *Artificial Intelligence*.

[Kondrak, 1994] Kondrak, G. (1994). A theoretical evaluation of selected backtracking algorithms. Technical Report TR-94-10, University of Alberta.

[Kumar, 1992] Kumar, V. (1992). Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44.

[Lhomme, 1993] Lhomme, O. (1993). Consistency techniques for numeric CSPs. In *IJCAI-93*, pages 232–238.

[Lottaz et al., 1999] Lottaz, C., Clément, D., Faltings, B., and Smith, I. (1999). Constraint-based support for collaboration in design and construction. *Journal of Computing in Civil Engineering*, 13(1):23–35.

[Mackworth, 1977] Mackworth, A. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8.

[McGregor, 1979] McGregor, J. J. (1979). Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250.

[Mittal and Falkenhainer, 1990] Mittal, S. and Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. In Dietterich, Tom; Swartout, W., editor, *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 25–32. MIT Press.

[Montanari, 1974] Montanari, U. (1974). Networks of constraints: Fundamental properties and applications to picture processing. *Inform. Sci.*, 7:95–132.

[Moore, 1979] Moore, R. E. (1979). *Methods and Applications of Interval Analysis.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[Prosser, 1993] Prosser, P. (1993). Domain filtering can degrade intelligent backjumping search. In *IJCAI-93*.

[Sabin and Freuder, 1994] Sabin, D. and Freuder, E. (1994). Contradicting conventional wisdom in constraint satisfaction. In Borning, A., editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).

[Sam-Haroud and Faltings, 1996] Sam-Haroud, D. and Faltings, B. (1996). Consistency techniques for continuous constraints. In *Constraints*, volume 1, pages 85–118.

[Tinelli and Harandi, 1996] Tinelli, C. and Harandi, M. (1996). Constraint logic programming over unions of constraint theories. *Lecture Notes in Computer Science*, 1118:436–450.

[Tsang, 1993] Tsang, E. (1993). *Foundations of Constraint Satisfaction.* Academic Press, London.

[van Beek and Dechter, 1995] van Beek, P. and Dechter, R. (1995). On the minimality and global consistency of row-convex constraint networks. *Journal of the ACM*, 42(3):543–561.

[Van Hentenryck et al., 1992] Van Hentenryck, P., Deville, Y., and Teng, C.-M. (1992). A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3):291–321.

[Van Hentenryck et al., 1995] Van Hentenryck, P., McAllester, D., and Kapur, D. (1995). Solving polynomial systems using a branch and prune approach. *SIAM Journal of Numerical Analysis.* (Accepted). (Also available as Brown University technical report CS-95-01.).

[Van Hentenryck et al., 1997] Van Hentenryck, P., Micher, L., and Deville, Y. (1997). *Numerica. A modeling language for global optimization.* MIT Press.

[Waltz, 1975] Waltz, D. L. (1975). Understanding line drawings of scenes with shadows. In Winston, P. H., editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill.