Solving Finite Domain Constraint Hierarchies by Local Consistency and Tree Search*

Stefano Bistarelli†
Istituto di Informatica e Telematica
CNR, Pisa, Italy
Stefano.Bistarelli@iit.cnr.it,
Dipartimento di Scienze
Universitá 'G. d'Annunzio', Pescara, Italy
bista@sci.unich.it

Philippe Codognet
Department of Computer Science
University of Paris 6, France
Philippe.Codognet@lip6.fr

H.K.C. Hui and J.H.M. Lee
Department of Computer Science and Engineering
The Chinese University of Hong Kong, Hong Kong SAR, China
{kchui,jlee}@cse.cuhk.edu.hk

Abstract

We provide a reformulation of the constraint hierarchies (CHs) framework based on the notion of *error indicators*. Adapting the generalized view of local consistency in semiring-based constraint satisfaction problems (SCSPs), we define *constraint hierarchy k-consistency* (CH-k-C) and give a CH-2-C enforcement algorithm. We demonstrate how the CH-2-C algorithm can be seamlessly integrated into the ordinary branch-and-bound algorithm to make it a finite domain CH solver. Experimentation confirms the efficiency and robustness of our proposed solver prototype. Unlike other finite domain CH solvers, our proposed method works for both local and global comparators. In addition, our solver can support arbitrary error functions.

Keywords: Constraint Hierarchies, Soft Constraints, Local Consistency.

*The work described in this paper was substantially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region (Project no. CUHK4358/02E).

†Part of this research was carried out while the author was visiting the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong SAR, China

1 Introduction

The Constraint Hierarchy (CH) framework [BFBW92] is a general framework for the specification and solution of over-constrained problems. Originating from research in interactive user-interface applications, the CH framework attracts much effort in the design of efficient solvers in the real number domain [BBS01, HMY96].

The CH framework is an interesting formalism and our goal is to use it also in the finite domain FD environment. The framework put together hierarchies of constraints, error functions and comparator (that consider the error functions locally or globally) and is able to specify in deep detail the preference function the user has in mind. As a motivating example consider the following (a model of the scenario using CH will be given in the next section in Example 2).

Example 1 (A motivating scenario: the sales representative promotion)**.** *Suppose a company wants to promote a sales representative to become the general sales manager from a pool of candidates. The ideal candidate must be a degree holder. (S)he should have at least 5 years of working experience and be able to meet a sales quota of 1 million per annum. Being in a supervisory role, the potential manager should also be familiar with and be able to sell at least 20 products of the company. While the education background is a firm requirement, working experience and sales performance are, in general, considered to be more important than versatility in product range. In considering candidates with similar working experience and sales performance, the last criterion should also be taken into account, although to a lesser degree, to differentiate the best candidate from the rest.*

To extend the benefit of the CH framework to also discrete domain applications, such as timetabling and resource allocation, the paper takes a step towards a general and efficient finite domain CH solver, based on consistency techniques and tree search. Central to the paper (that revises and extends [BCHL03a, BCHL03b]) is the notion of *constraint hierarchy k-consistency* (CH-k-C), defined using error indicators which are structures isomorphic to the structure of a given CH used for storing the error information of the CH problem. We give also an algorithm for enforcing CH-2-C of a CH problem. While classical consistency algorithms [Mac77] aim to reduce the size of constraint problems, our CH-2-C algorithm works by explicating error information that is originally implicit in CH problems. We also suggest ways of utilizing such extracted information to help prune non-fruitful computation in a branch-and-bound searching algorithm, which forms the basis of our finite domain CH solver. We have constructed a prototype of the solver, and performed experiments on a set of randomly generated CH problems that confirm the efficiency and robustness of our proposal.

The rest of the paper is organized as follows. Section 2 provides necessary background definitions. In Section 3, we present an equivalent redefinition of the CH framework using the notion of error indicators and hierarchy problem, which are central in the definition of constraint hierarchy k-consistency and the associated enforcement algorithm in Section 4. In Section 5, we give a constraint hierarchy 2-consistency enforcement algorithm and discuss its complexity. The finite domain CH solver, which has a branch-and-bound backbone, is introduced in Section 6, followed by experimental results in Section 7. Related works are presented in Section 8 before summarizing the major results and shedding light on possible future direction of research in Section 9.

2 Constraint Hierarchies

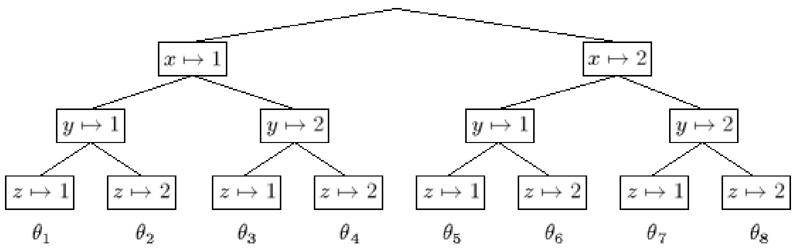
Let D be a constraint domain. A *variable* x is an unknown that has an associated *variable domain* $D(x) \subseteq D$, which defines the set of possible values for x. An *n-ary constraint* c is a relation over D^n. A *labeled constraint* c^s is a constraint c with a *strength* $s \in \{0, \ldots, k\}$. The strengths are totally ordered. Constraints with strength $s = 0$ are *required constraints* (or hard constraints) and those with strength $1 \leq s \leq k$ are *non-required constraints* (or soft constraints). The larger the strength, the weaker the constraint is. In addition, each labeled constraint may be associated with a weight w (for use with the global comparators). A *constraint hierarchy* H is a multiset of labeled constraints. The symbol H_i denotes a set of labeled constraints with strength $s = i$. H_0, the *required level*, denotes the set of required constraints which must be satisfied. $H_1, \ldots, H_k$, the *non-required level*, denote the sets of non-required constraints which can be violated but should be satisfied as much as possible.

$V = \{x, y, z\}$ and $D(x) = D(y) = D(z) = \{1, 2\}$
$H = \{H_0, H_1, H_2, H_3\}$ $H_0 = \emptyset$, $H_1 = \{c_1^1 : x > y, c_2^1 : x = 2\}$ $H_2 = \{c_1^2 : y = 3, c_2^2 : z < y\}$, and $H_3 = \{c_1^3 : z = 1, c_2^3 : x + y + z > 4\}$

Figure 1: An example of constraint hierarchy.

We use an example in Figure 1 to explain CHs in more details. There are three levels in the constraint hierarchy H. There are no required constraints in the required level H_0. However, there are two *strong* constraints c_1^1 and c_2^1 in H_1, two *medium* constraints c_1^2 and c_2^2 in H_2 and two *weak* constraints c_1^3 and c_2^3 in H_3.

A *valuation* $\theta = \{v_1 \mapsto d_1, \ldots, v_n \mapsto d_n\}$ for a set of variables $\{v_1, \ldots, v_n\}$ assigns to each v_i the value $d_i \in D(v_i)$. Let c be a constraint and θ a valuation. The expression $c\theta$ is the boolean result of applying θ to c. We say that $c\theta$ *holds* if $c\theta$ is *true*. An *error function* $e(c\theta)$ measures how well a constraint c is satisfied by valuation θ. The error function returns non-negative real numbers and must satisfy the property: $e(c\theta) = 0 \Leftrightarrow c\theta$ holds. A *trivial error function* is an error function that gives 0 if $c\theta$ holds and 1 otherwise. The value $e(c\theta)$ returned by an error function is an *error value*. We use $vars(c)$ (or $vars(\theta)$) to denote the set of all variables in constraint c (or valuation θ). The possible valuations for the variables $\{x, y, z\}$ are $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7, \theta_8\}$. Figure 2 gives the error values of all valuations in the complete search tree using the trivial error function. The error values of a valuation θ are computed for each constraint $(e(c_1^1\theta), e(c_2^1\theta), e(c_1^2\theta), e(c_2^2\theta), e(c_1^3\theta), e(c_2^3\theta))$. Since, for example, θ_1 satisfies c_1^3 but violates c_1^1, $e(c_1^3\theta_1) = 0$ and $e(c_1^1\theta_1) = 1$ respectively. We can obtain the error values of other valuations similarly. In order to compare values, a number of *comparators* are defined: *locally-better* (*l-b*), *weighted-sum-better* (*w-s-b*), *worst-case-better* (*w-c-b*), and *least-squares-better* (*l-s-b*). We can use these comparators to define *solutions* of CHs [BFBW92].



θ	Error values for $e(c_j^i\theta)$	θ	Error values for $e(c_j^i\theta)$
θ_1	$(1,1,1,1,0,1)$	θ_5	$(0,0,1,1,0,1)$
θ_2	$(1,1,1,1,1,1)$	θ_6	$(0,0,1,1,1,0)$
θ_3	$(1,1,1,0,0,1)$	θ_7	$(1,0,1,0,0,0)$
θ_4	$(1,1,1,1,1,0)$	θ_8	$(1,0,1,1,1,0)$

Figure 2: The possible valuations and their error values.

Example 2 (Modeling the sales representative promotion scenario in Example 1)**.** *Let consider the sales representative promotion scenario in Example 1. We model the problem to its nearest approximation as follows. There are four variables: D to denote if the candidate is a degree holder, Y to denote the candidate's working experience in number of years, Q to denote the candidate's sales figure in thousands of dollars, and P to denote the number of products that the candidate can sell. We get the following constraint hierarchy H.*

Level	*Constraints*
H_0	(c_1) $D = \texttt{degree}$
H_1	(c_2) $Y \geq 5$, (c_3) $Q \geq 1000$
H_2	(c_4) $P \geq 20$

The following error function e measures how well the constraints are satisfied:

$$e(c_1\theta) = \begin{cases} 1 & \textit{if } D\theta = \texttt{degree} \\ 0 & \textit{Otherwise} \end{cases} \qquad e(c_2\theta) = \begin{cases} \dfrac{5 - Y\theta}{5} & \textit{if } Y\theta < 5 \\ 0 & \textit{if } Y\theta \geq 5 \end{cases}$$

$$e(c_3\theta) = \begin{cases} \dfrac{1000 - Q\theta}{1000} & \textit{if } Q\theta < 1000 \\ 0 & \textit{if } Q\theta \geq 1000 \end{cases} \qquad e(c_4\theta) = \begin{cases} \dfrac{20 - P\theta}{20} & \textit{if } P\theta < 20 \\ 0 & \textit{if } P\theta \geq 20 \end{cases}$$

Consider two competing candidates A and B with their corresponding qualifications encoded in the valuations θ_A and θ_B respectively:

$$\theta_A = \{D \mapsto \texttt{degree}, Y \mapsto 2.9, Q \mapsto 700, P \mapsto 30\}$$
$$\theta_B = \{D \mapsto \texttt{degree}, Y \mapsto 3, Q \mapsto 690, P \mapsto 2\}$$

Applying the valuations to the constraints in H gives the following error sequences.

θ	$e(c_1\theta)$	$e(c_2\theta)$	$e(c_3\theta)$	$e(c_4\theta)$
θ_A	0	0.42	0.3	0
θ_B	0	0.4	0.31	0.9

The l-b comparator[1] *would find the candidates to be incomparable. Assuming all constraint weights to be 1.0, the global comparators would conclude B to be the better candidate.*

3 A Reformulation of Constraint Hierarchies

To facilitate subsequent illustration of the CH local consistency concept, we reformulate the CH framework [BFBW92] (in particular in the definition of comparators and solution set) using *error indicators*. Notice that the reformulation is equivalent to that given in [BFBW92] in the sense that the quality and the order of the solution is the same in the two approaches.

We denote an error value by ξ, possibly with subscripts. Given a constraint hierarchy $H = \{H_0, \ldots, H_n\}$ where n is the number of non-required levels, and for all $i \in \{0, \ldots, n\}$, $H_i = \{c^i_1, \ldots, c^i_{k_i}\}$ with k_i being the number of constraints in level i.

Definition 1 (Error Indicator). *An error indicator for H is a tuple of error values such that $\vec{\xi} = \langle\langle\xi^0_1, \ldots, \xi^0_{k_0}\rangle, \ldots, \langle\xi^n_1, \ldots, \xi^n_{k_n}\rangle\rangle$ and $\forall a \in \{0, \ldots, n\}$, $\forall b \in \{1, \ldots, k_a\}$, ξ^a_b is an error value.*

According to the definition, error values in an error indicator provide an estimate (perhaps provided by users in the specification of the problem) of how much each labeled constraint in the constraint hierarchy is satisfied. In case the error values are computed from a specific valuation, we have more specifically *error indicator for a valuation*.

Definition 2 (Error Indicator for a Valuation). *Given a valuation θ and a set of variables V, an error indicator $\vec{\xi_\theta}$ for θ and V is a tuple of error values such that $\vec{\xi_\theta} = \langle\langle{\xi_\theta}^0_1, \ldots, {\xi_\theta}^0_{k_0}\rangle, \ldots, \langle{\xi_\theta}^n_1, \ldots, {\xi_\theta}^n_{k_n}\rangle\rangle$ and $\forall a \in \{0, \ldots, n\}, \forall b \in \{1, \ldots, k_a\}, {\xi_\theta}^a_b = e(c^a_b\theta)$ if $vars(c^a_b) \subset V$ and ${\xi_\theta}^a_b = 0$ if $vars(c^a_b) \not\subset V$.*

Error indicators of a valuation provide a measure of the "badness" of valuations with respect to H. The two notions are similar, differing only in whether

[1] we would define precisely the *l-b* and global comparators in the next section.

the error values were specifically computed from a valuation. Thus, an error indicator of a valuation is also an error indicator, but not *vice versa*.

To explain the meaning of the error indicator of a valuation, we use the example in Figure 1 with the trivial error function. If $\theta = \{z \mapsto 2\}$, then $\vec{\xi_\theta} = \langle\langle\rangle, \langle 0,0\rangle, \langle 0,0\rangle, \langle 1,0\rangle\rangle$. If $\theta = \{x \mapsto 1, y \mapsto 2\}$, then $\vec{\xi_\theta} = \langle\langle\rangle, \langle 1,1\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle$. If $\theta = \{x \mapsto 2, y \mapsto 2, z \mapsto 1\}$, then $\vec{\xi_\theta} = \langle\langle\rangle, \langle 1,0\rangle, \langle 1,0\rangle, \langle 0,1\rangle\rangle$.

The comparator predicate *better* in the original CH formulation is redefined using a *partial order*, denoted by $\prec$. Let $I = \{\vec{\xi_1}, \ldots, \vec{\xi_N}\}$ be a poset (partially ordered set), each element $\vec{\xi_j}$ of which is an error indicator. We define $\prec$ to be irreflexive and transitive over I. Hence, it preserves the meaning of *better*. Intuitively, $\vec{\xi'} \prec \vec{\xi''}$ means $\vec{\xi''}$ is "better" than $\vec{\xi'}$ in I. In general, $\prec$ will not provide a total ordering. For convenience, we define $\preceq$ such that $\forall \vec{\xi'}, \vec{\xi''} \in I, \vec{\xi'} \preceq \vec{\xi''} \rightarrow (\vec{\xi'} \prec \vec{\xi''}) \vee (\vec{\xi'} = \vec{\xi''})$.

We can redefine *l-b* in the original formulation as a partial order $\prec_{l-b}$ as follows. Given any two valuations θ and σ, and the corresponding error indicators $\vec{\xi_\theta}$ and $\vec{\xi_\sigma}$, $\prec_{l-b}$ is defined as:

$$\begin{aligned}\vec{\xi_\theta} \prec_{l-b} \vec{\xi_\sigma} \equiv\ & \exists l > 0 \text{ such that } \forall i \in \{0, \ldots, l-1\}, \\ & \forall j \in \{1, \ldots, k_i\}, {\xi_\theta}_j^i = {\xi_\sigma}_j^i \\ & \wedge \exists a \in \{1, \ldots, k_l\}, {\xi_\sigma}_a^l < {\xi_\theta}_a^l \\ & \wedge \forall b \in \{1, \ldots, k_l\}, {\xi_\sigma}_b^l \leq {\xi_\theta}_b^l.\end{aligned}$$

The intuitive meaning of $\vec{\xi_\theta} \prec_{l-b} \vec{\xi_\sigma}$ is that valuation σ is *locally-better* than valuation θ.

Similarly, we can define *g-b* ($\prec_{g-b}$), and its instances *w-s-b* ($\prec_{w-s-b}$), *w-c-b* ($\prec_{w-c-b}$), and *l-s-b* ($\prec_{l-s-b}$) respectively. Given any two valuations θ and σ, and the corresponding error indicators $\vec{\xi_\theta}$ and $\vec{\xi_\sigma}$:

$$\begin{aligned}\vec{\xi_\theta} \prec_{g-b} \vec{\xi_\sigma} \equiv\ & \exists l > 0 \text{ such that } \forall i \in \{0, \ldots, l-1\}, \\ & g(\langle {\xi_\theta}_1^i, \ldots, {\xi_\theta}_{k_i}^i \rangle) = g(\langle {\xi_\sigma}_1^i, \ldots, {\xi_\sigma}_{k_i}^i \rangle) \\ & \wedge g(\langle {\xi_\sigma}_1^l, \ldots, {\xi_\sigma}_{k_l}^l \rangle) < g(\langle {\xi_\theta}_1^l, \ldots, {\xi_\theta}_{k_l}^l \rangle),\end{aligned}$$

where g is a *combining function* for error values:

$$\begin{aligned}&\vec{\xi_\theta} \prec_{w-s-b} \vec{\xi_\sigma} \equiv \vec{\xi_\theta} \prec_{g-b} \vec{\xi_\sigma}, \text{ where } g(\langle \xi_1^i, \ldots, \xi_{k_i}^i \rangle) \equiv \textstyle\sum_{j \in \{1,\ldots,k_i\}} \xi_j^i, \\ &\vec{\xi_\theta} \prec_{w-c-b} \vec{\xi_\sigma} \equiv \vec{\xi_\theta} \prec_{g-b} \vec{\xi_\sigma}, \text{ where } g(\langle \xi_1^i, \ldots, \xi_{k_i}^i \rangle) \equiv \max\{\xi_j^i \mid j \in \{1, \ldots, k_i\}\}, \\ &\vec{\xi_\theta} \prec_{l-s-b} \vec{\xi_\sigma} \equiv \vec{\xi_\theta} \prec_{g-b} \vec{\xi_\sigma}, \text{ where } g(\langle \xi_1^i, \ldots, \xi_{k_i}^i \rangle) \equiv \textstyle\sum_{j \in \{1,\ldots,k_i\}} {\xi_j^i}^2.\end{aligned}$$

Notice that, by definition, all local/global comparators ignore constraints in hierarchy levels greater than or equal to l.

We are now ready to define the solution set S of a CH with variables V.

Definition 3 (solution set for constraint hierarchies)**.** *The solution set S of a CH with variables V is defined as follows:*

$$S = \{\theta \in S_0 \mid \forall \sigma \in S_0, \vec{\xi_\theta} \not\prec \vec{\xi_\sigma}\},$$

where $S_0 = \{\theta \mid vars(\theta) = V, {\xi_\theta}_i^0 = 0 \text{ for all } i \in \{1, \ldots, k_0\}\}$.

The following lemmas give the monotonicity of the introduced comparators, which are collectively denoted by $\prec_{better}$ and $\preceq_{better}$ in the rest of the paper.

Lemma 1. *Given any two error indicators $\vec{\xi'}$ and $\vec{\xi''}$ for a constraint hierarchy. If we have ${\xi''}^a_b \leq {\xi'}^a_b$ for all a, b, then $\vec{\xi'} \preceq_{better} \vec{\xi''}$.*

Proof. If we have ${\xi''}^a_b \leq {\xi'}^a_b$ for all a, b, it means that we can find an index $l > 0$ such that $\forall i \in \{0, \ldots, l-1\}$, and $\forall j \in \{1, \ldots, k_i\}$, we have ${\xi''}^i_j = {\xi'}^i_j$, and $\exists e \in \{1, \ldots, k_l\}, {\xi''}^l_e < {\xi'}^l_e$, and $\forall f \in \{1, \ldots, k_l\}, {\xi''}^l_f \leq {\xi'}^l_f$.

But this is just the definition of $\vec{\xi'} \prec_{l-b} \vec{\xi''}$. Moreover, since $\vec{\xi'} \preceq_{l-b} \vec{\xi''} \implies \vec{\xi'} \preceq_{g-b} \vec{\xi''}$ [BFBW92], we have ${\xi''}^a_b \leq {\xi'}^a_b \implies \vec{\xi'} \preceq_{g-b} \vec{\xi''}$ by transitivity. □

Notice that Lemma 1 allows us to compare valuations for both local and global comparators (because the $\preceq_{better}$ order implies all the orders induced from any specific comparator) and for arbitrary error functions.

Lemma 2. *Given two valuations θ_1 and θ_2 for a constraint hierarchy. If $\theta_1 \subseteq \theta_2$, then $\vec{\xi}_{\theta_2} \preceq_{better} \vec{\xi}_{\theta_1}$.*

Proof. Note that, by Definition 2, the error value of a constraint is 0 if not all variables in the constraint are completely instantiated. Therefore, result holds directly from Lemma 1. □

We also introduce the notion of a *hierarchy problem* which is a CH augmented with a set of *soft membership constraints*.

Definition 4 (Soft Membership Constraints)**.** *Let H be a constraint hierarchy with variables V. A soft membership constraint for variable $x \in V$ and its associated domain $D(x)$ is in the form $x \in D(x)$. Each soft membership constraint $x \in D(x)$ is a function that assigns an error indicator $\vec{\xi}_{x=d}$ to each domain $d \in D(x)$.*

A soft membership constraint is similar to a soft constraint in the sense of SCSP [BMR97], in which each tuple of a soft constraint is associated with a semiring value. The error indicator $\vec{\xi}_{x=d}$ indicates the quality of assigning d to x among all values in $D(x)$.

Definition 5 (Hierarchy Problem)**.** *A hierarchy problem $P = \langle H, I_H \rangle$ is a pair, where H is a CH with variables V and I_H is a set of soft membership constraints for all variables in V. Each $\vec{\xi}_{x=d}$ in I_H is used for keeping an estimate of the errors of valuations involving $\{x \mapsto d\}$.*

Figure 3 shows a possible instance of the set I_H for the Hierarchy problem obtained from the example of Figure 1. For each assignment $\{x \mapsto d\}$ we have to compute the associated error indicator $\vec{\xi}_{x=d}$. So for instance for the assignment $\{x \mapsto 1\}$ we compute $\vec{\xi}_{x=1} = (0, \mathbf{1}, 0, 0, 0, 0)$. The error estimate for all the constraints except c^1_2 is set to 0, because these constraints involve other variables. The constraint c^1_2 instead involves only variable x and the error value

valuation	Error values	valuation	Error values
$\vec{\xi}_{x=1}$	$(0, \mathbf{1}, 0, 0, 0, 0)$	$\vec{\xi}_{x=2}$	$(0, \mathbf{0}, 0, 0, 0, 0)$
$\vec{\xi}_{y=1}$	$(0, 0, \mathbf{1}, 0, 0, 0)$	$\vec{\xi}_{y=2}$	$(0, 0, \mathbf{1}, 0, 0, 0)$
$\vec{\xi}_{z=1}$	$(0, 0, 0, 0, \mathbf{0}, 0)$	$\vec{\xi}_{z=2}$	$(0, 0, 0, 0, \mathbf{1}, 0)$

Figure 3: A possible instance of the error indicators I_H related to the problem in Figure 1.

can be computed. Since the assignment $\{x \mapsto 1\}$ does not satisfy constraint $c_2^1 : x = 2$, the error is set to **1**. Similarly we can compute all the other error indicators.

Definition 6 (Solution of a Hierarchy Problem)**.** *A valuation* θ *is a* solution *of* $P = \langle H, I_H \rangle$ *if (1)* θ *is a solution of* H *and (2)* $\vec{\xi}_\theta \preceq_{better} \vec{\xi}_{x=d}$ *for all* $(x \mapsto d) \in \theta$*.*

In other words, solutions of $P = \langle H, I_H \rangle$ are solutions of H which have a "worse" error than the estimates provided in I_H. By definition, the set of solutions of H always contains those of $\langle H, I_H \rangle$. Equality holds when the error estimates provided in I_H fails to "filter" out any solutions of H.

theorem 1. *Consider a CH* H *and a hierarchy problem* $P = \langle H, I_H \rangle$*, and denote the* solution sets *of* H *and* P *by* S_H *and* S_P *respectively.*

- $S_P \subseteq S_H$*, and*
- $S_P = S_H$ *if* $\vec{\xi}_\theta \preceq_{better} \vec{\xi}_{x=d}$ *for all* $(x \mapsto d) \in \theta$ *and* $\theta \in S_H$*.*

Proof. Trivially holds from Definition 6. □

In particular, a hierarchy problem $\langle H, I_H \rangle$ must share the same solution as H if all $\vec{\xi}_{x=d}$'s in I_H contain only the error value 0 (*i.e.* no error information).

Corollary 1. *Consider a CH* H *and a hierarchy problem* $P = \langle H, \emptyset_H \rangle$*, in which all error indicators in* $\emptyset_H$ *contain only zero error values. Denote the solution sets of* H *and* P *by* S_H *and* S_P *respectively. We have* $S_P = S_H$*.*

Proof. Trivially holds from Theorem 1 and Definition 6. □

In other words, constraint hierarchies are just special cases of hierarchy problems, and techniques developed for solving hierarchy problems are applicable for solving constraint hierarchies as well. This fact is useful in ensuring the correctness of our local consistency algorithm and the completeness of our branch-and-bound solver for solving constraint hierarchies.

4 Local Consistency in CHs

The classical notion of *local consistency* [Mac77] characterizes when a constraint problem contains non-fruitful values. The main purpose of detecting local inconsistency is thus to remove the inconsistent values from the variable domains and constraints. Hence, the problem is "simpler" to solve when the problem is smaller. However, we adopt a more general notion of local consistency used for SCSP: *"Applying a local consistency algorithm to a constraint problem means explicitating some implicit constraints, thus possibly discovering inconsistency at a local level"* [BMR97].

4.1 Local Consistency in Classical CSPs

In this paper we focus just on *node* and *arc* consistency algorithms which are common techniques to detect local inconsistency.

Let us illustrate the concepts using an example. Given a CSP P where $V = \{x, y\}$, $D(x) = \{1, 2, 3, 4, 5\}$, $D(y) = \{1, 2, 3, 4, 5\}$, and $C = 3 \le x \land x < y$. P is node inconsistent, since $\{x \mapsto 1\}$ and $\{x \mapsto 2\}$ are not solutions of the unary constraint $3 \le x$. It is possible to transform P into an equivalent CSP P' which is node consistent if the inconsistent domain values in $D(x)$ are removed. Hence, the equivalent CSP P' is $V = \{x, y\}$, $D(x) = \{3, 4, 5\}$, $D(y) = \{1, 2, 3, 4, 5\}$, and $C = 3 \le x \land x < y$.

Although P' is node consistent, it is arc inconsistent since $\{x \mapsto 5\}$ cannot find support from $D(y)$ to satisfy the binary constraint $x < y$. Also, $\{y \mapsto 1\}$, $\{y \mapsto 2\}$, and $\{y \mapsto 3\}$ cannot find support from $D(x)$ to satisfy $x < y$. Similarly, we can transform P' into an equivalent CSP P'' which is arc consistent if the inconsistency domain values in $D(x)$ and $D(y)$ are removed. Hence, the equivalent CSP P'' is $V = \{x, y\}$, $D(x) = \{3, 4\}$, $D(y) = \{4, 5\}$, and $C = 3 \le x \land x < y$.

P' and P'' are equivalent to P, since the solution sets of P' and P'' are the same as that of P. However, the domain size of P' and P'' is smaller. Hence P' and P'' have a smaller search space and are easier to solve. We can conclude that applying consistency algorithm to a classical CSP aims to reduce the variable domains of the CSP so that the CSP becomes node and arc consistent and equivalent to the original CSP.

4.2 Local Consistency in SCSPs

SCSPs [BMR97, Bis04] extends classical CSPs by allowing non-crisp features. Hence, classical CSPs are just an instance of SCSPs over the c-semiring $S_{CSP} = \langle \{true, false\}, \lor, \land, false, true \rangle$. In SCSPs, a general notion of local consistency is proposed but we just focus on semiring-based arc-consistency [BR98] in this paper.

Given the same CSP P (considered as an SCSP) where $V = \{x, y\}$, $D(x) = \{1, 2, 3, 4, 5\}$, $D(y) = \{1, 2, 3, 4, 5\}$, $C = 3 \le x \land x < y$. Note also the implicit constraints $x \in \{1, 2, 3, 4, 5\}$ and $y \in \{1, 2, 3, 4, 5\}$. Therefore, we have two unary constraints, namely ($x \in \{1,2,3,4,5\} \wedge 3 \leq x$) and $y \in \{1,2,3,4,5\}$, and one binary constraint $x < y$. Extensionally, we consider a constraint as a set of tuples with the associated semiring values. Initially, the extensional representation[2] of the two unary constraints is as follows:

- Constraint on x: $\{1(false), 2(false), 3(true), 4(true), 5(true)\}$.
- Constraint on y: $\{1(true), 2(true), 3(true), 4(true), 5(true)\}$.

The semiring values in the unary constraints take no notice of the constraint information in the binary constraint, and thus P is semiring-based arc-inconsistent.

However, a semiring-based arc consistency algorithm can transform the semiring values in the unary constraints as follows:

- Constraint on x: $\{1(false), 2(false), 3(true), 4(true), 5(false)\}$.
- Constraint on y: $\{1(false), 2(false), 3(false), 4(true), 5(true)\}$.

to make the SCSP arc-consistent. The resultant SCSP expresses the same information as P'' in the last section.

Although the domain size of the resultant (S)CSP remains unchanged after applying the semiring-based arc consistency algorithm, we still gain useful information since we are "explicitating some implicit constraints" as semiring values in the unary constraints. Based on this inconsistency information, a search algorithm can know not to try the domain values that are marked $false$. Hence, semiring-based arc consistency is a generalization of classical local consistency.

4.3 Local Consistency in CHs

We adapt the general notion of local consistency for CH, and define *constraint hierarchy k-consistency* (CH-k-C).

Before defining CH-k-C, we need two operations, $\mathcal{MAX}$ and $\mathcal{MIN}$, on error indicators. Given a CH H with n non-required levels and any two error indicators, $\vec{\xi_\theta}, \vec{\xi_\sigma} \in I$, for H. $\mathcal{MAX}(\vec{\xi_\theta}, \vec{\xi_\sigma})$ is defined as

$$\langle\langle max({\xi_\theta}_1^0, {\xi_\sigma}_1^0), \ldots, max({\xi_\theta}_{k_0}^0, {\xi_\sigma}_{k_0}^0)\rangle, \ldots, \langle max({\xi_\theta}_1^n, {\xi_\sigma}_1^n), \ldots, max({\xi_\theta}_{k_n}^n, {\xi_\sigma}_{k_n}^n)\rangle\rangle$$

and $\mathcal{MIN}(\vec{\xi_\theta}, \vec{\xi_\sigma})$ is

$$\langle\langle min({\xi_\theta}_1^0, {\xi_\sigma}_1^0), \ldots, min({\xi_\theta}_{k_0}^0, {\xi_\sigma}_{k_0}^0)\rangle, \ldots, \langle min({\xi_\theta}_1^n, {\xi_\sigma}_1^n), \ldots, min({\xi_\theta}_{k_n}^n, {\xi_\sigma}_{k_n}^n)\rangle\rangle$$

where k_i is the number of constraints in level i of H.

Given two error indicators, $\mathcal{MIN}$ (or $\mathcal{MAX}$) combines the two indicators by taking the best (or the worst). Obviously $\mathcal{MAX}$ and $\mathcal{MIN}$ are commutative and associative. Thus, it makes sense to write $\mathcal{MAX}\{\vec{\xi_1}, \ldots, \vec{\xi_K}\}$ and $\mathcal{MIN}\{\vec{\xi_1}, \ldots, \vec{\xi_K}\}$) for any $K > 2$.

[2]We adopt the convention of putting the associated semiring value of a tuple in parentheses.

Given a CH H with variables V. If $x \in V$ and $d \in D(x)$, we define

$$approx_k(x \mapsto d) = \mathcal{MAX}\{\mathcal{MIN}\{\vec{\xi}_\theta \mid vars(\theta) = \{x\} \cup U, (x \mapsto d) \in \theta\} \mid U \subset V, |U| = k-1\}$$

for any $1 \leq k \leq |V|$. Informally, to compute $approx_k(x \mapsto d)$, given a constraint involving variable x, we can select among all the assignment containing $x \mapsto d$ the best one (that is the assignment with minimum error). This is the approximation coming from a specified constraint. If more than one constraint involve variable x, we have to consider the approximation for all of them and then compute the worst error (the maximum). That is the "MAX" in the $approx_k$ definition simply collects error values for different constraints into one error indicator. We call it *k-approximation*, which provides an estimate of the "badness" of valuations involving the assignment $x \mapsto d$ for all m_1-ary constraints involving x with $m_1 \leq k$ and all m_2-ary constraints not involving x with $m_2 < k$. Since the error indicators of all valuations involving $x \mapsto d$ might not be comparable, we can only give an approximation, and $approx_{|V|}(x \mapsto d)$ gives an error estimate involving all constraints in the problem. However, calculating $approx_{|V|}(x \mapsto d)$ is computationally expensive, and $approx_k(x \mapsto d)$ for some small $k < |V|$ gives a more practical approximation.

Referring to the same example in Section 2,

$$\begin{aligned} approx_2(y \mapsto 2) &= \mathcal{MAX}\{\mathcal{MIN}\{\vec{\xi}_{\{x\mapsto 1, y\mapsto 2\}}, \vec{\xi}_{\{x\mapsto 2, y\mapsto 2\}}\}, \\ &\qquad \mathcal{MIN}\{\vec{\xi}_{\{y\mapsto 2, z\mapsto 1\}}, \vec{\xi}_{\{y\mapsto 2, z\mapsto 2\}}\}\} \\ &= \mathcal{MAX}\{\mathcal{MIN}\{\langle\langle\rangle, \langle 1,1\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle, \langle\langle\rangle, \langle 1,0\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle\}, \\ &\qquad \mathcal{MIN}\{\langle\langle\rangle, \langle 0,0\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle, \langle\langle\rangle, \langle 0,0\rangle, \langle 1,1\rangle, \langle 1,0\rangle\rangle\}\} \\ &= \mathcal{MAX}\{\langle\langle\rangle, \langle 1,0\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle, \langle\langle\rangle, \langle 0,0\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle\} \\ &= \langle\langle\rangle, \langle 1,0\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle \end{aligned}$$

The following theorem states that $approx_k(x \mapsto d)$ is monotonically decreasing in k with respect to $\preceq_{better}$.

theorem 2. *If H is a CH with variables V, $x \in V$ and $d \in D(x)$, then $approx_{k_2}(x \mapsto d) \preceq_{better} approx_{k_1}(x \mapsto d)$, $\forall 1 \leq k_1 \leq k_2 \leq |V|$.*

Proof. Define

$$L(k, U, x, d) = \{\vec{\xi}_\theta \mid vars(\theta) = \{x\} \cup U, (x \mapsto d) \in \theta\}$$

Therefore,

$$approx_k(x \mapsto d) = \mathcal{MAX}\{\mathcal{MIN}(L(k, U, x, d)) \mid U \subset V, |U| = k-1\}$$

Given H with variables V. Consider the case when $k_1 > 1$. For all $U_1 \subset V$ such that $|U_1| = k_1 - 1$, there exists $U_2 \subset V$ such that $|U_2| = k_2 - 1$ and $U_1 \subset U_2$. In addition, for all $\vec{\xi}_{\theta_2} \in L(k_2, U_2, x, d)$, we can find $\vec{\xi}_{\theta_1} \in L(k_1, U_1, x, d)$ such that $\theta_1 \subset \theta_2$. By Lemma 2, $\vec{\xi}_{\theta_2} \preceq_{better} \vec{\xi}_{\theta_1}$. Therefore,

$$\mathcal{MIN}(L(k_2, U_2, x, d)) \preceq_{better} \mathcal{MIN}(L(k_1, U_1, x, d))$$

The situation is similar and simpler for the case of $k_1 = 1$.

In other words, every $\mathcal{MIN}(L(k_1, U_1, x, d))$ for each U_1 in $approx_{k_1}(x \mapsto d)$ has a higher-error counterpart in $approx_{k_2}(x \mapsto d)$. Thus, the theorem holds, since $\mathcal{MAX}$ is an extensive operator (returning the error indicator with the highest error values). □

By using Lemma 1 we can show that k-approximations of $x \mapsto d$ provide upper bounds with respect to $\preceq_{better}$ (the best possible scenario) for the error indicators of complete valuations involving $x \mapsto d$ for any comparators.

theorem 3. *If H is a CH with variables V, $x \in V$ and $d \in D(x)$, then $\vec{\xi_\theta} \preceq_{better} approx_{|V|}(x \mapsto d) \preceq_{better} approx_k(x \mapsto d)$ for all $1 \leq k \leq |V|$ and all θ such that $vars(\theta) = V$ and $(x \mapsto d) \in \theta$, where $\preceq_{better}$ represents any locally/globally better comparator.*

Proof. Theorem 2 implies that $approx_{|V|}(x \mapsto d) \preceq_{better} approx_k(x \mapsto d)$. We have only to prove that $\vec{\xi_\theta} \preceq_{better} approx_{|V|}(x \mapsto d)$.

$$approx_{|V|}(x \mapsto d) =$$
$$\mathcal{MAX}\{\mathcal{MIN}\{\vec{\xi_\theta} \mid vars(\theta) = \{x\} \cup U, (x \mapsto d) \in \theta\} \mid U \subset V, |U| = |V| - 1\} =$$
$$\mathcal{MAX}\{\mathcal{MIN}\{\vec{\xi_\theta} \mid vars(\theta) = \{x\} \cup U, (x \mapsto d) \in \theta\} \mid U = V - \{x\}\} =$$
$$\mathcal{MAX}\{\mathcal{MIN}\{\vec{\xi_\theta} \mid vars(\theta) = V, (x \mapsto d) \in \theta\}\} =$$
$$\mathcal{MIN}\{\vec{\xi_\theta} \mid vars(\theta) = V, (x \mapsto d) \in \theta\}.$$

Now, since $approx_{|V|}(x \mapsto d)$ is obtained from the $\mathcal{MIN}$ (to get the least error values) of error indicators of all possible θ containing $x \mapsto d$, we easily have $\vec{\xi_\theta} \preceq_{better} approx_{|V|}(x \mapsto d)$. □

Referring to the same example in Section 2 again, θ_3, θ_4, θ_7, and θ_8 in Fig. 2 contain $(y \mapsto 2)$. The error indicators $\vec{\xi}_{\theta_3}$, $\vec{\xi}_{\theta_4}$, $\vec{\xi}_{\theta_7}$, and $\vec{\xi}_{\theta_8}$ are $\langle\langle\rangle, \langle 1,1\rangle, \langle 1,0\rangle, \langle 0,1\rangle\rangle$, $\langle\langle\rangle, \langle 1,1\rangle, \langle 1,1\rangle, \langle 1,0\rangle\rangle$, $\langle\langle\rangle, \langle 1,0\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle$, and $\langle\langle\rangle, \langle 1,0\rangle, \langle 1,1\rangle, \langle 1,0\rangle\rangle$ respectively. Thus, $approx_{|V|}(y \mapsto 2) =$
$\mathcal{MIN}\{\vec{\xi}_{\theta_3}, \vec{\xi}_{\theta_4}, \vec{\xi}_{\theta_7}, \vec{\xi}_{\theta_8}\} = \langle\langle\rangle, \langle 1,0\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle$, which is equal to $approx_2(y \mapsto 2)$ as computed and given (before Theorem 2). We have $\vec{\xi}_{\theta_i} \preceq_{better} approx_{|V|}(y \mapsto 2) \preceq_{better} approx_2(y \mapsto 2)$, where $i \in \{3, 4, 7, 8\}$.

Theorem 3 suggests that k-approximations can be used as the basis of the notion of local consistency in CH.

A hierarchy problem $P = \langle H, I_H\rangle$ is *constraint hierarchy k-consistent (CH-k-C)* if the error indicators in I_H explicitly indicate the implicit inconsistency information in all m-ary constraints in H where $m \leq k$. Formally, we define CH-k-C as follows.

Definition 7 (CH-k-Consistency (CH-k-C))**.** *Given a hierarchy problem $P = \langle H, I_H\rangle$ with variables V. Given $1 \leq k \leq |V|$, P is CH-k-C if, for all $\vec{\xi}_{x=d}$ in I_H, $\vec{\xi}_{x=d} \preceq_{better} approx_k(x \mapsto d)$.*

The CH-k-C condition of $P = \langle H, I_H \rangle$ imposes that the estimated error information of H placed in the error indicators in I_H is *at least* as accurate as that provided by k-approximations. In addition, explicating the error $P = \langle H, I_H \rangle$ using k-approximations makes P CH-k-C without changing the solution space of P.

theorem 4. *Given a hierarchy problem $P = \langle H, I_H \rangle$ with variables V. If each $\vec{\xi'}_{x=d} \in I'_H$ is defined as follows:*

$$\vec{\xi'}_{x=d} = \begin{cases} \vec{\xi}_{x=d} & \text{if } \vec{\xi}_{x=d} \preceq_{better} approx_k(x \mapsto d) \\ approx_k(x \mapsto d) & \text{if } approx_k(x \mapsto d) \preceq_{better} \vec{\xi}_{x=d} \end{cases}$$

where $\vec{\xi}_{x=d}$ is in I_H, then (1) the hierarchy problem $P' = \langle H, I'_H \rangle$ is CH-k-C and (2) $S_P = S_{P'}$, where S_P and $S_{P'}$ are the solution sets of P and P' respectively.

Proof. (1) holds directly from Definition 7. Let's consider now (2).

We note that $\vec{\xi'}_{x=d} \preceq_{better} \vec{\xi}_{x=d}$ for all x and d. By Definition 6, $S_{P'} \subseteq S_P$.

Suppose $\vec{\xi}_\theta \in S_P$. We have $\vec{\xi}_\theta \preceq_{better} \vec{\xi}_{x=d}$ for all x and d. By Theorem 3, $\vec{\xi}_\theta \preceq_{better} approx_k(x \mapsto d)$. Therefore, $\vec{\xi}_\theta \in S_{P'}$, and $S_P \subseteq S_{P'}$. □

Two corollaries follow directly from Theorems 1 and 4.

Corollary 2. *Given a hierarchy problem $P = \langle H, I_H \rangle$ with variables V, and $P' = \langle H, I'_H \rangle$ defined so that each $\vec{\xi'}_{x=d}$ in I'_H is:*

$$\vec{\xi'}_{x=d} = \begin{cases} \vec{\xi}_{x=d} & \text{if } \vec{\xi}_{x=d} \preceq_{better} approx_k(x \mapsto d) \\ approx_k(x \mapsto d) & \text{if } approx_k(x \mapsto d) \preceq_{better} \vec{\xi}_{x=d} \end{cases}$$

where $\vec{\xi}_{x=d}$ is in I_H. Denote the solution sets of H, P, and P' by S_H, S_P, and $S_{P'}$ respectively.

$$S_H = S_P \Leftrightarrow S_H = S_{P'}$$

Proof. Holds directly from Theorem 4 and by Definition 6. □

Corollary 3. *Given a hierarchy problem $P = \langle H, \emptyset_H \rangle$ with variables V, and all error indicators in $\emptyset_H$ contain only zero error values. Define $P' = \langle H, I'_H \rangle$ so that each $\vec{\xi'}_{x=d}$ in I'_H is:*

$$\vec{\xi'}_{x=d} = approx_k(x \mapsto d)$$

Denote the solution sets of H, P, and P' by S_H, S_P, and $S_{P'}$ respectively.

$$S_H = S_P = S_{P'}$$

Proof. Holds directly from Theorem 4 and by Definition 6. □

Corollary 3 ensures that the CH-k-consistency techniques are also applicable to constraint hierarchies.

To perform constraint checking only on unary and binary constraints is the most commonly used technique for detecting local inconsistency in classical CSPs. Therefore, we discuss CH-2-C and provide a CH-2-C enforcement algorithm in the next section.

5 A CH-2-C Enforcement Algorithm

Arc-consistency algorithm is a common and practical technique to detect local inconsistency in classical CSPs [BFR95, GS96]. We design and implement an algorithm to enforce CH-2-C. The purpose of the CH-2-C algorithm is to explicate and place in I_H the implicit error information in a CH that is otherwise not visible. Such an algorithm is given in Figure 4. The subroutines **ch1c_pri** and **ch2c_pri**, in Figures 5 and 6 respectively, are responsible for handling unary and binary constraints respectively. The CH-2-C algorithm ensures that all error indicator stores $\vec{\xi}_{x=d}$ are updated to reach $approx_2(x \mapsto d)$.

Algorithm 1: The CH-2-C algorithm.

```
ch2c(H, V, D, I_H)
begin
    for l ← 1 to n do
        for k ← 1 to |H_l| do
            let c be the k^th constraint in H_l;
            I_H ← ch1c_pri(c, l, k, D, I_H);
            I_H ← ch2c_pri(c, l, k, D, I_H);
    return I_H;
end
```

Figure 4: The CH-2-C algorithm.

```
ch1c_pri(c, l, k, D, I_H)
begin
    if |vars(c)| = 1 then
        let {x} = vars(c);
        for each d ∈ D(x) do
            let θ = {x ↦ d};
            let ξ⃗ = ξ⃗_{x=d} in I_H;
            if ξ^l_k < e(cθ) then ξ^l_k ← e(cθ);
    return I_H;
end
```

Figure 5: A subroutine to check unary constraints.

Consider a general CH of n_c labeled constraints with n_v number of variables. In addition, the size of the largest variable domain is of n_d. The time complexity

```
ch2c_pri(c, l, k, D, I_H)
begin
    if |vars(c)| = 2 then
        let {x, y} = vars(c);
        # Update each ξ⃗_{x=d_x} in I_H
        I_H ← update(x, y, c, l, k, D, I_H);
        # Update each ξ⃗_{y=d_y} in I_H
        I_H ← update(y, x, c, l, k, D, I_H);
    return I_H;
end
```

Figure 6: A subroutine to check binary constraints.

of the subroutine **ch1c_pri** is simply of $O(n_d)$, since the only repeating operations, lines 4 to 6 in Figure 5, are placed inside a single loop. These operations are repeated until each element in a variable domain is tested. However, the time complexity of the subroutine **update** (Figure 7) is of $O({n_d}^2)$. Therefore, in the worst case, the time complexity of the subroutine **ch2c_pri** is of $O({n_d}^2)$ as shown in Figure 6. Lines 3 to 5 in the pseudocode of the CH-2-C algorithm are the operations for checking constraints as shown in Figure 4. Since these operations should repeat until all the constraints are considered, the time complexity should be of $O(n_c {n_d}^2)$.

Since an error indicator is a tuple which stores error values of the corresponding constraints, the space complexity for each error indicator is of $O(n_c)$. The memory requirement of the CH-2-C algorithm depends on the number of error indicators in I_H. Therefore, we require $n_v n_d$ error indicators. The space complexity of the CH-2-C algorithm is simply of $O(n_v n_d n_c)$ in the worst case.

6 A Branch-and-Bound Finite Domain CH Solver

The simplest way to find the solution set of a CH is to construct the complete search tree for the problem, so that we can calculate and compare the error values of each valuation. However, traversing the complete search tree and comparing all the valuations are tedious and time-consuming. We propose to combine the CH-2-C and the branch-and-bound algorithms so as to prune nonfruitful branches of the search tree.

The input to our solver is a hierarchy problem $P = \langle H, I_H \rangle$, in which I_H contains *no* error information. In other words, the error indicator stores in I_H contain only the error value 0. The backbone of our solver is a standard branch-and-bound algorithm, since CH-solving is an optimization problem. A branch-and-bound algorithm always maintains the set of potential best solutions collected so far. The idea is to invoke the CH-2-C algorithm at each node in

update(x, y, c, l, k, D, I_H)
begin
  let ξ_{min} be an error value;
  for *each* $d_x \in D(x)$ **do**
    $\xi_{min} \leftarrow \infty$;
    for *each* $d_y \in D(y)$ **do**
      let $\theta = \{x \mapsto d_x, y \mapsto d_y\}$;
      if $e(c\theta) < \xi_{min}$ **then** $\xi_{min} \leftarrow e(c\theta)$;
    let $\vec{\xi} = \vec{\xi}_{x=d_x}$ in I_H;
    if $\xi_k^l < \xi_{min}$ **then** $\xi_k^l \leftarrow \xi_{min}$;
  return I_H;
end

Figure 7: A subroutine to update error indicator stores.

the search tree, hoping that the overhead in the CH-2-C algorithm can be more than compensated by the pruning that can take place. The correctness and completeness of this step is ensured by Corollary 2, so that maintaining CH-2-C will not change the solution space of the hierarchy problem and the associated CH. At each CH-2-C tree node, before search proceeds down a selected branch corresponding to a variable assignment, say $x \mapsto d$, the solver tries to verify if $\vec{\xi}_{x=d}$ in I_H of that tree node is not worse than the error indicator of each potential solution. If that is the case, search proceeds; otherwise, there is no point to explore the selected branch any further, and search is backtracked to try another branch. When a leaf node is reached, we compare the error indicator $\vec{\xi}$ of the valuation associated with the leaf node against the error indicators of all the collected solutions. If the error indicator of any collected solution is worse than $\vec{\xi}$, then the collected solution will be replaced by the current valuation. The details of our finite domain CH solver are shown in Figure 8, which is a simple adaptation of a basic branch-and-bound solver with the CH-2-C algorithm. The numbered lines give the backbone of the algorithm, while the unnumbered lines are new additions to enable CH-2-C enforcement. The algorithm use as parameters the constraints in H and and the stores in I_H, the variables V and the domain D. It also needs the set of assignments S_0 satisfying constraints in H_0, and the corresponding set of error indicators I_{S_0}. The algorithm is also parametric w.r.t. the type of comparator we want to use ($\prec_{better}$).

Although CH-2-C could also deal with crisp constraints, we employ classical algorithms [Mac77] for processing the required constraints in H_0 (lines 1) for two reasons. First, the CH-2-C lacks the "propagation phase" of traditional crisp algorithms (the algorithm update the domains and the modification is used to reconsider the fact that other values may also change). The second reason is

Algorithm 2: A Branch-and-bound CH Solver with Pruning

bb_solv(H, I_H, V, D, S_0, **in out** I_{S_0}, $\prec_{better}$)
begin
  # Any classical arc consistency algorithm
  $D \leftarrow$ **arc_consistent**(H_0, D);
  if *D contains an empty variable domain* **then**
    return S_0;
  else if *D contains all singleton variable domain* **then**
    let θ be the valuation corresponding to D;
    let $\vec{\xi_\theta}$ be the error indicator corresponding to θ;
    $\vec{\xi_\theta} \leftarrow$ **cal_error_values**(H, θ, $\vec{\xi_\theta}$);
    for *each* $\sigma \in S_0$ **do**
      if $\vec{\xi_\sigma} \prec_{better} \vec{\xi_\theta}$ **then**
        $S_0 \leftarrow S_0 - \{\sigma\}$; $I_{S_0} \leftarrow I_{S_0} - \{\vec{\xi_\sigma}\}$;
      else if $\vec{\xi_\theta} \prec_{better} \vec{\xi_\sigma}$ **then return** S_0;
    $S_0 \leftarrow S_0 \cup \{\theta\}$; $I_{S_0} \leftarrow I_{S_0} \cup \{\vec{\xi_\theta}\}$;
    return S_0;
  for *each* $\vec{\xi}_{x=d}$ *in* I_H **do**
    if $d \notin D(x)$ **then**
      $I_H \leftarrow I_H - \{\vec{\xi}_{x=d}\}$;
  $I_H \leftarrow$ **ch2c**(H, V, D, I_H);
  choose variable $x \in V$ for which $|D(x)| \geq 2$;
  $W \leftarrow D(x)$;
  for *each* $d \in W$ **do**
    if **go**($\vec{\xi}_{x=d}$, S_0, I_{S_0}, $\prec_{better}$) **then**
      $S_0 \leftarrow$ **bb_solv**($\{H_0 \wedge x = d, H_1, \ldots, H_n\}$, I_H, V, D, S_0, I_{S_0}, $\prec_{better}$);
  return S_0;
end

Figure 8: A Branch-and-bound CH Solver with Pruning

```
go(ξ⃗_c, S_0, I_{S_0}, ≺_better)
begin
    for each θ ∈ S_0 do
        if ξ⃗_c ≺_better ξ⃗_θ then return false;
    return true;
end
```

Figure 9: A subroutine.

```
cal_error_value(H, θ, ξ⃗_θ)
begin
    for l ← 1 to n do
        for k ← 1 to |H_l| do
            let c be the k^th constraint in H_l;
            ξ_θ^l_k ← e(cθ);
    return ξ⃗_θ;
end
```

Figure 10: A subroutine to calculate error value.

performance. Crisp propagation is faster. Lines 5 to 13 deal with the case of a leaf node. Here there is a call to subroutine **cal_error_value** that computes the error $e(c\theta)$ for each θ. The CH-2-C algorithm is invoked between lines 13 and 14. Lines 14 to 17 perform the basic variable instantiation (or searching) recursively. The call to the subroutine **go** determines whether the error indicator store of the variable assignment of the selected branch in I_H of the current node is not worse than the error indicator of each of the collected solutions so far. This is the "bounding" part of the algorithm to determine if the search should proceed down the branches at a node.

7 Experiments

DeltaStar is only a theoretical framework [FB02], and clp(FD,S) cannot in the current implementation deal with hierarchies. We compare the performance of our proposed solver (S_c) with generate-and-test (S_g), basic branch-and-bound (S_b), and the reified constraint approach (S_r) by Lua (the Lua's solver hereafter) [Lua01].

Lua [Lua01] proposed a method to transform constraint hierarchy into ordinary constraint system. In this approach, an error value (a value returned by error function) is related to a special type of constraint called *reified constraint* (or *error constraint*) and it is used to replace the error function. A constraint c is associated with a variable ϵ_c where $\epsilon_c \geq 0$. This variable represents the degree of satisfaction of constraint c and this formulation preserves the original meaning in the theory of CH ($c\theta$ holds $\Leftrightarrow \epsilon_c = 0$). For example, given a constraint c and a variable ϵ_c. It is possible to replace the trivial error function by using reified constraint such as *Reified*(c, ϵ_c) provided by many CLP systems. A value 0 will be assigned to ϵ_c if the constraint c is satisfied, or else, a value 1 will be assigned to ϵ_c. Since it is possible to use reified constraint and variable ϵ_c to represent the error function and error value respectively, it is possible to use an *error vector* E_C to store all the combined error values of the constraints. The form of error vector E_C is a tuple of variables, $\langle E_{C_1}, \ldots, E_{C_n} \rangle$ where each E_{C_i} represents the combined error value of the constraints in C_i (or H_i). Intuitively, E_{C_i} represents the combined error values returned by combining function g in the original formulation in CH. For example, it is possible to replace the combining function g of *weighted-sum-better* by an *error combining constraint* such that $E_{C_i} = \sum_{c \in C_i} w_c \epsilon_c$ and w_c is the weight of constraint c. It is easy to transform the other combining functions (g for *worst-case-better* and *least-squares-better*) in a similar way. By using different error combining constraint, it is possible to define *globally-better* as follows.

$$\textit{globally-better}(E_C, E'_C) = b(E_C, E'_C, 1), \text{ where}$$

$$b(E_C, E'_C, i) = \begin{cases} \textit{false} & \text{if } i > n \\ E_{C_i} < E'_{C_i} \vee (E_{C_i} = E'_{C_i} \wedge b(E_C, E'_C, i+1)) & \text{otherwise} \end{cases}$$

However, it is unclear how the *locally-better* comparator can be implemented using this approach. We note that reified constraints are closely related to the meta-constraints proposed by Petit *et al.* [PRB00].

Since both Lua's solver and ours are based on a branch-and-bound backbone, we first implement a solver engine S_g ("g" stands for "Generate-and-Test"), which searches using ILOG's default *goal* definition, in ILOG Solver 4.4 in a generate-and-test fashion. In order to provide a basic Branch-and-Bound solver (without CH-2-C enforcement) for comparison, we define an alternative ILOG goal G_b to obtain S_b ("b" stands for "Branch-and-Bound"). The *goal* G_b follows the same searching order as the default *goal*, but compares the errors of the current best valuations and the accumulated errors so far at each search node. The search proceeds if the accumulated errors is not "worse" than the errors of the current best valuations. Otherwise, the search is backtracked to another branch as in the ordinary Branch-and-Bound algorithm.

Our proposed solver S_c ("c" stands for "CH-2-C") is obtained by implementing additional functions and an alternative *goal* definition G_c in S_g. The *goal* G_c follows the same searching order as the default *goal*, but enforces CH-2-C at each search node. While the input to our solvers is a CH, the input to Lua's solver S_r ("r" stands for "reified constraint") is a CSP with reified constraints for implementing a specific comparator and error function. The solver S_r also requires an alternative *goal* G_r that implements the *reified arithmetic comparison propagators* and *reified logic operation propagators*. In the solver S_r, the program variables are instantiated during search. However, the value of each variable ϵ_c, corresponding to a constraint c, is obtained automatically by reified propagation. The value of each variable (or *error vector*) E_{C_i}, which stores the combined error values of the reified constraints in level i, is obtained by normal propagation of an *error combining constraint*. The values stored in the error vectors will be compared to the values stored in the current best error vectors at each search node. Similarly, the search proceeds if the error vectors are not "worse" than the current best error vectors. Otherwise, the search is backtracked to another branch. Our comparison ensures *fairness* since all four solvers share the same backbone.

7.1 Experimental Setup

There is a lack of benchmarks for finite domain CH in the public domain. For simplicity, our testing CH instances are comprised of randomly generated unary and binary arithmetic constraints, which involve the usual arithmetic operations $(+, -, \times, /)$ and relations $(<, \leq, \neq, =)$. Thus the generated constraints can be both linear and non-linear. More in details, we first generate whether the constraint is unary or binary, then we generate the coefficients of each of the variables and a constant. In case of a binary constraint, we generate further the operators used to combined the two terms.

Each *test data suite* consists of three parts to test the effect of *variable domain size*, *number of variables*, and *number of hierarchy levels* on the performance of the solvers. In each part, four sets of CHs, each of which contains 15 randomly generated problem instances. Thus, each test data suite has 180 problem instances. All problem instances have no hard constraints (in level 0) to make the problem more "difficult" to solve.

The first part consists of problem sets: P'_1, P'_2, P'_3, and P'_4, each of which contains 15 problem instances. The number of variables and constraints are fixed ($|V| = 5$, $H = \{H_0, H_1, H_2\}$, $|H_0| = 0$, and $|H_1| = |H_2| = 5$) across all instances, while problems in the same set share a specific domain size: P'_i has variable domains of size $10i$ for $i \in \{1, 2, 3, 4\}$.

The second part consists of problem sets: P''_1, P''_2, P''_3, and P''_4. The domain size and the number of constraints are fixed ($\forall x \in V, |D(x)| = 5$, $H = \{H_0, H_1, H_2\}$, $|H_0| = 0$, and $|H_1| = |H_2| = 5$) across all instances, while problems in the same set share a specific number of variables: P''_i has $2(i+1)$ number of variables for $i \in \{1, 2, 3, 4\}$.

The third part consists of problem sets: P'''_1, P'''_2, P'''_3, and P'''_4. The number of variables and the domain size are fixed ($|V| = 5$ and $\forall x \in V, |D(x)| = 20$) across all instances, while problems in the same set share a specific number of hierarchies (or constraints): P'''_i has $i+1$ non-required levels for $i \in \{1, 2, 3, 4\}$ such that $|H_0| = 0$ and $\forall j \in \{1, \ldots, i+1\}, |H_j| = 5$.

We benchmark the performance of our solver S_c by conducting two different experiments. In the first experiment, we want to gain a high level view of the time efficiency, the memory requirement, and the pruning power of the various comparators our solver. We would also like to investigate how the CH-2-C algorithm is compared to the other approaches in terms of the measured performance. To ensure variety of test cases, we generate a different test data suite for each comparator. In the second experiment, we want to study the performance, in terms of execution time, of our solver among the different comparators against the other approaches. The purpose is to identify whether the CH-2-C algorithm is strong/weak in dealing with particular comparator(s). Therefore, we generate one test data suite and use the same suite for all comparators. For simplicity reason, we apply the trivial error function to test the solvers in both experiments.

For global comparators, we benchmark the performance of our solver by comparing S_c with S_g, S_b, and S_r accordingly. Since it is unclear how the *locally-better* can be implemented using Lua's reified constraint approach, we only compare S_c with S_g and S_b for the local comparator.

Our experiments are conducted on Sun Ultra 5/400 workstations with 256MB RAM. We use the default variable and value ordering in ILOG Solver (which are essentially natural variable ordering according to the variable index and least-to-largest for value), and search for all optimal (undominated) solutions.

We collect the following information of solvers S_g, S_b, S_r, and S_c from all the experiments:

- The execution time T_i
- The maximum memory requirement M_i
- The number of leaf nodes visited L_i in searching
- The number of choice points C_i in searching

7.2 Efficiency, Memory Requirement and Pruning Power

This test consists of a test data suite for each comparator, totaling 720 (4×180) problem instances. Results are reported in Table 1, which gives both the mean

Comparison between S_g and S_c				
Comparator	T_g/T_c	M_g/M_c	L_g/L_c	C_g/C_c
l-b	176 (9)	0.88 (0.88)	1860 (70)	230 (13)
w-s-b	206 (9)	0.89 (0.88)	3149 (86)	249 (13)
w-c-b	54 (4)	0.89 (0.88)	517 (22)	80 (7)
l-s-b	105 (5)	0.89 (0.88)	2394 (33)	130 (9)

Comparison between S_b and S_c				
Comparator	T_b/T_c	M_b/M_c	L_b/L_c	C_b/C_c
l-b	46 (7)	0.88 (0.88)	215 (31)	70 (7)
w-s-b	128 (6)	0.89 (0.88)	1263 (22)	117 (6)
w-c-b	41 (3)	0.89 (0.88)	291 (13)	53 (5)
l-s-b	18 (3)	0.89 (0.88)	152 (16)	17 (6)

Comparison between S_r and S_c				
Comparator	T_r/T_c	M_r/M_c	L_r/L_c	C_r/C_c
w-s-b	87 (3)	1.22 (1.25)	1142 (18)	94 (3)
w-c-b	37 (3)	1.19 (1.23)	258 (11)	47 (3)
l-s-b	11 (2)	1.19 (1.25)	123 (11)	12 (2)

Table 1: A summary of the comparative performance of S_c.

and median (in brackets) performances. The table is divided into three sub-tables, each reports results of the S_c solver as compared to the S_g, S_b, and S_r solvers respectively. Each row in a sub-table corresponds to the results for a particular comparator. Performances are measured in terms of ratios in each column: T_x/T_c, M_x/M_c, L_x/L_c, and C_x/C_c, where $x \in \{g, b, r\}$. A number greater than 1 indicates that the S_c solver is better. As a reference, the absolute time performance of the solvers range from 0.01 to around 8000 seconds.

In terms of time, the S_c solver is in general faster than the other solvers by 1 to 2 orders of magnitude in mean performance, and a few times faster in median performance. The mean and median results agree in trends, but not in magnitudes. The discrepancy suggests that we have test instances in which the S_c solver is much more efficient. This could be due to the fact that some benchmarks generated are significantly more difficult for the solvers. As expected, the S_g solver performs the worst, followed by the S_b and the S_r solvers respectively.

In terms of memory consumption, the S_c solver incurs a slightly larger overhead over the S_g and S_b solvers, since extra memories are needed to store the consistency information in solver S_c. On the other hand, the S_r solver requires even more memory, mainly to handle the extra reified constraints for error calculations. There is little discrepancy between the mean and the median performance.

A choice point corresponds to a branching node in a search tree. The number of leaf nodes visited and the number of choice points are good indicators of the pruning power of the solvers. As expected, the S_c solver is significantly better than the S_g and S_b solvers, which support no local consistency notions for pruning. Compared to the S_r solver, S_c still flares well, since consistency maintained in reified constraints is relatively weak for pruning. The discrepancy in the magnitudes of the mean and median results agrees with that in the time comparison: some generated benchmarks are significantly more difficult and the S_c solver is able to solve these instances much more better than the other solvers.

In summary, the S_c solver is faster than the other approaches since it is able to prune larger part of the search tree. At the same time, the memory requirement of S_c is basically on par with the other solvers.

7.3 Performance Among Different Comparators

This experiment evaluates the solvers under different settings: change in variable domain size, number of variables, and number of hierarchy levels.

7.3.1 Domain Size

In this part of the experiment, P'_1, P'_2, P'_3, and P'_4 contains benchmarks of increasing domain sizes with number of variables and number of hierarchy levels being kept constant. Results are reported in Table 2, which give both the mean (upper table) and the median (lower table) performances. Again, performances are measured in terms of ratios in each sub-tables: T_g/T_c, T_b/T_c, and T_r/T_c respectively. Each column in a sub-table corresponds to a comparator, while each row corresponds to a problem set (of 15 instances) with the same variable domain size.

	T_g/T_c (Mean)				T_b/T_c (Mean)				T_r/T_c (Mean)		
CHs	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*
P'_1	8	5	7	10	6	4	6	7	5	4	5
P'_2	36	15	37	13	18	22	19	9	9	19	9
P'_3	267	67	261	171	121	47	123	31	113	42	115
P'_4	385	72	342	76	37	35	39	23	17	27	18

	T_g/T_c (Median)				T_b/T_c (Median)				T_r/T_c (Median)		
CHs	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*
P'_1	4	1.5	4	4	2	1.3	2	2	1	0.8	0.9
P'_2	7	0.6	7	3	6	0.6	6	1.3	5	0.6	5
P'_3	74	6	72	22	2	5	2	4	0.9	5	0.9
P'_4	13	5	13	12	5	3	5	3	2	2	2

Table 2: A comparison by varying the domain size.

7.3.2 Number of Variables

In this part of the experiment, P''_1, P''_2, P''_3, and P''_4 contains benchmarks of increasing number of variables with variable domain size and number of hierarchy levels being kept constant. Results are reported in Table 3, which give the mean and median performances respectively.

	T_g/T_c (Mean)				T_b/T_c (Mean)				T_r/T_c (Mean)		
CHs	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*
P''_1	1.2	0.9	1.3	1.2	1.2	1.3	1.5	1.4	1.1	1.1	1.4
P''_2	6	3	6	5	5	3	5	4	5	3	5
P''_3	7	3	7	4	5	4	5	3	4	4	4
P''_4	24	8	24	26	3	7	3	5	1.4	6	1.4
	T_g/T_c (Median)				T_b/T_c (Median)				T_r/T_c) (Median)		
CHs	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*
P''_1	0.8	0.6	1	1	1.3	1	1.2	1.3	1.2	0.8	1
P''_2	3	1.4	3	2	2	1.4	2	1.5	1.4	1.4	1.3
P''_3	3	1.5	2	2	2	1.4	2	1.4	2	1.4	1.7
P''_4	4	1.4	3	2	3	0.7	3	1.4	1	0.6	1

Table 3: A comparison by varying the number of variables.

7.3.3 Number of Hierarchy Levels

In this part of the experiment, P'''_1, P'''_2, P'''_3, and P'''_4 contains benchmarks of increasing number of hierarchy levels with variable domain size and number of variables being kept constant. Results are reported in Table 4, which give the mean and median performances respectively.

	T_g/T_c (Mean)				T_b/T_c (Mean)				T_r/T_c (Mean)		
CHs	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*
P'''_1	146	108	151	122	44	44	44	32	37	39	39
P'''_2	209	130	212	116	51	116	50	34	38	104	39
P'''_3	232	168	219	50	42	121	44	21	31	113	29
P'''_4	122	154	124	75	58	132	60	26	51	128	52
	T_g/T_c (Median)				T_b/T_c (Median)				T_r/T_c (Median)		
CHs	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*
P'''_1	24	6	26	27	3	4	3	4	2	4	2
P'''_2	52	12	54	29	6	10	6	5	6	9	6
P'''_3	63	29	70	10	6	8	5	3	5	6	5
P'''_4	44	52	47	15	11	53	11	4	9	51	9

Table 4: A comparison by varying the number of hierarchy levels.

7.3.4 Discussions

The CH-2-C algorithm incurs overhead in the branch-and-bound search. For the larger problems, the extra effort paid by the CH-2-C algorithm at each search node is demonstrated worthwhile. This result is in line with the behavior of embedding classical consistency techniques in basic tree search in solving classical CSPs. In general, the data in Tables 2–4 also exhibit a similar increasing trend in the T_x/T_c ratio, where $x \in \{g, b, r\}$. In other words, the T_c solver gives more advantages over the other approaches as the problems grow in size and difficulty. There are two points to note.

First, sometimes the ratios increase and then drop, for example, in the case of the T_g/T_c ratios for the *l-b* comparator in Table 2. This looks like a "phase transition" phenomenon, but we note that the results reported are performance ratios but not absolute execution time. In fact, we can observe increase in execution time in the raw experimental data as the problems grow in size and difficulty, as expected. We conjecture that this increase-decrease phenomenon is a result of the large variance in the experimental results, as observed in the discrepancy between the mean and the median results. This is also the cause for a few median performance ratio results being less than 1. Such large variance and the increase-decrease phenomena are topics for future research.

Second, in general, the advantages of the S_c solver over the other approaches are the worst for the *w-c-b* and *l-b* comparators, which are more likely to find two error indicators being incomparable. Thus, there is less opportunities for pruning with these comparators.

8 Related Work

Many efficient algorithms have been proposed to solve CH, such as DeltaBlue [FBMB90], SkyBlue [San94], DETAIL [HMT$^+$94], Indigo [BAFB96], Generalized Local Propagation [HMY96], and Ultraviolet [BFB98], apply Local Propagation [SS79]. Besides, Cassowary and QOCA algorithms [BMSX97], adapting the Simplex algorithm [NM65], can also solve CHs efficiently. However, they are designed for the real number domain. We focus on finite domain CHs solving techniques; we can categorize the techniques into three different approaches.

First, the Incremental Hierarchical Constraint Solver (IHCS) [MBC93] proposes to transform a given constraint hierarchy into a set of *best configurations* (a set of constraints). Therefore, a given CH can be transformed into a set of classical CSPs. However, it can only find *l-b* solutions using the trivial error function. The second approach is to transform CHs into ordinary constraint systems based on *reified constraint propagation* [Lua01]. This approach can only find solutions for *global comparators* (*w-s-b*, *w-c-b*, and *l-s-b*). The last is the refining approach used by DeltaStar [FBWB92]. It is a generic finite domain CH solver which can find solutions for arbitrary comparators in theory. However, it recomputes the solution in each recursive step causing significant overhead. Hence, it is used only as a general and theoretical framework for solution, from which efficient algorithms, such as DeltaBlue (only equality constraints) and Cassowary (a very restricted finite domain subsolver), are inspired and designed for some subset of the general problem [FB02]. In addition, to the previous one, Henz *et al.* [HYF$^+$04] used local search methods to solve constraint hierarchies over the Finite Domains.

This paper is also related to many work in soft constraint processing. These frameworks demonstrate how information gained through local consistency checking during preprocessing can be used to enhance branch-and-bound search using local computations as global bounds. In fact, when dealing with Constraint Hierarchies, *w-s-b* and *l-s-b* can be modeled by Weighted CSPs [Lar02, LS04, LS03] and *w-c-b* by fuzzy CSPs [Coo03] (However notice that WCSP cannot model all global comparators). Both Weighted CSPs and Fuzzy CSPs are instances of the Valued CSPs framework [Sch00, Coo03, CS04]. The same idea is used by Larrosa and Schiex [LS99]. From the last level to the first, one must multiply the base error at each level in such a way that the smallest strictly positive error at one level k is larger than the largest error at level $k+1$ multiplied by the number of constraints at level $k+1$.

The bounds computed by these works are better than ours when we restrict our computations to only 2 consistency levels, and to a specific comparator. Our framework are somewhat more general. We are able to compute bounds for CH with varying levels of consistency (from 1 to k) and *without fixing a priori a comparator*.

In addition, Valued CSPs can only model global comparators. In fact, the locally better comparator induces a partial order structure that cannot be used in the Valued CSP framework, which is based on total orders.

The Constraint Hierarchies framework can sometimes be more natural in modeling applications. Examples come from the area of animation [BG95], planning [Bar97], web documents [LMS99] and also routing [YYA02].

9 Conclusion

We formally define constraint hierarchy k-consistency (CH-k-C), based on error indicators. Incorporating a CH-2-C enforcement algorithm in a branch-and-bound algorithm, we obtain a general finite domain CH solver, which works for arbitrary comparators. Search space is pruned by utilizing the error information generated by the CH-2-C algorithm. Experiments confirm the efficiency of our research prototype, which brings us one step towards practical finite domain CH solving.

There is room for future research. First, our implementation and even the CH-2-C algorithm are hardly optimized. They have much scope for improvement. Second, we test our solver on random problems, and observe large variance in the performance ratio results, which is worth investigation. It will also be interesting to study phase transition phenomena similar to that identified by Larrosa and Meseguer for MAX-CSPs [LM96]. In addition, experiments on more structured problems and real-life problems are needed. Third, our consistency-based and Lua's reified constraint approaches do not compete. It would be interesting to study if the two methods can be combined to produce more pruning. Fourth, the efficiency of branch-and-bound algorithms can be sensitive to variable and value orderings. It is worthwhile to investigate good ordering heuristics specific to the CH-2-C and the branch-and-bound algorithms. Fifth, the current proposal of our solver guarantees the correctness of local and global comparators. In addition, it is easy to check that our solver can support the *regional comparator* [WB93]. The existing comparators, although rigorously and mathematically defined, might be too general for a specific real-life situation. It would be interesting to introduce new comparators that should be of particular relevance to real-life problems and applicable to our solver.

References

[BAFB96] A. Borning, R. Anderson, and B. Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *Proceedings of the 1996 ACM Symposium on User Interface Software and Technology*, pages 129–136, 1996.

[Bar97] R. Bartak. A generalized framework for constraint planning, 1997.

[BBS01] G.J. Badros, A. Borning, and P.J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction*, 8(4):267–306, 2001.

[BCHL03a] S. Bistarelli, P. Codognet, H.K.C. Hui, and J.H.M. Lee. Solving finite domain constraint hierarchies by local consistency and tree search. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1364–1365, 2003.

[BCHL03b] Stefano Bistarelli, Philippe Codognet, Kin Chuen Hui, and Jimmy Ho-Man Lee. Solving finite domain constraint hierarchies by local consistency and tree search. In *Proceedings 9th International Conference Principles and Practice of Constraint Programming (CP 2003)*, volume 2833, pages 138–152. Springer, 2003.

[BFB98] A. Borning and B. Freeman-Benson. Ultraviolet: A constraint satisfaction algorithm for interactive graphics. *Constraints: An International Journal*, 3(1):9–32, 1998.

[BFBW92] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.

[BFR95] C. Bessière, E.C. Freuder, and J.C. Régin. Using inference to reduce arc consistency computation. In *Proceedings of IJCAI'95*, pages 592–598, 1995.

[BG95] Jean-Francis Balaguer and Enrico Gobbetti. Supporting interactive animation using multi-way constraints. In *Eurographics Workshop on Programming Paradigms in Graphics*, pages 37–48, 1995.

[Bis04] S. Bistarelli. *Semirings for Soft Constraint Solving and Programming*, volume 2962 of *Lecture Notes in Computer Science*. Springer, 2004.

[BMR97] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint solving and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[BMSX97] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 87–96, 1997.

[BR98] S. Bistarelli and F. Rossi. About arc-consistency in semiring-based constraint problems. In *AMAI98 (Symposium on Mathematics and Artificial Intelligence)*, 1998.

[Coo03] Martin C. Cooper. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3):311–342, mar 2003.

[CS04] Martin Cooper and Thomas Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1–2):199–227, 2004.

[FB02] B. Freeman-Benson. Efficiency of DeltaStar. Private Communication, April 2002.

[FBMB90] B. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, 1990.

[FBWB92] B. Freeman-Benson, M. Wilson, and A. Borning. DeltaStar: A general algorithm for incremental satisfaction of constraint hierarchies. In *The 11th Annual IEEE Phoenix Conference on Computers and Communications*, pages 561–568, 1992.

[GS96] S.A. Grant and B.M. Smith. The phase transition behavior of maintaining arc consistency. In *Proceedings of ECAI'96*, pages 175–179, 1996.

[HMT+94] H. Hosobe, K. Miyashita, S. Takahashi, S. Matsuoka, and A. Yonezawa. Locally simultaneous constraint satisfaction. In *Proceedings of PPCP'94*, pages 51–62, 1994.

[HMY96] H. Hosobe, S. Matsuoka, and A. Yonezawa. Generalized local propagation: A framework for solving constraint hierarchies. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 237–251, 1996.

[HYF+04] Martin Henz, Roland H. C. Yap, Lim Yun Fong, Lua Seet Chong, J. Paul Walser, and Shi Xiao Ping. Solving hierarchical constraints over finite domains with local search. *Ann. Math. Artif. Intell.*, 40(3–4):283–302, 2004.

[Lar02] J. Larrosa. Node and arc consistency in weighted CSP. In *Proceedings of AAAI'02*, pages 48–53, 2002.

[LM96] Javier Larrosa and Pedro Meseguer. Phase transition in MAX-CSP. In *ECAI'96*, pages 190–194, 1996.

[LMS99] R. Lin, K. Marriott, and P. Stuckey. Flexible font-size specification in web documents, 1999.

[LS99] Javier Larrosa and Thomas Schiex. Semiring-based CSPs and valued CSPs: Frameworks, properties,and comparison. *Constraints*, 4(3):199–240, 1999.

[LS03] Javier Larrosa and Thomas Schiex. In the quest of the best form of local consistency for weighted CSP. In *Proceedings of IJCAI 2003*, pages 239–244, 2003.

[LS04] Javier Larrosa and Thomas Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1–2):1–26, 2004.

[Lua01] S.C. Lua. Solving finite domain hierarchical constraint optimization problems. Master's thesis, Department of Computer Science, National University of Singapore, 2001.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.

[MBC93] F. Menezes, P. Barahona, and P. Codognet. An incremental hierarchical constraint solver. In *First Workshop on Principle and Practice of Constraint Processing*, 1993.

[NM65] J.A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7:308–313, 1965.

[PRB00] C. Petit, J-C. Regin, and C. Bessiere. Meta-constraints on violations for over-constrained problems. In *Proceedings of ICTAI-2000*, pages 358–365, 2000.

[San94] M. Sannella. The SkyBlue constraint solver and its applications. In V.A. Saraswat and P.V. Hentenryck, editors, *Proceedings of the First Workshop on Principles and Practice of Constraint Programming*. MIT Press, 1994.

[Sch00] Thomas Schiex. Arc consistency for soft constraints. In *Proc. 6th International Conference on Principles and Practice of Constraint Programming (CP2000)*, volume 1894, pages 411–424. Springer, 2000.

[SS79] G.L. Steele and G.J. Sussman. Constraints. In *APL conference proceedings part 1*, pages 208–225, 1979.

[WB93] M. Wilson and A. Borning. Hierarchical constraint logic programming. *Journal of Logic Programming*, 16:277–318, 1993.

[YYA02] Mohamed Younis, Moustafa Youssef, and Khaled Arisha. Energy-aware routing in cluster-based sensor networks. In *IEEE/ACM MASCOTS 2002*, October 2002.