Solving Finite Domain Constraint Hierarchies by Local Consistency and Tree Search

By

Hui Kau Cheung Henry

A Thesis Submitted in Parital Fulfilment

of the requirements for the degree of

Master of Philosophy

in

Computer Science & Engineering

June 2002

摘要

現在我們要解決的是 *有限域的等級限制性* 問題。在論文中，我們提出以 *誤差指標* 的概念再形成 *等級限制性架構*〔CH〕。我們模仿 SCSP 架構裏對 *局部的一致性* 的概括看法爲 *等級限制 k 一致性*〔CH-*k*-C〕下定義及提供 *等級限制弧一致性*〔CH-*2*-C 又稱爲 CHAC〕的演算法。*等級限制性* 問題也是 *最佳化* 問題。我們說明了如何結合 *等級限制弧一致性* 的演算法與通常的 *分支及約束* 的演算法成爲一個 *有限域的等級限制性解答者*〔又稱爲 *等級限制弧一致性的分支及約束解答者*〕。我們討論了現有的 *有限域的等級限制性解答者* 之局限。實驗證明我們提出了有效率及穩定的解答者模範。我們提出的方法在 *局部的比較* 與及 *總體的比較* 也起了作用，在這方面的比較之下我們所提出的方法和其他的 *有限域的等級限制性解答者* 是有所不同。另外，我們的解答者能夠支援任何的 *誤差函數*。

Abstract

The task at hand is to tackle constraint hierarchy problems in the finite domain. In this thesis, 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, namely *constraint hierarchy arc-consistency* (CHAC), enforcement algorithm. CH problems are optimization problems. We demonstrate how the CHAC algorithm can be seamlessly integrated into the ordinary Branch-and-Bound algorithm to make it a finite domain CH solver (Branch-and-Bound CHAC solver). We discuss the limitation of existing finite domain CH solvers. 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.

Acknowledgments

I am deeply indebted to my supervisor, Prof. Jimmy Ho-man Lee, for his continuous guidance, support, and encouragement throughout the period of my master's degree program. I am very grateful for his invaluable advice, which helps me to finish the work.

My gratefulness further goes to Prof. Lauren Lai-wan Chan for her belief in my ability to be a researcher. This work could not have been done without her encouragement.

I would like to express my thanks and appreciation to my external examiner, Prof. Phillippe Codognet, for his thoughtful questions at my defense. His questions focused on the absence of my own voice in this thesis.

I would like to thank Prof. Ho-fung Leung and Prof. Kwong-sak Leung, who provided many helpful comments on this research.

I would also like to thank Prof. Armin Wolf, Dr. Bjorn Freeman-Benson, Prof. Martin Henz, Prof. Roman Barták, and Dr. Yan Georget for e-mail discussions. They provided many constructive comments and suggestions on this research.

I cannot thank my mother enough for her support, patience, and love during the period of my master's degree program and especially during writing of this thesis.

Last but not the least, I would like to give a special thank you to Terry Sze-hang Chui for his continuous help and numerous suggestions. I would also like to thank Bernard Boon-lub Loo, Clotho Wing-yee Tsang, Grace Chi-fun Yuen, Peter Chun-kit Pang, and Tony Chun-yin Mak. This thesis could not have been written without their support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The Constraint Hierarchy (CH) framework [13] is a general framework for the specification and solutions of over-constrained problems, which have no solution in the classical sense, caused by some of the constraints contradicting others. Originating from research in interactive user-interface applications [19, 45], the CH framework attracts much effort in the design of efficient solvers in the real number domain [22, 46, 34, 11, 33, 12, 2]. To extend the benefit of the CH framework to also discrete domain applications, such as timetabling and resource allocation, some finite domain CH solving techniques [23, 39, 3, 8, 36] have been proposed. Incremental Hierarchical Constraint Solver (IHCS) [39] is a finite domain CH solver, but it can only find *locally-predicate-better* solution [13]. DeltaStar [23] is also a CH solver, it does not restrict to finite domain and it can find solution for arbitrary comparators [13, 51]. It is built upon a flat constraint solver to filter "worse" valuations recursively from the highest level of the hierarchy. However, DeltaStar requires too many memories to store possible valuations in practice. Lua [36] has proposed a reified constraint approach to solve finite domain CH for global comparators. This idea is realized by combining reified constraint propagation and the ordinary Branch-and-Bound algorithm [42]. This approach is based on existing technique, which is clever and clean, but reified constraint propagation is in fact a relatively weak propagation.

Our work is motivated by the demand of a general and efficient finite domain CH solver. The main idea is to combine consistency techniques and tree search. Central to the thesis 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, namely *constraint hierarchy arc-consistency* (CHAC), of a CH problem. While classical consistency algorithms [37] aims to reduce the size of constraint problems, our CHAC algorithm works by explicating error information that is originally implicit in CH problems. Such error information is used to represent the "goodness" of a value in the corresponding variable domain. We also suggest ways of utilizing such extracted information to help prune non-fruitful computation in the ordinary Branch-and-Bound algorithm, which forms the basis of our finite domain CH solver. Unlike other finite domain CH solvers, our proposed solver is applicable to arbitrary comparators. We have constructed a prototype of the solver, and performed experiments on a set of randomly generated CH problems. Our experiments confirm the efficiency and robustness of our proposal. In addition, experiments show that our solver can produce more pruning than Lua's solver in most of the time.

1.2 Organizations of the Thesis

The thesis is organized as follows. In Chapter 2, we gives necessary background and related work. Since our work is directly related to consistency technique and tree search, we outline the concepts of the classical notion of arc-consistency (AC) [37] and the general notion of semiring-based arc-consistent (SAC) [10], as well as the ordinary backtracking tree search [41, 27] and the Branch-and-Bound algorithm [42]. We give a detailed discussion of the over-constrained problems [25, 47, 44] and in particular CH [13]. We also present the existing techniques in solving finite domain CH [39, 23, 8, 36, 3]. In Chapter 3, we present an equivalent redefinition of the CH framework using the notion of error indicators, which is central in the definition of CH-k-C and the associated enforcement algorithm in particular for CH-2-C (or CHAC). The correctness of the CHAC algorithm is established. In Chapter 4, we show how to combine the CHAC algorithm and the ordinary Branch-and-Bound algorithm into our proposed finite domain CH solver, which is called Branch-and-Bound CHAC solver. The correctness of our solver is established. We randomly generate problem instances as the benchmark problems for our solver. We provide detail discussion of our experimental results. In Chapter 5, we summarize our contributions and discuss some directions for further research.

Chapter 2

Background

This chapter provides the theoretical background to the thesis. The basic definitions of *Constraint Satisfaction Problems* (CSPs) [38], *Over-Constrained Constraint Satisfaction Problems* [25, 47, 44, 9, 48], and *Constraint Hierarchies* (CHs) [13] are presented. We also present the existing techniques [39, 36, 8, 23, 3] for solving CHs.

2.1 Constraint Satisfaction Problems

A *constraint satisfaction problem* (CSP) is a framework. Some real-life applications, such as the N-Queens Problem [1] and the Map Coloring Problem, can be modeled as instances of CSP. We present the basic definition of CSP based on Marriott and Stuckey [38]. A *constraint domain* is a tuple $\langle \mathcal{D}, \mathcal{F}, \mathcal{R} \rangle$, where $\mathcal{D}$ is a set of values, $\mathcal{F}$ is a set of operators on $\mathcal{D}$, and $\mathcal{R}$ is a set of relations on $\mathcal{D}$. For example, let $\mathcal{D}$ be the set of all integers. The usual operators on $\mathcal{D}$ are $+$, $-$, $\times$, and $\div$, and the usual relations on $\mathcal{D}$ are $=$, $\neq$, $>$, $\geq$, $<$, and $\leq$. A CSP is a tuple $\langle V, D, C \rangle$, where V is a set of *variables*, a *domain* D is a set of *variable domains*, and C is a set of *constraints*. A variable is an unknown. Each variable $x \in V$ can be assigned a value from its variable domain $D(x)$, where $D(x) \subseteq \mathcal{D}$.

A constraint is a relation among the variables. The constraint domain defines the syntax of a constraint, because it specifies the operators and relations on $\mathcal{D}$. An n-ary constraint is a relation among n variables. Hence, an n-ary constraint $c \in C$ is a relation over $\mathcal{D}^n$. For example, the arithmetic constraint "$x+y=2$" is a relation between two variables (x and y). A *finite domain* CSP is a CSP such that $\mathcal{D}$ is a finite set. We mainly focus on finite domain constraint satisfaction throughout the thesis.

A *valuation* for a set of variables V, denoted by θ, is an assignment of values from the corresponding variable domains to the variables in V. If $V = \{v_1, \ldots, v_n\}$, then θ can be written as $\{v_1 \mapsto d_1, \ldots, v_n \mapsto d_n\}$ which means each variable $v_i \in V$ is assigned with a value $d_i \in D(v_i)$. A boolean value (*true* or *false*) is returned by applying a valuation θ to a constraint c, denoted by $c\theta$. When applying a valuation θ to a constraint c and *true* is returned, this means the constraint c is satisfied by the valuation θ. We say that $c\theta$ holds. When applying a valuation θ to a constraint c and *false* is returned, this means the constraint c is violated by the valuation θ. We say that $c\theta$ does not hold. Let $vars(c)$ denotes the set of variables occurring in the constraint c. If θ is a valuation for V ($vars(c) \subseteq V$) and for each $c \in C$ such that $c\theta$ holds, then the valuation θ is the *solution* to the CSP.

A *constraint solver* (or *solver*) is an algorithm to find the solution to a CSP. Since we limit our scope to finite domain (the set $\mathcal{D}$ is a finite set), we mainly present the finite domain constraint satisfaction techniques, finite domain solvers, in this thesis.

2.1.1 Local Consistency Algorithm

The notion of *local consistency* [37, 24] deals with the situation when a CSP contains inconsistency information. Mackworth [37] defines *node-consistency*, *arc-consistency*, and *path-consistency* that characterize local consistency of a CSP. Since maintaining node-consistency and arc-consistency during search is proven to be worthwhile technique [6, 30], we mainly focus on node-consistency and arc-consistency in this thesis.

The formal definition of node-consistent can be defined as *"A primitive constraint c is node consistent with domain D if either $|vars(c)| \neq 1$ or, if $vars(c) = \{x\}$, then for each $d \in D(x)$, $\{x \mapsto d\}$ is a solution of c. A CSP with constraint $c_1 \wedge \cdots \wedge c_n$ and domain D is node consistent if each primitive constraint c_i is node consistent with D for $1 \leq i \leq n$"* [38]. In other words, a CSP is node-inconsistent if there exists a value $d \in D(x)$ such that the valuation $\{x \mapsto d\}$ violates any unary constraint c ($|vars(c)| = 1$). When the node-inconsistent values are detected, these values should be removed. A node-consistency algorithm [38] is shown in Figure 2.2. The subroutine **nc_primitive** (in Figure 2.1) removes the node-inconsistent values from the variable domains by determining all the unary constraints.

```
nc_primitive($c$, $D$)
begin
1   if $|vars(c)| = 1$ then
2       let $\{x\} = vars(c)$;
3       $D(x) \leftarrow \{d \in D(x) \mid \{x \mapsto d\}$ is a solution to $c\}$;
4   return $D$;
end
```

Figure 2.1: A subroutine to remove node-inconsistent values.

The formal definition of arc-consistent can be defined as *"A primitive constraint c is arc consistent with domain D if either $|vars(c)| \neq 2$ or, if $vars(c) = \{x, y\}$, then for each $d_x \in D(x)$, there is some $d_y \in D(y)$ such that $\{x \mapsto d_x, y \mapsto d_y\}$ is a solution of c and for each $d_y \in D(y)$, there is some $d_x \in D(x)$ such that*

```
nc_algorithm($C$, $D$)
begin
    let $C$ be a set of constraints $\{c_1, \ldots, c_n\}$;
    for $i \leftarrow 1$ to $n$ do
        $D \leftarrow$ nc_primitive($c_i$, $D$);
    return $D$;
end
```

Figure 2.2: A node-consistency algorithm.

$\{x \mapsto d_x, y \mapsto d_y\}$ *is a solution of* c*. A CSP with constraint* $c_1 \wedge \cdots \wedge c_n$ *and domain* D *is arc consistent if each primitive constraint* c_i *is arc consistent with* D *for* $1 \leq i \leq n$" [38]. An arc-consistency algorithm [38] shown in Figure 2.4 is a simple algorithm for illustrating the idea. **ac_primitive** shown in Figure 2.3 is the subroutine to remove the arc-inconsistent values from the variable domains. There exists some sophisticated arc-consistency algorithms such as AC-3 [37], AC-4 [40], AC-5 [32], AC-6 [5], and AC-7 [6].

```
ac_primitive($c$, $D$)
begin
    if $|vars(c)| = 2$ then
        let $\{x, y\} = vars(c)$;
        $D(x) \leftarrow \{d_x \in D(x) \mid \exists d_y \in D(y), \{x \mapsto d_x, y \mapsto d_y\}$ is a solution to $c\}$;
        $D(y) \leftarrow \{d_y \in D(y) \mid \exists d_x \in D(x), \{x \mapsto d_x, y \mapsto d_y\}$ is a solution to $c\}$;
    return $D$;
end
```

Figure 2.3: A subroutine to remove arc-inconsistent values.

```
ac_algorithm($C$, $D$)
begin
1   let $C$ be a set of constraints $\{c_1, \dots, c_n\}$;
2   let $D'$ be a domain;
3   repeat
4       $D' \leftarrow D$;
5       for $i \leftarrow 1$ to $n$ do
6           $D \leftarrow$ ac_primitive($c_i$, $D$);
    until $D' = D$;
7   return $D$;
end
```

Figure 2.4: An arc-consistency algorithm.

2.1.2 Backtracking Solver

To determine the satisfaction of a finite domain CSP, we can search through all the combinations of valuations, because the number of combinations is finite. *Backtracking solver* [41, 27] is such an algorithm to find the solution of a given CSP based on tree search.

A backtracking solver [38] as shown in Figure 2.6 is a depth-first traversal of a search tree. When a leaf node of the search tree is traversed, a valuation is generated and the satisfaction of the valuation is tested. *true* is returned if all the constraints can be satisfied by this valuation. Otherwise, another valuation is generated and tested again. Until all the possible combinations are generated and tested, *false* will be returned meaning that no solution to the problem.

A backtracking solver is expensive in terms of the running time. Since it is a complete solver, it has exponential time complexity in the worst case. A common approach to speedup search is to combine local consistency algorithm (arc-consistency algorithm) and backtracking tree search. The local consistency algorithm is used to refine the domain. Whenever a value is chosen from a variable domain during tree search, the local consistency algorithm is invoked. Such an algorithm [38] is given in Figure 2.7. There are many existing systems, such as clp(FD) [17], SICStus Prolog [49], and CHIP [31], provide the efficient finite domain solver. Besides, there is a C++ library for solving CSP called ILOG Solver [35]. All of them facilitate backtracking tree search by local consistency algorithm.

```
satisfiable($C$)
begin
    let $C$ be a set of constraints $\{c_1, \ldots, c_n\}$;
    for $i \leftarrow 1$ to $n$ do
        if $vars(c_i) = \emptyset$ then
            if $c_i$ is unsatisiable then
                return false;
    return true;
end
```

Figure 2.5: A subroutine to test the satisfaction of constraints.

We give an example here to illustrate the idea of combining arc-consistency algorithm and backtracking tree search. For example, given a CSP P, where $V = \{x, y, z\}$, $D(x) = \{1, 2, 3\}$, $D(y) = \{1, 2\}$, $D(z) = \{1\}$, and $C = \{x \leq 2, x \leq y, x + y + z \neq 3\}$. It is possible to construct a complete search tree as shown in Figure 2.8. All the possible valuations $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$ are generated. The solution to P is $\{\theta_2, \theta_4\}$. If an arc-consistency algorithm is invoked before each variable assignment as described in **enhanced_backtracking_solver** (in Figure 2.7), then it is possible to find the solution to P by traversing part of a complete search tree as shown in Figure 2.9. Before each variable assignment to take place, **nc_algorithm** and **ac_algorithm** are invoked (in Figure 2.7) in order to remove inconsistent values. Before variable x is assigned to a value, node-

```
backtracking_solver($C$, $D$)
begin
    # $vars(C)$ is a set of variables occurring in $C$
    if $vars(C) = \emptyset$ then
        return satisfiable($C$);
    else
        choose $x \in vars(C)$;
        for each $d \in D(x)$ do
            let $C'$ be obtained from $C$ by assigning $d$ to $x$;
            if satisfiable($C'$) = $true$ then
                if backtracking_solver($C'$, $D$) = $true$ then
                    return $true$;
        return $false$;
end
```

Figure 2.6: A backtracking solver.

consistency and arc-consistency algorithms are invoked. The value $3 \in D(x)$ will be removed when applying **nc_algorithm** to P, because $\{x \mapsto 3\}$ violates the constraint "$x \leq 2$." Therefore, the branch to node $x \mapsto 3$ and its subtree are pruned as shown in Figure 2.9. Similarly, the left subtree of node $x \mapsto 2$ is pruned, because $\{y \mapsto 1\}$ cannot find any support from variable domain $D(x)$ to satisfy constraint "$x \leq y$" when **ac_algorithm** is invoked.

2.1.3 The Branch-and-Bound Algorithm

We have presented the definition of CSP. Given a CSP, it is possible to determine whether there exists a valuation that satisfies all the constraints or not. The solution to the CSP is then the set of valuations that satisfies all the constraints. It is possible to define the "best" (or optimal) solution to the CSP. Finding the "best" solution to the CSP is called an *optimization problem* [38]. This requires

```
enhanced_backtracking_solver($C$, $D$)
begin
    $D \leftarrow$ nc_algorithm($C$, $D$);
    $D \leftarrow$ ac_algorithm($C$, $D$);
    if $D$ contains an empty variable domain then
        return false;
    else if $D$ contains all singleton variable domain then
        let $\theta$ be the valuation corresponding to $D$;
        if $C\theta$ holds then
            return $D$;
        else
            return false;
    choose $x$ such that $|D(x)| \geq 2$;
    for each $d \in D(x)$ do
        let $D'$ be a domain;
        $D' \leftarrow$ enhanced_backtracking_solver($C \wedge (x = d)$, $D$);
        if $D' \neq$ false then
            return $D'$;
    return false;
end
```

Figure 2.7: An enhanced backtracking solver.

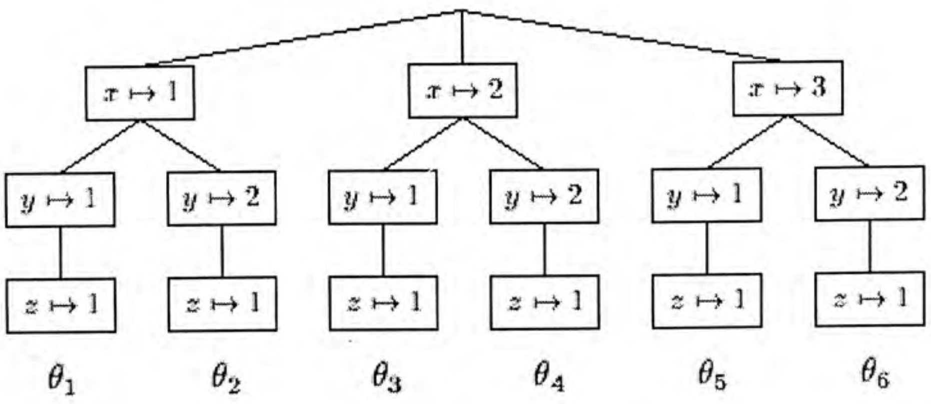


Figure 2.8: A complete search tree.

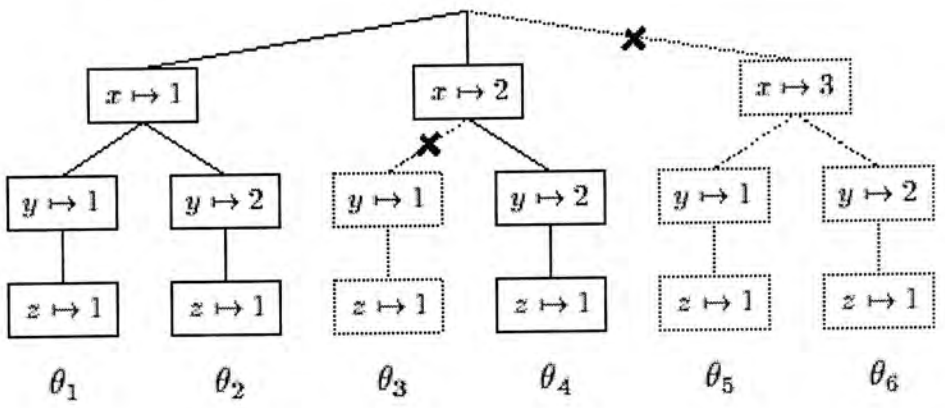


Figure 2.9: A search tree with pruning.

defining some ways to compare valuations in the given CSP. For example, given a CSP, where $V = \{x, y\}$, $D(x) = \{1, 2, 3\}$, $D(y) = \{1, 2, 3\}$, and $C = \{x > y, y \neq 2\}$. It is possible to define the "best" solution such that the value of "$x + y$" should be the minimum. There are two valuations, $\theta_1 = \{x \mapsto 2, y \mapsto 1\}$ and $\theta_2 = \{x \mapsto 3, y \mapsto 1\}$, satisfy all the constraints. Hence, $\{\theta_1, \theta_2\}$ is the solution to the CSP, but the "best" solution should be $\{\theta_1\}$ (the minimum value of "$x + y$" should be "$2 + 1$" instead of "$3 + 1$").

The Branch-and-Bound algorithm [42] was originally proposed for feature subset selection or combinatorial optimization. *"The problem of feature subset selection is to select a subset of (m) features from a larger set of (n) features or measurements to optimize the value of a criterion over all subsets of the size m"* [42]. However, it is also possible to apply the Branch-and-Bound algorithm for constraint optimization problem. The Branch-and-Bound algorithm is based on backtracking tree search. It focuses on finding an optimal solution and it guarantees global optimality under the assumption of monotonicity [42]. To apply the Branch-and-Bound algorithm in constraint optimization problem, it is necessary to define a function f to evaluate a valuation such that the input of f is a valuation and the output of f is a real number over $\leq$. Given a CSP, where $V = \{x_1, \ldots, x_n\}$, for all $d_{x_i} \in D(x_i)$, the function f should be monotonic such that

$$f(\{x_1 \mapsto d_{x_1}\}) \leq f(\{x_1 \mapsto d_{x_1}, x_2 \mapsto d_{x_2}\}) \leq \cdots \leq f(\{x_1 \mapsto d_{x_1}, \ldots, x_n \mapsto d_{x_n}\})$$

Let B be a lower bound which is the current minimum evaluation of a valuation. If $f(\{x_1 \mapsto d_{x_1}, \ldots, x_k \mapsto d_{x_k}\}) \geq B$, where $k < n$, then $f(\{x_1 \mapsto d_{x_1}, \ldots, x_n \mapsto d_{x_n}\}) \geq B$. The bound B will be updated if a leaf node is reached and the evaluation returned by f to the valuation for this leaf node is less than B.

When an evaluation returned by f to a valuation for any node (in the search tree) is larger than the bound B, then each evaluation returned by f to all valuations for nodes that are successors of that node is also larger than B (f is monotonic). Hence, it is not possible to find an optimal solution in the subtree under that node. This comparison is called a *check bound* operation. By applying the Branch-and-Bound algorithm, it is possible to find an optimal solution without exhaustive search.

To illustrate the idea of the Branch-and-Bound algorithm, we use the same example in Section 2.1.2. In addition, the function f is "$x + y$." The bound B is initialized to ∞. At each step of traversing the search tree as shown in Figure 2.10, the function f is applied to a valuation corresponding to the traversed node. The returned evaluation is compared to the bound B. Since $B = \infty$, no pruning occurs before node X is traversed. When node X is traversed, the corresponding valuation θ_1 violates the constraint "$x + y + z \neq 3$." Therefore, θ_1 is not a solution to the problem. The bound B remains unchanged ($B = \infty$). No pruning occurs before node Y is traversed. When node Y is traversed, the corresponding valuation θ_2 satisfies all the constraints. Hence, θ_2 becomes the current "best" solution to the problem. The bound B is updated to 3, because $f(\theta_2) = 3$. No pruning occurs before node Z is traversed. However, when node Z is traversed, the corresponding valuation is $\{x \mapsto 2, y \mapsto 2\}$. The evaluation returned by $f(\{x \mapsto 2, y \mapsto 2\})$ is 4. It is larger than the current bound $B = 3$.

Therefore, the successors of node Z is pruned.

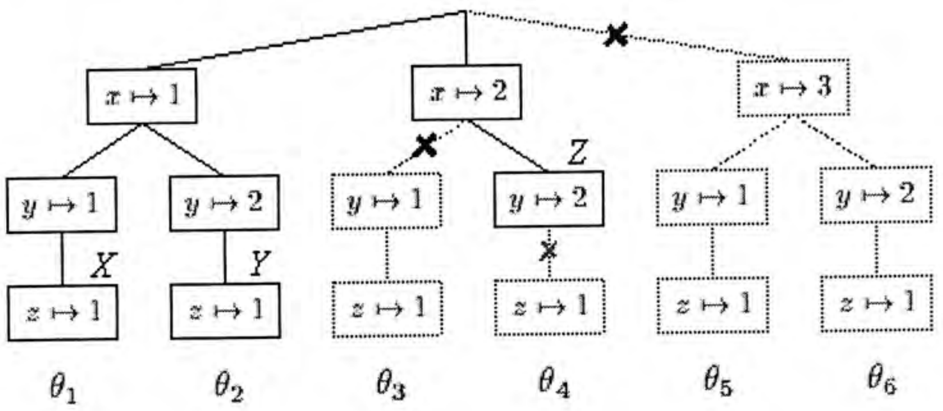


Figure 2.10: A branch-and-bound search tree.

2.2 Over-constrained Problems

Classical CSPs are sometimes inadequate in modeling certain real-life applications [25, 44]. If the real problem to be modeled is over-constrained, then it is impossible to find a solution to satisfy all the constraints if modeled as classical CSP. Such problems are called *over-constrained problems.* Given an over-constrained problem, we can get an answer for the problem if we are willing to "weaken" some of the constraints [25]. Therefore, it is more suitable to use alternative approaches to model over-constrained problems instead of using classical CSP. Some alternatives have been proposed to model over-constrained problems. They are *weighted constraint satisfaction problem* (weighted CSP) [25], *possibilistic constraint satisfaction problem* (possibilistic CSP) [47], *fuzzy constraint satisfaction problem* (FCSP) [44], *partial constraint satisfaction problem* (PCSP) [25], and *constraint hierarchy* (CH) [13]. Since we mainly work on CH in the research, we will present CH in more detail in Section 2.3. We also present two meta-frameworks: *semiring-based constraint satisfaction problem* (SCSP) [9] and *valued constraint satisfaction problem* (VCSP) [48] that generalize classical CSP and over-constrained CSPs into a generic framework.

2.2.1 Weighted Constraint Satisfaction Problems

Weighted CSP is a framework in which the constraints are not equally important. *"Preferences can be reflected in the branch and bound metric by assigning weights to constraints"* [25]. In this framework, a *weighted constraint* c is a constraint associated with a weight w_c, which is an element of a totally ordered set W such that each element $w_c \in W$ is ordered by a relation $\leq$. If the weight of another weighted constraint c' is $w_{c'}$ such that $w_c < w_{c'}$, then c' is more "important" than c. A weighted CSP is a tuple $\langle V, D, C_w \rangle$, where V is a set of variables, D is a set of variable domains, and C_w is a set of weighted constraints. The satisfaction degree of a valuation to a weighted CSP is given by the sum of weights of all violated constraints. A solution to a weighted CSP is defined as the set of valuations such that their satisfaction degrees are the minimum. *Maximal constraint satisfaction* (MAX CSP) [25] is a special instance of weighted CSP, in which all the weights of the weighted constraints are equal to one. We seek a solution that satisfies as many constraints as possible.

2.2.2 Possibilistic Constraint Satisfaction Problems

In possibilistic CSP, a *possibility distribution* π over labelings (or valuations) is used to represent the preferences among labelings [47]. Two measures over constraints, *possibility measure* Π_π and *necessity measure* N_π, are defined in terms of possibility distribution π. The possibility (or necessity) measure represents the bound on possibility (or necessity) measure of constraints. If the necessity bound on a constraint c is less than the necessity bound on another constraint c' ($N_\pi(c) < N_\pi(c')$), then the satisfaction of c' is preferred to the satisfaction of c. By applying the measures, it is possible to define a set of "most possible" valuations satisfying the bounds. Possibilistic CSP is a tuple $\langle V, D, C_{nv} \rangle$, where V is a set of variables, D is a set of variable domains, and C_{nv} is a set of *necessity-valued* *constraints.* Each necessity-valued constraint is a pair (c, α), where c is a classical constraint and $\alpha \in [0, 1]$. The necessity-valued constraint (c, α) represents that the necessity bound of c is at least α. Therefore, a necessity-valued constraint $(c, 1)$ represents that c must be absolutely satisfied and a necessity-valued constraint $(c, 0)$ represents that c is always satisfied. A necessity-valued constraint (c, α) is satisfied by a possibility distribution π if and only if N_π induced by π such that $N_\pi(c) \geq \alpha$. The possibilistic CSP is satisfied by a possibility distribution π if and only if N_π induced by π verifies $\forall(c, \alpha) \in C_{nc}, N_\pi(c) \geq \alpha$. *"Thus a possibilistic CSP has not a set of consistent labelings, but a set of possibility distributions on the set of all labelings"* [47]. Solution to possibilistic CSP does not require an optimal satisfaction of all the necessity-valued constraints, but it requires finding a possibility distribution in order to satisfy all the necessity-valued constraints.

2.2.3 Fuzzy Constraint Satisfaction Problems

"In the case of a fuzzy constraint, different tuples satisfy the given constraint to a different degree" [44]. When a valuation, denoted by $\underline{v}$, applies to a *fuzzy constraint* c, a satisfaction degree (from an interval [0, 1]) is returned, denoted by $c(\underline{v}) \in [0, 1]$. We say that a valuation *fully satisfies* a fuzzy constraint if the satisfaction degree is 1. A valuation *fully violates* a fuzzy constraint if the satisfaction degree is 0. A FCSP is a tuple $\langle V, D, C_f \rangle$, where V is a set of variables, D is a set of variable domains, and C_f is a set of fuzzy constraints. The solution to a FCSP is defined by a *degree of joint satisfaction* in terms of the satisfaction degree of each individual fuzzy constraint. The degree of joint satisfaction indicates the goodness of a valuation in satisfying the fuzzy constraints globally. There are three different ways in defining the degree of joint satisfaction. The first definition is based on the *conjunctive combination principle.* Given a list of fuzzy constraints $(c_1, \ldots, c_N)$ and a valuation $\underline{v}$, C_{min} is

a degree of joint satisfaction indicating the minimum of the satisfaction of each individual fuzzy constraint as

$$C_{min}((c_1, \ldots, c_N), \underline{v}) = min\{c_i(\underline{v}_i) \mid i \in \{1, \ldots, N\}\}.$$

The second definition is based on the *productive combination principle.* Given a list of fuzzy constraints $(c_1, \ldots, c_N)$ and a valuation $\underline{v}$, C_{pro} is a degree of joint satisfaction indicating the product of the satisfaction of each individual fuzzy constraint as

$$C_{pro}((c_1, \ldots, c_N), \underline{v}) = \prod_{i=1}^{N} c_i(\underline{v}_i).$$

The third definition is based on the *average combination principle.* Given a list of fuzzy constraints $(c_1, \ldots, c_N)$ and a valuation $\underline{v}$, C_{ave} is a degree of joint satisfaction indicating the average of the satisfaction of each individual fuzzy constraint as

$$C_{ave}((c_1, \ldots, c_N), \underline{v}) = \frac{1}{N} \sum_{i=1}^{N} c_i(\underline{v}_i).$$

Based on the three types of the degree of joint satisfaction: C_{min}, C_{pro}, and C_{ave}, the *best solution* to a FCSP can be defined as if the degree of joint satisfaction of all the fuzzy constraints is the maximal possible. By applying different types of the degree of joint satisfaction (C_{min}, C_{pro}, and C_{ave}), we can obtain different types of best solution.

2.2.4 Partial Constraint Satisfaction Problems

"Partial constraint satisfaction involves finding values for a subset of the variables that satisfy a subset of the constraints" [25]. Therefore, it is possible to "weaken"

some of the constraints to permit additional acceptable valuations. A *problem space* is a partially ordered set $(PS, \leq)$. PS is a set of classical CSPs with a partial order $\leq$ over PS. Given two classical CSPs P and P', $P \leq P'$ holds if and only if the solutions to P' is a subset of the solutions to P. In addition, if the solutions to P' is a subset of the solutions to P and the two solution sets are not equal, then we say that P is "weaker" than P', denoted by $P < P'$. There are four operations to weaken a classical CSP: enlarging a variable domain, enlarging the domain of constraint, removing a variable, and removing a constraint. Weakening constraints mean creating a different problem. The solution to a PCSP should be close to the original in the sense of having a solution set similar to the original. PCSP can be defined in a more formal way. A PCSP is a tuple $\langle P, (PS, \leq), M, (N, S)\rangle$, where P is an initial over-constrained CSP, $(PS, \leq)$ is pair of the problem space and the partial order over PS respectively, M is a metric on PS (a distance function over PS), and (N, S) is a pair of the *necessary solution distance* and the *sufficient solution distance* respectively. Different metrics are possible, but the obvious one is derived from the problems space. One possible metric M on two problems P and P' is that $M(P, P')$ returns the number of solutions not shared by P and P'. When $P' \leq P$ then M measures how many solutions have been added by weakening P to P'. A solution to a PCSP is defined as the solution to a weaken problem P' (P' is the relaxation of initial over-constrained problem P from the problems space) where $M(P, P') < N$. A solution is *sufficient* if $M(P, P') \leq S$. When the distance between P and P' is the minimal over PS, then the solution is an *optimal solution.*

2.2.5 Semiring-Based Constraint Satisfaction Problems

SCSP is a general framework for constraint satisfaction and optimization [9]. This framework is based on a *semiring structure.* It is possible to define different semirings in order to provide different instances of SCSP. Some of the previous discussed frameworks, such as classical CSP, weighted CSP, and FCSP, are possible instances. In SCSP, a set of *semiring values* specifies the values to be associated with each tuple of values of the variable domain. In addition, there are two *semiring operations*, *additive operation* $+$ and *multiplicative operation* $\times$, to model *constraint projection* and *constraint combination* respectively. A *c-semiring* (it is a semiring and "c" stands for "constraint") is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$. A is a set of elements (or semiring values) such that $\mathbf{0}, \mathbf{1} \in A$. $\mathbf{0}$ is the minimum element in A meaning that it is the "worst" element in A. $\mathbf{1}$ is the maximum element in A meaning that it is the "best" element in A.

An additive operation $+$ is an operation over A with the following properties:

- $+$ is a closed operation over $A \Leftrightarrow \forall a, b \in A \rightarrow a + b \in A$
- $+$ is a commutative operation over $A \Leftrightarrow \forall a, b \in A, a + b = b + a$
- $+$ is an associative operation over $A \Leftrightarrow \forall a, b \in A, a + (b + c) = (a + b) + c$
- $+$ is an idempotent operation over $A \Leftrightarrow \forall a \in A, a + a = a$
- $\mathbf{0}$ is an unit element$\Leftrightarrow \forall a \in A, a + \mathbf{0} = a = \mathbf{0} + a$
- $\mathbf{1}$ is an absorbing element $\Leftrightarrow \forall a \in A, a + \mathbf{1} = \mathbf{1} = \mathbf{1} + a$

In the original formulation, a partial ordering over A, denoted by $\leq_S$, is defined by the additive operation. Such a partial ordering is defined as $\forall a, b \in A, a \leq_S b \Leftrightarrow a + b = b$. In other words, $a \leq_S b$ means b is "better" than a. This partial ordering is used for determining the "best" solution in this framework.

A multiplicative operation $\times$ is an operation over A with the following properties:

- $\times$ is a closed operation over $A \Leftrightarrow \forall a, b \in A \rightarrow a \times b \in A$

- $\times$ is a commutative operation over $A \Leftrightarrow \forall a, b \in A, a \times b = b \times a$
- $\times$ is an associative operation over $A \Leftrightarrow \forall a, b \in A, a \times (b \times c) = (a \times b) \times c$
- $\times$ is an intensive operation over $A \Leftrightarrow \forall a \in A, a \times b \leq_S a$
- $\mathbf{1}$ is an unit element $\Leftrightarrow \forall a \in A, a \times \mathbf{1} = a = \mathbf{1} \times a$
- $\mathbf{0}$ is an absorbing element $\Leftrightarrow \forall a \in A, a \times \mathbf{0} = \mathbf{0} = \mathbf{0} \times a$
- $\times$ distributes over $+ \Leftrightarrow \forall a, b, c \in A, a \times (b + c) = (a \times b) + (a \times c)$

A *constraint system* is a tuple $CS = \langle S, D, V \rangle$, where S is a c-semiring, D is a set of variable domains, and V is a set of variables. A *constraint* over CS is a tuple $\langle def, con \rangle$, where con is called the *type* of the constraint such that $con \subseteq V$, and def is called the *value* of the constraint and it is a mapping such that $def : D^k \to A$, where k is the cardinality of con. A *constraint problem* P over CS is a tuple $\langle C, con' \rangle$, where C is a set of constraints over CS and con' is called the type of the problem such that $con' \subseteq V$. $\times$ is used to combine the semiring values of the tuples of each constraint in order to get the semiring value of a tuple for all variables. $+$ is used to obtain a semiring value of the tuples of a constraint in the type of con' (the type of the problem).

Two operations, *combination* $\otimes$ and *projection* $\Downarrow$ over constraints, are used to model constraint combination and constraint projection respectively. Given a constraint system $CS = \langle S, D, V \rangle$, where V is totally ordered via an ordering $\prec$. Consider any k-tuple $t = \langle t_1, \ldots, t_k \rangle$ of values of D, two sets of variables $W' = \{w'_1, \ldots, w'_m\}$ and $W = \{w_1, \ldots, w_k\}$ such that $W' \subseteq W \subseteq V$, where $w_i \prec w_j$ and $w'_i \prec w'_j$ if $i \leq j$. Then the projection of t from W to W', denoted by $t \downarrow^{W}_{W'}$, is defined as a m-tuple $t' = \langle t'_1, \ldots, t'_m \rangle$ with $t'_i = t_j$ if $w'_i = w_j$. Then, combination is defined as follows. Given a constraint system $CS = \langle S, D, V \rangle$, where $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, and two constraints $c_1 = \langle def_1, con_1 \rangle$ and $c_2 = \langle def_2, con_2 \rangle$ over CS, their combination, denoted by $c_1 \otimes c_2$, is the constraint $c = \langle def, con \rangle$

with $con = con_1 \cup con_2$, and $def(t) = def_1(t \downarrow^{con}_{con_1}) \times def_2(t \downarrow^{con}_{con_2})$. Since $\times$ is commutative and associative, $\otimes$ is also commutative and associative. Hence, we can extend it to more than two arguments. For example, if C is a set of constraints where $C = \{c_1, \ldots, c_n\}$, we can combine all the constraints by $\otimes$, written as $c_1 \otimes \cdots \otimes c_n$. This can be denoted by a notation $\bigotimes C$. On the other hand, projection is defined as follows. Given a constraint system $CS = \langle S, D, V \rangle$, a constraint $c = \langle def, con \rangle$ over CS, and a set of variables W such that $W \subseteq V$, the projection of c over W, denoted by $c \Downarrow_W$, is the constraint $\langle def', con' \rangle$ over CS, where $con' = W \cap con$ and $def'(t') = \sum_{\{t | t \downarrow^{con}_{W \cap con} = t'\}} def(t)$.

It is possible to define the solution to a SCSP by combination and projection. In order to find the solution to SCSP, the first step is to combine all the constraints into a single constraint. Then, the combined constraint is projected over the variables in the type of the problem. Since def can be used to determine the satisfaction degree of a constraint over con, we can compare the semiring value of each tuple of the combined constraint, by partial ordering $\leq_S$, in order to get the best solution to the problem. Formally, the solution to SCSP is defined as follows. Given a SCSP problem $P = \langle C, con' \rangle$ over a constraint system CS, the solution to P is a constraint defined as $Sol(P) = (\bigotimes C) \Downarrow_{con'}$.

Given an SCSP $P = \langle C, con' \rangle$. Assume there is only one unary constraint c_x for each variable $x \in con'$. Also, assume there is only one binary constraint c_{xy} for variables $x, y \in con'$. P is *semiring-based arc-consistent* (SAC) [10] if $\forall x \in con' \wedge \forall y \in con' - \{x\} \wedge c_x = \{c_x \otimes c_{xy} \otimes c_y\} \Downarrow_x$. The intuitive meaning is that the semiring values of constraint c_x is given by the semiring values of constraints c_x, c_{xy}, and c_y.

2.2.6 Valued Constraint Satisfaction Problems

VCSP is a general framework to encompass most existing CSP extensions [48]. The frameworks such as classical CSP, weighted CSP, and possibilistic CSP can be casted to VCSP. Each constraint in this framework is annotated with an element called *valuation* to represent the degree of its violation. The meaning of valuation in VCSP is different from our previous definition (we define a valuation to be an assignment of values from variable domains to corresponding variables). Hence, we will simply use *assignment* to represent an assignment of values from variable domains to corresponding variables. Based on an algebraic framework called *valuation structure* [48], it is possible to define VCSP. A valuation structure is a tuple $\langle E, \circledast, \succ \rangle$, where E is a set such that the elements in the set are called valuations. Valuations are totally ordered by $\succ$ with a maximum valuation, denoted by $\top$, and a minimum valuation, denoted by $\bot$. $\circledast$ is a commutative, associative, and closed binary operation on E. A VCSP $\mathcal{P}$ is a tuple $\langle X, D, C, S, \varphi \rangle$, where X is a set of variables, D is a set of variable domains, C is a set of constraints, $S = \langle E, \circledast, \succ \rangle$ is a valuation structure, and φ is a mapping such that $\varphi : C \to E$ and $\varphi(c)$ is called the valuation of constraint c. An assignment A of values from variable domains to corresponding variables $Y \subset X$ is evaluated by combining the valuations of all the violated constraints using $\circledast$. Given a VCSP $\mathcal{P} = \langle X, D, C, S, \varphi \rangle$ and an assignment A of the variables of $Y \subset X$, the valuation of an assignment is denoted by $\mathcal{V}_{\mathcal{P}}(A)$ such that $\mathcal{V}_{\mathcal{P}}(A) = \circledast\{\varphi(c) \mid c \in C \wedge vars(c) \subset Y \wedge A \text{ violates } c\}$. The solution to VCSP is an assignment A with minimum valuation.

2.3 The Theory of Constraint Hierarchies

We have presented some existing frameworks to model over-constrained constraint satisfaction problem. In this section, we present a framework called *constraint hierarchy* (CH) [13] which is also used to model over-constrained constraint satisfaction problem. Let $\mathcal{D}$ be a constraint domain. A variable x is an unknown that has an associated variable domain $D(x) \subseteq \mathcal{D}$, which defines the set of possible values for x. An n-ary constraint c is a relation over $\mathcal{D}^n$. 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 is associated with a weight w. 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.

We use an example in Figure 2.11 to explain CHs in more detail. There are three levels in the constraint hierarchy H. There is no required constraint in the required level H_0. However, there are two *strong* constraints c_1^1 and c_2^1 in H_1 and three *weak* constraints c_1^2, c_2^2, and c_3^2 in H_2.

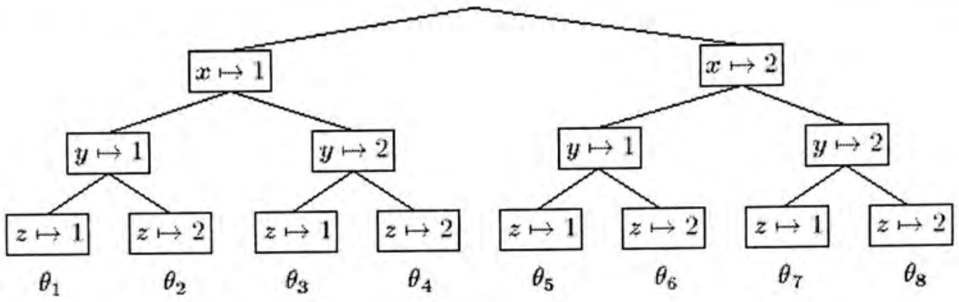
Variable	V	x, y, z
Domain	D	$D(x) = \{1,2\}, D(y) = \{1,2\}, D(z) = \{1,2\}$
Hierarchy	H	$H_0 = \emptyset$, $H_1 = \{c_1^1 : z = 1(w_1^1 = 0.3), c_2^1 : y + z = 3(w_2^1 = 0.5)\}$, $H_2 = \{c_1^2 : x + y \neq 2(w_1^2 = 0.8), c_2^2 : y = 2(w_2^2 = 0.5)$, $c_3^2 : x + y + z < 5(w_3^2 = 0.6)\}$

Figure 2.11: An example of constraint hierarchy.

A *valuation* $\theta = \{v_1 \mapsto d_1, \ldots, v_n \mapsto d_n\}$ for a set of variables $\{v_1, \ldots, v_n\}$ means that each v_i is assigned the value d_i where $d_i \in D(v_i)$. 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 the value of $c\theta$ is *true*. An *error function* $e(c\theta)$ measures how well a constraint c is satisfied by valuation θ. In classical CSP, an error function is also useful for local search, such as the Tabu Search algorithm [26] and the Adaptive Search algorithm [18]. 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 such that $e(c\theta) = 0$ if $c\theta$ holds; otherwise, $e(c\theta) = 1$. A *metric error function* is an error function such that $e(c\theta) = 0$ if $c\theta$ holds; otherwise $e(c\theta) > 0$. The value $e(c\theta)$ returned by an error function is an *error value*, indicating how nearly a constraint c is satisfied by a valuation θ. 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 θ_1, θ_2, θ_3, θ_4, θ_5, θ_6, θ_7, and θ_8. Figure 2.12 gives the error values of all valuations in the complete search tree using the trivial error function. The error values of valuation θ_1 can be grouped into a tuple $\langle\langle\rangle, \langle e(c_1^1\theta_1), e(c_2^1\theta_1)\rangle, \langle e(c_1^2\theta_1), e(c_2^2\theta_1), e(c_3^2\theta_1)\rangle\rangle$. Since, for example, θ_1 satisfies c_1^1 but violates c_1^2, $e(c_1^1\theta_1) = 0$ and $e(c_1^2\theta_1) = 1$ respectively. We can obtain the error values of other valuations similarly.

A solution set S to a CH is a set of valuations. Each valuation in S must satisfy all the required constraints in H_0. In addition, each valuation should satisfy the non-required constraints as much as possible with respect to their strengths. To formalize this, we need to obtain a valuation set S_0 in which each valuation satisfies all the constraints in H_0. Then, we compare every valuation in S_0 and eliminate all potential valuations that are worse than some other potential valuations using comparator predicate *better* as following.



Valuation	Trival error values	Valuation	Trival error values
θ_1	$\langle\langle\rangle, \langle 0,1\rangle, \langle 1,1,0\rangle\rangle$	θ_5	$\langle\langle\rangle, \langle 0,1\rangle, \langle 0,1,0\rangle\rangle$
θ_2	$\langle\langle\rangle, \langle 1,0\rangle, \langle 1,1,0\rangle\rangle$	θ_6	$\langle\langle\rangle, \langle 1,0\rangle, \langle 0,1,1\rangle\rangle$
θ_3	$\langle\langle\rangle, \langle 0,0\rangle, \langle 0,0,0\rangle\rangle$	θ_7	$\langle\langle\rangle, \langle 0,0\rangle, \langle 0,0,1\rangle\rangle$
θ_4	$\langle\langle\rangle, \langle 1,1\rangle, \langle 0,0,1\rangle\rangle$	θ_8	$\langle\langle\rangle, \langle 1,1\rangle, \langle 0,0,1\rangle\rangle$

Figure 2.12: Valuations and error values.

$$S_0 = \{\theta \mid \forall c \in H_0 \; c\theta \text{ holds}\}, \text{ and}$$
$$S = \{\theta \mid \theta \in S_0 \wedge \forall \sigma \in S_0 \; \neg better(\sigma, \theta, H)\}.$$

In the original formulation of CH, two kinds of comparator are defined. The first comparator is *locally-better* (*l-b*), each constraint is considered in H individually. We can determine whether a valuation θ is *locally-better* than another valuation σ. θ is *locally-better* than σ if the error after applying θ is equal to the error after applying σ for each constraint through some level $k-1$, and there exists at least one constraint the error after applying θ is strictly less than and less than or equal for the rest. To formalize this, *locally-better* can be defined as follows.

$$\begin{aligned} locally\text{-}better(\theta, \sigma, H) \equiv \; & \exists k > 0 \text{ such that} \\ & \forall i \in \{1, \ldots, k-1\}, \forall p \in H_i, e(p\theta) = e(p\sigma) \\ & \wedge \exists q \in H_k, e(q\theta) < e(q\sigma) \\ & \wedge \forall r \in H_k, e(r\theta) \leq e(r\sigma). \end{aligned}$$

The rest of the comparators are *global comparators*. A schema called *globally-better* (*g-b*) can be defined for global comparators which are parameterized by a function g that combines the errors of all the constraints H_i at level i. The intuitive meaning of *globally-better* is to compare valuations by level. A valuation θ is *globally-better* than a valuation σ if the combined errors of the constraints after applying θ is the same as the combined errors of the constraints after applying σ for each level through some level $k-1$, and it is strictly less at level k. It is possible to define *globally-better* in a more formal way as follows.

$$\begin{aligned} \textit{globally-better}(\theta, \sigma, H, g) \equiv \exists k > 0 \text{ such that} \\ \forall i \in \{1, \ldots, k-1\}, g(\theta, H_i) = g(\sigma, H_i) \\ \wedge g(\theta, H_k) < g(\sigma, H_k). \end{aligned}$$

Three global comparators can be defined using *globally-better* and different combining functions g: *weighted-sum-better* (*w-s-b*), *worst-case-better* (*w-c-b*), and *least-squares-better* (*l-s-b*). The weight for constraint p is denoted by w_p where the weight is a positive real number.

$$\textit{weighted-sum-better}(\theta, \sigma, H) \equiv \textit{globally-better}(\theta, \sigma, H, g),$$
where $g(\tau, H_i) \equiv \sum_{p \in H_i} w_p e(p\tau)$,

$$\textit{worst-case-better}(\theta, \sigma, H) \equiv \textit{globally-better}(\theta, \sigma, H, g),$$
where $g(\tau, H_i) \equiv \max\{w_p e(p\tau) \mid p \in H_i\}$, and

$$\textit{least-squares-better}(\theta, \sigma, H) \equiv \textit{globally-better}(\theta, \sigma, H, g),$$
where $g(\tau, H_i) \equiv \sum_{p \in H_i} w_p e(p\tau)^2$.

2.4 Related Work

Many efficient algorithms designed for real domain CHs have been proposed [22, 46, 34, 11, 33, 12, 2]. Since real domain CH solvers have been successfully applied to problems in computer graphics such as geometric design and user-interface construction [19, 45]. *Local propagation* [50] is a widely used technique to solve real domain constraint hierarchies [34, 33]. *"Local Propagation is an efficient constraint satisfaction algorithm that takes advantage of potential locality of constraint systems. It is often used in graphical user-interfaces (GUIs) to solve constrain systems that describe structures and layouts of figures"* [34]. It is a linear process to detect the values of variables by determining the constraints [50]. For example, given two constraints "$y = 1$" and "$x = y + 2$," it is possible to detect the value of variable y by determining the constraint "$y = 1$." The variable y can be eliminated and replaced by 1 in the constraint "$x = y + 2$." Hence, we can repeat the process to determine the value of variable x by determining the constraint "$x = (1) + 2$." Finally, it is easy to detect that the value of variable x is 3. Therefore, a core step in local propagation is to determine the fixed value of a variable by a constraint. Variable y has a fixed value of 1 when determining the constraint "$y = 1$" and variable x also has a fixed value of 3 when determining the constraint "$x = y+2$." Many real domain CH solvers, such as DeltaBlue [22], SkyBlue [46], DETAIL [34], Indigo [11], Generalized Local Propagation [33], and Ultraviolet [12], apply local propagation.

An existing technique, the Simplex algorithm [43], for solving real domain CSPs is also applicable for solving real domain CHs, in particular for graphical user interface (GUI) applications. The Simplex algorithm is applied for solving optimization problem in classical CSPs. An *objective function* is used to guide the search for the global optimal solution. However, the original Simplex algorithm cannot be applied for handling GUI applications directly for two main reasons. First, the Simplex algorithm cannot solve similar problems efficiently, such as moving an object with a mouse, adding a constraint, or removing a constraint. Second, it requires all variables to be non-negative that is not the case in GUI applications. The Cassowary and QOCA algorithms [14] adapt the

Simplex algorithm for solving real domain CHs by introducing special objective functions for different comparators. A pair of non-negative variables, δ^+ and δ^-, is introduced to indicate the deviation of a desired value from a variable for the Cassowary algorithm. For *locally-better* and *weighted-sum-better*, an objective function is formed by adding all these pairs of non-negative variables. For *least-squares-better*, an objective function is formed by adding the square of the errors of each labeled constraint. Therefore, the QOCA algorithm is designed for solving the convex quadratic programming problem. In the following, we focus on the techniques in solving finite domain CHs.

2.4.1 An Incremental Hierarchical Constraint Solver

An incremental hierarchical constraint solver (IHCS) is *"an incremental method to solve hierarchies of constraints over finite domains, which borrows techniques developed in intelligent backtracking, and finds locally-predicate-better solutions [locally-better using trivial error function]"* [39]. Given a hierarchy H, a *configuration* Φ of H is a triple $\langle AS, RS, US\rangle$, where AS is an *active store* which is a set of consistent constraints, RS is a *relaxed store* which is a set of relaxed constraints, and US is an *unexplored store* which is a set of constraints "queuing" for activation. In IHCS a *final configuration* is a configuration Φ of a given hierarchy H such that 1) the active store AS is consistent denoted by $AS \not\vdash_X \perp$(X designates a network consistency algorithm such as arc-consistency algorithm), 2) $\forall c \in RS, AS \cup \{c\} \vdash_X \perp$, and 3) $US = \emptyset$. A *locally-predicate-better* comparator for configurations is defined such that $\Phi = \langle AS, RS, US\rangle$ is *locally-predicate-better* than $\Phi' = \langle AS', RS', US'\rangle$ if and only if there exists some level $k > 0$, $\forall i < k$, $|(AS_i \cup US_i)| = |(AS'_i \cup US'_i)|$ and $|(AS_k \cup US_k)| > |(AS'_k \cup US'_k)|$. By using this comparator for configurations, it is possible to define the *best configuration* of a given hierarchy H. The best configuration is a final configuration Φ if there is no other final configuration Φ' which is *locally-predicate-better* then Φ.

Applying IHCS, we can transform a given hierarchy H (corresponding to a CH) into a set of best configurations (corresponding to a set of classical CSPs). Then, it is possible to transform a CH P into a set of classical CSPs $\{P_1, \dots, P_n\}$ such that S is the solution to P and S_i is the solution to P_i and $S \equiv S_1 \cup \dots \cup S_n$. For example, a given CH P has two non-required levels, $H_0 = \{c_1, c_2\}$, $H_1 = \{c_3, c_4\}$, and $H_2 = \{c_5\}$. The solution to P is $S = \{\theta_1, \theta_2\}$. Valuation θ_1 satisfies c_1, c_2, c_3, and c_5, another valuation θ_3 satisfies c_1, c_2, c_4, and c_5. If we can transform the hierarchy H to $C_1 = \{c_1, c_2, c_3, c_5\}$ and $C_2 = \{c_1, c_2, c_4, c_5\}$ (best configurations), then we can find θ_1 and θ_2 simply by solving C_1 and C_2 respectively.

2.4.2 Transforming Constraint Hierarchies into Ordinary Constraint System

Lua [36] 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, $\langle E_{C_1}, \dots, E_{C_n} \rangle$ 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.

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

$$b(E_C, E'_C, i) = \textit{false}, \text{ if } i > n,$$
$$b(E_C, E'_C, i) = E_{C_i} \leq E'_{C_i} \wedge (E_{C_i} = E'_{C_i} \rightarrow b(E_C, E'_C, i+1)), \text{ if } i > n.$$

However, it is unclear how the *locally-better* comparator can be implemented using this approach.

2.4.3 The SCSP Framework

Bistarelli *et. al.* [8] shows how a c-semiring can be constructed to model all instances of *globally-better*. In other words, this approach exploits the fact that CH is an instance of the SCSP framework [9]. Let ε denotes the largest possible error value returned by error function e for any labeled constraint c in hierarchy H and valuation θ, where $\varepsilon \in E, E = \{0\} \cup \mathbb{R}^+ \cup \{\infty\}$. The error combining function is defined as $g_{\phi,\psi}(\theta, H_i) = \phi(\{w_c(\psi(e(c\theta))) \mid c \in H_i\})$, where ϕ is a mapping such that $\phi : \mathcal{P}(E) \rightarrow E$ and ψ is a mapping such that $\psi : E \rightarrow E$. The error combining function for *weighted-sum-better*, *least-squares-better*, and *worst-case-better* can be defined as $g_{\Sigma,\lambda x.x}$, $g_{\Sigma,\lambda x.x^2}$, and $g_{max,\lambda x.x}$ respectively.

$S_{g-b} = \langle E^n, +_{g-b}, \times_{g-b}, \mathbf{0}_{g-b}, \mathbf{1}_{g-b} \rangle$ is the semiring for global comparators, where E^n is the set A such that n is the number of non-required levels, $+_{g-b}$ is the additive operation, $\times_{g-b}$ is the multiplicative operation, $\mathbf{0}_{g-b}$ is the "worst" semiring value, and $\mathbf{1}_{g-b}$ is the "best" semiring value. Suppose $c_{i,j}$, denoting the j constraint in level H_i, is a k-ary labeled constraint such that $var(c_{i,j}) = \{x_1, \dots, x_k\}$. A tuple of values $t = \langle a_1, \dots, a_k \rangle$ associated with constraint $c_{i,j}$ can construct a valuation $\theta = \{x_1 \mapsto a_1, \dots, x_k \mapsto a_k\}$. If r is the error value returned by $e(c_{i,j}\theta)$, then the semiring value associated with tuple t is defined as:

$$\langle \underbrace{0, \dots, 0}_{i-1}, w\psi(r), \underbrace{0, \dots, 0}_{n-i} \rangle, \text{ where } w \text{ is the weight associated to constraint } c_{i,j}.$$

Given two semiring values (tuples of k real numbers) $\vec{a} = \langle a_1, \dots, a_k \rangle$ and $\vec{b} = \langle b_1, \dots, b_k \rangle$. The additive operation $+_{g-b}$ is defined as follows:

$$\vec{a} +_{g-b} \vec{b} = \begin{cases} \langle a_1 \mid \langle a_2, \dots, a_k \rangle +_{g-b} \langle b_2, \dots, b_k \rangle \rangle & \text{if } a_1 = b_1 \\ \langle a_1, \dots, a_k \rangle & \text{if } a_1 < b_1 \\ \langle b_1, \dots, b_k \rangle & \text{if } a_1 > b_1. \end{cases}$$

The multiplication operation $\times_{g-b}$ is defined as $\vec{a} \times_{g-b} \vec{b} = \langle \phi(\{a_1, b_1\}), \dots, \phi(\{a_k, b_k\}) \rangle$. In addition, $\mathbf{0}_{g-b}$ is defined as $\langle \varepsilon, \dots, \varepsilon \rangle$ and $\mathbf{1}_{g-b}$ is defined as $\langle 0, \dots, 0 \rangle$ assuming that $\psi(0) = 0$ and $\psi(\varepsilon) = \varepsilon$.

The same construction fails for the *locally-better* comparator since $\times$ does not distribute over $+$. Only the $\times_{g-b}$ operator (modeled using max) of the *worst-case-better* is idempotent, so that it can enjoy semiring-based arc-consistency techniques (soft constraint local consistency technique) [10] supported in clp(FD, S) [29], while the other global comparators ($\times_{g-b}$ modeled using $\sum$) have to rely on dynamic programming. The clp(FD,S) solver, however, limits the size of the semiring to only 32 elements [29], making it difficult to model any reasonably sized finite domain CH problems.

2.4.4 The DeltaStar Algorithm

DeltaStar [23] is built upon a flat constraint solver which performs the actual constraint solving task. There is a key routine *filter* which takes a set of valuations and a set of constraints as input and it will return a subset of valuations that minimize the error over the input constraints. While most of the finite domain CH solvers are designed for specific (classes of) comparators, DeltaStar is a generic finite domain CH solver which can find solution for arbitrary comparators in theory. The original definition of the solution to a CH is proved difficult to translate into efficient implementations [23]. The major problem is that it is required to compare all the valuations across each level in the hierarchy. Therefore, a recursive definition for solution is defined in this approach.

The new definition for *globally-better* solution is defined as follows.

$$
\begin{aligned}
&S_0 = \{\theta \mid \forall c \in H_0, e(c\theta) = 0\},\\
&S_i = \{\theta \mid \theta \in S_{i-1}, \forall \sigma \in S_{i-1}, \neg(g(e(\sigma, H_i)) < g(e(\theta, H_i)))\}, \text{ and}\\
&S_R = S_n.
\end{aligned}
$$

The intuitive meaning of the definition is that S_R is defined using S_i, where S_i is the set of "best" valuations to satisfy the constraints through level i. For each level $i-1$, only the best valuations in S_{i-1} will be passed to the next level i, and finally S_n is the set of best valuations that satisfy the constraints in H_n. Therefore, S_R is simply the solution set to the CH.

The new definition for *locally-better* solution is defined as follows.

$$
\begin{aligned}
&\mathcal{P}(\Theta, H_i) = \{\mathcal{Q}_1, \ldots, \mathcal{Q}_m\},\\
&\text{where } \vec{\theta} = g(e(\theta, H_i)),\ \vec{\sigma} = g(e(\sigma, H_i)),\\
&\qquad \forall \theta, \sigma \in \Theta, (\theta, \sigma \in \mathcal{Q}_j \Leftrightarrow \vec{\theta} = \vec{\sigma})\\
&\qquad \wedge \forall \theta \in \Theta, (\theta \in \mathcal{Q}_j \Leftrightarrow \nexists \sigma \in \Theta, \vec{\sigma} < \vec{\theta}).
\end{aligned}
$$

$T_0 = S_0$,

$T_i = \bigcup_{\Theta \in T_{i-1}} \mathcal{P}(\Theta, H_i)$, and

$T_R = \bigcup_{V \in T_n} V$.

The new definition for *locally-better* solution is more complicated, because *locally-better* comparator use a partial order instead of total order (for global comparators). Similar to S_R, T_R is defined using T_i which consists of the "best" valuations that satisfy constraints through level i. However, a function $\mathcal{P}$ is introduced to partition the set of incomparable valuations in T_R. Therefore, the final solution set is an union of the sets of solutions at level n. The S_R and T_R constructions can be converted into the algorithms as shown in Figure 2.13 and Figure 2.14 respectively.

```
deltastar_s(solver, $H$)
begin
    let $n$ be an integer;
    $n \leftarrow$ number of levels in $H$;
    let $S[n]$ be a global array of a set of valuations;
    $S[0] \leftarrow$ solver.all_solutions($H_0$);
    for $i \leftarrow 1$ to $n$ do
        $S[i] \leftarrow$ solver.filter($S[i-1]$, $H_i$);
    return $S[n]$;
end
```

Figure 2.13: The DeltaStar algorithm for S_R construction.

However, DeltaStar recomputes the solution in each recursive step causing significant overhead. In practice, it is only used as a general and theoretical framework for solution, from which efficient algorithms, such as DeltaBlue and Cassowary, are inspired and designed for some subset of the general problem space [21].

```
deltastar_t(solver, $H$)
begin
    let $n$ be an integer;
    $n \leftarrow$ number of levels in $H$;
    let $T[n]$ be a global array of a set of valuations;
    $T[0] \leftarrow$ solver.all_solutions($H_0$);
    for $i \leftarrow 1$ to $n$ do
        let $s$ be a set of valuations;
        $s \leftarrow \emptyset$;
        for each $\Theta$ in $T[i-1]$ do
            $s \leftarrow s \cup$ solver.filter($\Theta$, $H_i$);
        $T[i] \leftarrow s$;
    return union of all $\Theta$ in $T[n]$;
end
```

Figure 2.14: The DeltaStar algorithm for T_R construction.

2.4.5 A Plug-In Architecture of Constraint Hierarchy Solvers

In this section, we discuss a general framework (a Plug-In Architecture) [3] of CH solvers instead of focusing on a particular algorithm to solve constraint hierarchies.

There are four standard modules in this architecture: *meta-interpreter*, *general hierarchy solver*, *flat constraint solver*, and *comparator code*. The meta-interpreter and general hierarchy solver form the *kernel* of the architecture. The kernel is a generic part of the architecture such that it is independent of the chosen flat constraint solver and comparator. The meta-interpreter is very similar to a traditional Prolog meta-interpreter. It interprets constraints (goals) and then passes the constraints to other modules to perform the constraints solving tasks. A constraint is passed to the flat constraint solver if the constraint is a required constraint. However, a non-required constraint is passed to the general hierarchy solver. The general hierarchy solver can perform two tasks. It can add a non-required constraint to a constraint hierarchy and it can solve a collected constraint hierarchy along with the solution of the required constraints. This general hierarchy solver will try to satisfy a stronger level, and then a weaker level later on. They regard this method as a so-called *refining method* and this is independent of a chosen comparator.

The major property of this framework is the flexibility of the plug-in modules. It is possible to construct a constraint hierarchy solver with arbitrary flat constraint solver and comparator code. The flat constraint solver is a plug-in module used to determine the satisfaction of a constraint. The comparator code is another plug-in module used to define a particular comparator. Since the kernel modules are generic, it is possible to define a constraint hierarchy solver over arbitrary domain if a flat constraint solver for the domain exists. It is also possible to define a constraint hierarchy solver over arbitrary comparator, because the comparator is defined in the plug-in module.

Chapter 3

Local Consistency in Constraint Hierarchies

The classical notion of local consistency [37, 24] deals with the situation when variables and constraints contain inconsistent values. A CSP is locally consistent if all the inconsistent values are removed by determining a subset of constraints to the CSP (determining at a local level) and the solution to the CSP remains unchanged afterward. The main purpose of detecting local inconsistency in a classical CSP is to remove the inconsistent values from the variable domains and constraints. Hence, the CSP becomes "simpler" to solve if the size of the CSP is smaller. Local consistency had been proven to be an important concept in classical CSP [6, 30]. 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"* [9]. In particular, we borrow from the general notion of arc-consistency, semiring-based arc-consistency (SAC), used for SCSP [10]. We reformulate CH framework with a different notation in order to define the general notion of *constraint hierarchy k-consistency* (CH-k-C). We design and implement an enforcing algorithm in particular for CH-2-C namely CHAC. The correctness of the CHAC algorithm is established.

3.1 A Reformulation of Constraint Hierarchies

To facilitate subsequent illustration of the CH local consistency concept, we reformulate the CH framework [13] (particular in the definition of comparators and solution set) using a different notation.

3.1.1 Error Indicators

Let H be a constraint hierarchy with n non-required levels. Then, $H = \{H_0, \ldots, H_n\}$. For each level $i \in \{0, \ldots, n\}$, H_i is a set of labeled constraints in the form $H_i = \{c_1^i, \ldots, c_{k_i}^i\}$ where k_i is the number of constraints in level i. A valuation $\theta = \{v_1 \mapsto d_1, \ldots, v_N \mapsto d_N\}$ for a set of variables $V = \{v_1, \ldots, v_N\}$ means that each v_i is assigned the value d_i where $d_i \in D(v_i)$. Let c_b^a be the b^{th} constraint in H_a and θ a valuation. An error function $e(c_b^a\theta)$ measures how well a constraint c_b^a is satisfied by valuation θ. The value $e(c_b^a\theta)$ returned by an error function is an error value, denoted by ξ_b^a, indicating how nearly a constraint c_b^a is satisfied by valuation θ. We introduce a new notation called an *error indicator of a valuation* to represent the error values of a valuation θ applying to the constraint hierarchy H. An error indicator of a valuation θ for a set of variables V is a tuple of error values, denoted by $\vec{\xi_\theta}$, such that $\vec{\xi_\theta} = \langle\langle {\xi_\theta}_1^0, \ldots, {\xi_\theta}_{k_0}^0\rangle, \ldots, \langle {\xi_\theta}_1^n, \ldots, {\xi_\theta}_{k_n}^n\rangle\rangle$ where $\forall a \in \{0, \ldots, n\}, \forall b \in \{1, \ldots, k_a\}, {\xi_\theta}_b^a = e(c_b^a\theta)$ if $vars(c_b^a) \subseteq vars(\theta)$ and ${\xi_\theta}_b^a = 0$ if $vars(c_b^a) \not\subseteq vars(\theta)$. Intuitively, error indicators provide a measure of the "goodness" of valuations with respect to the constraint hierarchy H. Therefore, each possible valuation θ will be associated with a corresponding error indicator $\vec{\xi_\theta}$ to measure the degree of satisfaction with respect to the hierarchy. We use I to denote this set of error indicators such that I is a poset (partially ordered set), each element $\vec{\xi_\theta} \in I$ represents the degree of satisfaction of the corresponding valuation θ in this reformulation.

To explain the meaning of such a reformulation, we use the example in Figure 2.11 again. If a valuation $\theta = \{z \mapsto 2\}$ is given, then the associated error indicator of valuation $\vec{\xi_\theta}$ can be obtained easily by definition. The error indicator associated with valuation θ is $\vec{\xi_\theta} = \langle\langle\rangle, \langle\underline{1}, 0\rangle, \langle 0, 0, 0\rangle\rangle$. The underlined error values are returned by trivial error function. Since the constraint c_1^1 is $z = 1$ and the valuation $\theta = \{z \mapsto 2\}$, it is clear that $vars(c_1^1) \subseteq vars(\theta)$. ${\xi_\theta}_1^1$ is an error value returned by error function. However, this is not the case for other constraints (c_2^1, c_1^2, c_2^2, and c_3^2). For example, the constraint c_1^2 is $x + y \neq 2$, it is clear that $vars(c_1^2) \not\subseteq vars(\theta)$ ($\{x, y\} \not\subseteq \{z\}$). Therefore, ${\xi_\theta}_1^2$ is simply assigned with 0 to mean that the constraint will not be violated by this valuation. The same operation is applied to ${\xi_\theta}_2^2$ and ${\xi_\theta}_3^2$. Suppose valuation $\theta = \{x \mapsto 1, y \mapsto 2\}$, then $\vec{\xi_\theta} = \langle\langle\rangle, \langle 0, 0\rangle, \langle\underline{0}, \underline{0}, 0\rangle\rangle$. Similarly, if valuation $\theta = \{x \mapsto 2, y \mapsto 2, z \mapsto 1\}$, then $\vec{\xi_\theta} = \langle\langle\rangle, \langle\underline{0}, \underline{0}\rangle, \langle\underline{0}, \underline{0}, \underline{1}\rangle\rangle$.

3.1.2 A Reformulation of Comparators

The comparator predicate *better* in the original CH formulation is redefined using a *partial order*, denoted by $\prec$. We define $\prec$ to be irreflexive and transitive over the error indicators I with respect to a hierarchy H. Hence, it preserves the meaning of *better*. Intuitively, $\vec{\xi'} \prec \vec{\xi''}$ means $\vec{\xi''}$ is "better" than $\vec{\xi'}$. In general, $\prec$ will not provide a total ordering. That means we may have two error indicators $\vec{\xi'}$ and $\vec{\xi''}$ such that $\vec{\xi'} \not\prec \vec{\xi''} \wedge \vec{\xi''} \not\prec \vec{\xi'}$. For convenience, we define $\preceq$ such that $\vec{\xi'} \preceq \vec{\xi''} \rightarrow (\vec{\xi'} \prec \vec{\xi''} \vee \vec{\xi'} = \vec{\xi''})$.

We can redefine *locally-better* 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:

$$\vec{\xi_\theta} \prec_{l-b} \vec{\xi_\sigma} \equiv \exists l > 0 \text{ such that}$$
$$\forall i \in \{0, \ldots, l-1\}, \forall j \in \{1, \ldots, k_i\}, {\xi_\theta}_j^i = {\xi_\sigma}_j^i$$

$$\wedge \exists a \in \{1, \ldots, k_l\}, \xi_{\sigma a}^{\ l} < \xi_{\theta a}^{\ l}$$
$$\wedge \forall b \in \{1, \ldots, k_l\}, \xi_{\sigma b}^{\ l} \leq \xi_{\theta b}^{\ l}.$$

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 *globally-better* $\prec_{g-b}$, and its instances *weighted-sum-better* $\prec_{w-s-b}$, *worst-case-better* $\prec_{w-c-b}$, and *least-squares-better* $\prec_{l-s-b}$ respectively. Given any two valuations θ and σ, and the corresponding error indicators $\vec{\xi_\theta}$ and $\vec{\xi_\sigma}$:

$$\begin{aligned}\vec{\xi_\theta} \prec_{g-b} \vec{\xi_\sigma} \equiv\ & \exists l > 0 \text{ such that} \\ & \forall i \in \{0, \ldots, l-1\}, \\ & g(\langle \xi_{\theta 1}^{\ i}, \ldots, \xi_{\theta k_i}^{\ i} \rangle) = g(\langle \xi_{\sigma 1}^{\ i}, \ldots, \xi_{\sigma k_i}^{\ i} \rangle) \\ & \wedge g(\langle \xi_{\sigma 1}^{\ l}, \ldots, \xi_{\sigma k_l}^{\ l} \rangle) < g(\langle \xi_{\theta 1}^{\ l}, \ldots, \xi_{\theta k_l}^{\ l} \rangle), \\ & \text{where } g \text{ is a combining function for error values;}\end{aligned}$$

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

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

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

The following lemma gives the monotonicity of the introduced comparators, which shall be collectively denoted by $\prec_{better}$ and $\preceq_{better}$ in the rest of the thesis.

Lemma 1 Given any two error indicators $\vec{\xi'}$ and $\vec{\xi''}$. If for all ${\xi''}_j^i \leq {\xi'}_j^i$, then $\vec{\xi'} \preceq_{better} \vec{\xi''}$.

Proof. The above Lemma holds for *locally-better* $\prec_{l-b}$, *weighted-sum-better* $\prec_{w-s-b}$, *least-squares-better* $\prec_{l-s-b}$, and *worst-case-better* $\prec_{w-c-b}$. For *locally-better*, since for each level i and for each $j \in \{1, \ldots, k_i\}$, the error value ${\xi''}^i_j$ is less than or equal to ${\xi'}^i_j$. By definition, $\vec{\xi'} \preceq_{l-b} \vec{\xi''}$ holds. For *weighted-sum-better*, as the combining function g is $\sum_{j\in 1}^{k_i} w^i_j \xi^i_j$ for a particular level i and $\sum$ is monotonic, $g(\langle {\xi''}^i_1, \ldots, {\xi''}^i_{k_i} \rangle) \leq g(\langle {\xi'}^i_1, \ldots, {\xi'}^i_{k_i} \rangle)$ holds for each level i. Therefore, $\vec{\xi'} \preceq_{w-s-b} \vec{\xi''}$ holds. The same argument can be used to verify that $\vec{\xi'} \preceq_{l-s-b} \vec{\xi''}$ holds. For *worst-case-better*, the combining function g is $\max\{w^i_j \xi^i_j \mid j \in \{1, \ldots, k_i\}\}$ for a particular level i and max is monotonic, $g(\langle {\xi''}^i_1, \ldots, {\xi''}^i_{k_i} \rangle) \leq g(\langle {\xi'}^i_1, \ldots, {\xi'}^i_{k_i} \rangle)$ must hold for each level i. Hence, $\vec{\xi'} \preceq_{w-c-b} \vec{\xi''}$ holds. ∎

3.1.3 A Reformulation of Solution Set

The solution set of a constraint hierarchy is defined to be a set of valuations that satisfy all the required constraints and satisfy the non-required constraints as much as possible. We can define the solution set S by using error indicators for valuations as follows.

$$S_0 = \{\theta \mid i \in \{1, \ldots, k_0\}, {\xi_\theta}^0_i = 0\}, \text{ and}$$
$$S = \{\theta \mid \theta \in S_0, \forall \sigma \in S_0, \vec{\xi_\theta} \not\prec \vec{\xi_\sigma}\}.$$

The original meaning of solution set is preserved. The difference of this reformulation to the original formulation is that we use error indicators and partial order to define solution set.

3.2 Local Consistency in Classical CSPs

In this section we focus on node-consistency and arc-consistency algorithms which are common techniques to detect local inconsistency [6, 30]. 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 $C = \{3 \leq x, x < y\}$. P is node-inconsistent, since $d_x \in \{1,2\}$ and $\{x \mapsto d_x\}$ is not a solution of the unary constraint "$3 \leq x$." 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' would become $V = \{x, y\}$, $D(x) = \{3,4,5\}$, $D(y) = \{1,2,3,4,5\}$, and $C = \{3 \leq x, x < y\}$. Although P' is node-consistent, it is arc-inconsistent since $d_x \in \{5\}$ and $\{x \mapsto d_x\}$ cannot find support from $D(y)$ to satisfy the binary constraint "$x < y$." Also, $d_y \in \{1,2,3\}$ and $\{y \mapsto d_y\}$ 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 $C = \{3 \leq x, x < y\}$. 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-consistent and arc-consistent and equivalent to the original CSP.

We can use a different point of view to present the notion of local consistency in classical CSPs. Given a CSP P, we associate a constraint set C_u ("u" stands for "unary constraint") to P. Each constraint in C_u can explicitly indicate the implicit inconsistency information in P. A tuple $\langle P, C_u \rangle$ represents the consistency status of P.

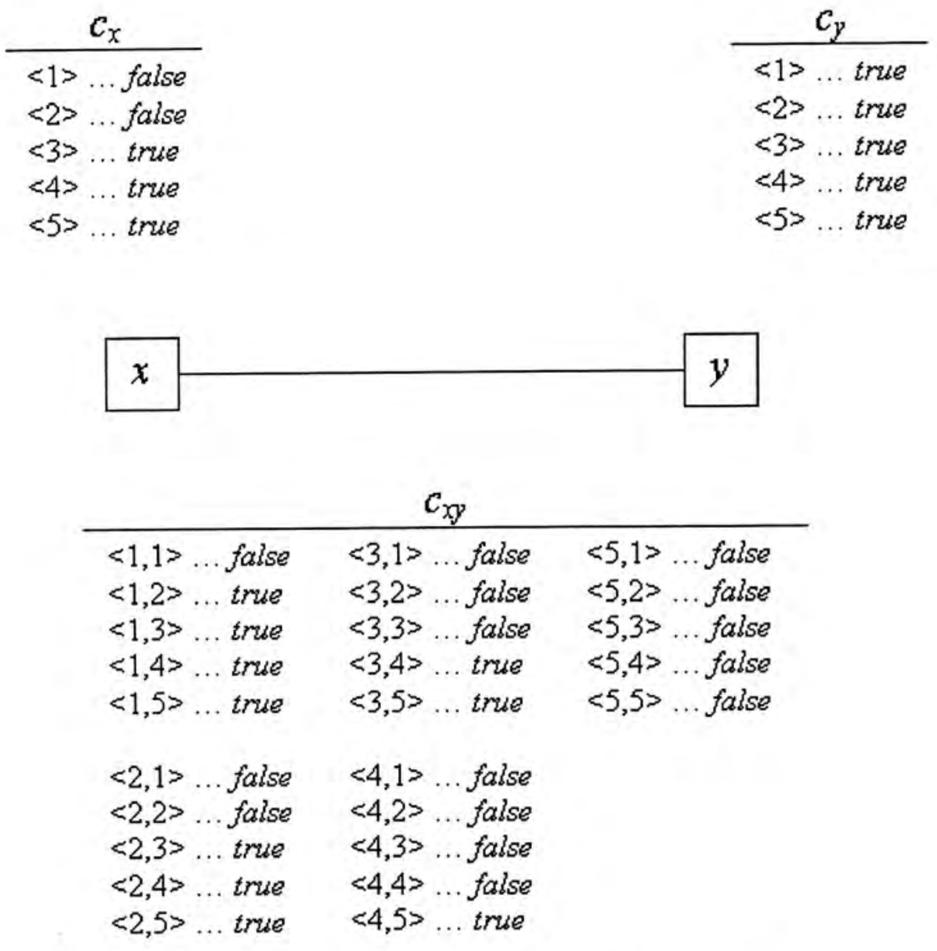
We use the previous example to illustrate the idea. Given a CSP P where $V = \{x, y\}$, $D(x) = \{1,2,3,4,5\}$, $D(y) = \{1,2,3,4,5\}$, $C = \{3 \leq x, x < y\}$, and P is associated with C_u in order to explicitly indicate the inconsistency information in P. Initially, the consistency status of P is represented by a tuple $\langle P, \emptyset \rangle$. $C_u = \emptyset$ means no explicit inconsistency information is known currently. P is node-inconsistent, since $D(x)$ contains inconsistency domain values 1 and 2. This implicit node-inconsistency information in P should be explicitly indicated by the constraint set C_u, but $C_u = \emptyset$. P becomes node-consistent, because the tuple $\langle P, \{x \neq 1, x \neq 2\} \rangle$ is node-consistent such that the tuple expresses the same information as P' in the previous example. Similarly, P is arc-inconsistent, because the tuple $\langle P, \{x \neq 1, x \neq 2\} \rangle$ is arc-inconsistent. However, P becomes arc-consistent as the tuple $\langle P, \{x \neq 1, x \neq 2, x \neq 5, y \neq 1, y \neq 2, y \neq 3\} \rangle$ is arc-consistent such that the tuple expresses the same information as P'' in the previous example. The variable domains are not reduced in such a point of view, but the equivalent local inconsistency information is recorded.

3.3 Local Consistency in SCSPs

SCSPs [9] extends classical CSPs by allowing non-crisp features. Hence, classical CSP is an instance of SCSP over the c-semiring $S_{CSP} = \langle \{true, false\}, \vee, \wedge, false, true \rangle$. In SCSPs, a general notion of local consistency is proposed [9] and we focus on semiring-based arc-consistency (SAC) [10].

We use the same example in Section 3.2 to illustrate the idea of SAC. Given a CSP P where $V = \{x, y\}$, $D(x) = \{1,2,3,4,5\}$, $D(y) = \{1,2,3,4,5\}$, and $C = \{3 \leq x, x < y\}$. This CSP can be modeled as an instance of SCSP over the c-semiring $S_{CSP} = \langle \{true, false\}, \vee, \wedge, false, true \rangle$ as shown in Figure 3.1. We use a graph-like representation to represent a CSP. The nodes and arcs in the graph are variables and constraints respectively. The tuples and the corresponding labels in the graph are the tuples of domain values and the corresponding semiring values respectively. In Figure 3.1, c_x is the unary constraint "$3 \leq x$" and c_{xy} is the binary constraint "$x < y$." Since there is no unary constraint for variable y, all the semiring values corresponding to c_y is *true* that means y is unconstrained.

Since the semiring values in the constraints c_x and c_y (def is the value of constraint) do not coincide with those in the constraints $\{c_x \otimes c_{xy} \otimes c_y\} \Downarrow_x$ and $\{c_x \otimes c_{xy} \otimes c_y\} \Downarrow_y$ respectively, P is not SAC. The constraint $c_x \otimes c_{xy} \otimes c_y$, which is obtained from constraint combination, is shown in Table 3.1. The constraint projections of $c_x \otimes c_{xy} \otimes c_y$ over $\{x\}$ and $\{y\}$ are shown in Table 3.2. If the semiring values in the constraints c_x and c_y are made to coincide with those in the constraints $\{c_x \otimes c_{xy} \otimes c_y\} \Downarrow_x$ and $\{c_x \otimes c_{xy} \otimes c_y\} \Downarrow_y$ respectively as shown in Figure 3.2, then P is SAC.



c_x
<1> ... *false*
<2> ... *false*
<3> ... *true*
<4> ... *true*
<5> ... *true*

c_y
<1> ... *true*
<2> ... *true*
<3> ... *true*
<4> ... *true*
<5> ... *true*

c_{xy}		
<1,1> ... *false*	<3,1> ... *false*	<5,1> ... *false*
<1,2> ... *true*	<3,2> ... *false*	<5,2> ... *false*
<1,3> ... *true*	<3,3> ... *false*	<5,3> ... *false*
<1,4> ... *true*	<3,4> ... *true*	<5,4> ... *false*
<1,5> ... *true*	<3,5> ... *true*	<5,5> ... *false*
<2,1> ... *false*	<4,1> ... *false*	
<2,2> ... *false*	<4,2> ... *false*	
<2,3> ... *true*	<4,3> ... *false*	
<2,4> ... *true*	<4,4> ... *false*	
<2,5> ... *true*	<4,5> ... *true*	

Figure 3.1: A constraint graph of a CSP.

We can also use a different point of view to present the notion of SAC. We

$c_x \otimes c_{xy} \otimes c_y$	$c_x \otimes c_{xy} \otimes c_y$	$c_x \otimes c_{xy} \otimes c_y$
$\langle 1,1\rangle \ldots \mathit{false}$	$\langle 3,1\rangle \ldots \mathit{false}$	$\langle 5,1\rangle \ldots \mathit{false}$
$\langle 1,2\rangle \ldots \mathit{false}$	$\langle 3,2\rangle \ldots \mathit{false}$	$\langle 5,2\rangle \ldots \mathit{false}$
$\langle 1,3\rangle \ldots \mathit{false}$	$\langle 3,3\rangle \ldots \mathit{false}$	$\langle 5,3\rangle \ldots \mathit{false}$
$\langle 1,4\rangle \ldots \mathit{false}$	$\langle 3,4\rangle \ldots \mathit{true}$	$\langle 5,4\rangle \ldots \mathit{false}$
$\langle 1,5\rangle \ldots \mathit{false}$	$\langle 3,5\rangle \ldots \mathit{true}$	$\langle 5,5\rangle \ldots \mathit{false}$
$\langle 2,1\rangle \ldots \mathit{false}$	$\langle 4,1\rangle \ldots \mathit{false}$	
$\langle 2,2\rangle \ldots \mathit{false}$	$\langle 4,2\rangle \ldots \mathit{false}$	
$\langle 2,3\rangle \ldots \mathit{false}$	$\langle 4,3\rangle \ldots \mathit{false}$	
$\langle 2,4\rangle \ldots \mathit{false}$	$\langle 4,4\rangle \ldots \mathit{false}$	
$\langle 2,5\rangle \ldots \mathit{false}$	$\langle 4,5\rangle \ldots \mathit{true}$	

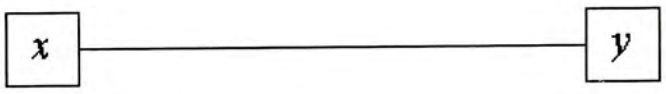
Table 3.1: A constraint combination.

$\{c_x \otimes c_{xy} \otimes c_y\} \Downarrow_x$	$\{c_x \otimes c_{xy} \otimes c_y\} \Downarrow_y$
$\langle 1\rangle \ldots \mathit{false}$	$\langle 1\rangle \ldots \mathit{false}$
$\langle 2\rangle \ldots \mathit{false}$	$\langle 2\rangle \ldots \mathit{false}$
$\langle 3\rangle \ldots \mathit{true}$	$\langle 3\rangle \ldots \mathit{false}$
$\langle 4\rangle \ldots \mathit{true}$	$\langle 4\rangle \ldots \mathit{true}$
$\langle 5\rangle \ldots \mathit{false}$	$\langle 5\rangle \ldots \mathit{true}$

Table 3.2: Constraint projections.

c_x
<1> ... *false*
<2> ... *false*
<3> ... *true*
<4> ... *true*
<5> ... *false*

c_y
<1> ... *false*
<2> ... *false*
<3> ... *false*
<4> ... *true*
<5> ... *true*



	c_{xy}	
<1,1> ... *false*	<3,1> ... *false*	<5,1> ... *false*
<1,2> ... *true*	<3,2> ... *false*	<5,2> ... *false*
<1,3> ... *true*	<3,3> ... *false*	<5,3> ... *false*
<1,4> ... *true*	<3,4> ... *true*	<5,4> ... *false*
<1,5> ... *true*	<3,5> ... *true*	<5,5> ... *false*
<2,1> ... *false*	<4,1> ... *false*	
<2,2> ... *false*	<4,2> ... *false*	
<2,3> ... *true*	<4,3> ... *false*	
<2,4> ... *true*	<4,4> ... *false*	
<2,5> ... *true*	<4,5> ... *true*	

Figure 3.2: A constraint graph of a SAC CSP.

associate each CSP P, also an SCSP, with a constraint set C_u, which contains constraints of the form $x = d$ for all variables x in P and for all $d \in D(x)$. Each constraint in C_u is associated with a semiring value either *true* or *false*. The semiring value will explicitly indicate the implicit inconsistency information in P. We can use a tuple $\langle P, C_u \rangle$ to represent the semiring-based arc-consistency status in P. We use the same example to illustrate the idea.

Given a CSP P where $V = \{x, y\}$, $D(x) = \{1, 2, 3, 4, 5\}$, $D(y) = \{1, 2, 3, 4, 5\}$, $C = \{3 \leq x, x < y\}$, and P is associated with C_u in order to explicitly indicate the inconsistency information in P. The inconsistency information in P is represented by a tuple $\langle P, \{x = 1(true), x = 2(true), x = 3(true), x = 4(true), x = 5(true), y = 1(true), y = 2(true), y = 3(true), y = 4(true), y = 5(true)\}\rangle$ initially. No explicit inconsistency information is known currently and P is semiring-based arc-inconsistent. However, an SAC algorithm can transform P to become SAC such that this tuple becomes $\langle P, \{x = 1(false), x = 2(false), x = 3(true), x = 4(true), x = 5(false), y = 1(false), y = 2(false), y = 3(false), y = 4(true), y = 5(true)\}\rangle$. It is easy to check that this tuple expresses the same information as P'' in Section 3.2.

Although the domain size of the CSP P remains unchanged after applying the SAC algorithm, we still gain useful information since we are "explicitating some implicit constraints" of P to C_u. Based on this inconsistency information, a search algorithm can know not to try the domain values that are marked $false$. Hence, SAC is a generalization of classical node-consistency and arc-consistency.

3.4 Local Consistency in CHs

We adapt the general notion of local consistency in SCSP for CH, and define *constraint hierarchy k-consistency* (CH-k-C). Given a CH P associated with a constraint set C_u, which contains constraints of the form $x = d$ for all variables x in P and for all $d \in D(x)$. Each constraint $c \in C_u$ is associated with an error indicator $\vec{\xi_c}$, which stores the (partial) inconsistency information in P. We can use a tuple $\langle P, C_u \rangle$ to represent the constraint hierarchy k-consistency status of P. Since arc-consistency algorithm is a common technique to detect local inconsistency in classical CSPs [6, 30], we design and implement an algorithm to enforce CH-2-C, which we also called *constraint hierarchy arc-consistency* (CHAC). Hence, CHAC is CH-2-C.

3.4.1 The Operations of Error Indicator

Before we can formally define CH-k-C, we need two operations, $\mathcal{MAX}$ and $\mathcal{MIN}$, on error indicators. Given a CH P with n non-required levels and any two error indicators, $\vec{\xi_\theta}, \vec{\xi_\sigma} \in I$ such that $\vec{\xi_\theta} = \langle\langle \xi_{\theta}{}_1^0, \ldots, \xi_{\theta}{}_{k_0}^0 \rangle, \ldots, \langle \xi_{\theta}{}_1^n, \ldots, \xi_{\theta}{}_{k_n}^n \rangle\rangle$ and $\vec{\xi_\sigma} = \langle\langle \xi_{\sigma}{}_1^0, \ldots, \xi_{\sigma}{}_{k_0}^0 \rangle, \ldots, \langle \xi_{\sigma}{}_1^n, \ldots, \xi_{\sigma}{}_{k_n}^n \rangle\rangle$, for P, $\mathcal{MAX}$ and $\mathcal{MIN}$ are defined as:

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

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

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

Given two error indicators, the $\mathcal{MIN}$ (or $\mathcal{MAX}$) operation combines the two indicators by taking the best (or the worst) of both worlds. We can easily verify that $\mathcal{MAX}$ and $\mathcal{MIN}$ are commutative. Thus, it makes sense to write $\mathcal{MAX}\{\vec{\xi_1}, \ldots, \vec{\xi_K}\}$ (or $\mathcal{MIN}\{\vec{\xi_1}, \ldots, \vec{\xi_K}\}$) which is equal to $\mathcal{MAX}(\vec{\xi_1}, \ldots, \vec{\xi_K})$ (or $\mathcal{MIN}(\vec{\xi_1}, \ldots, \vec{\xi_K})$), for any $K > 2$.

Lemma 2 If P is a CH with variables V, $x \in V$, $d \in D(x)$, $Y \subset V - \{x\}$, and $|Y| = k-1$, where $k \in \{1, \ldots, |V|\}$. Then, $\mathcal{MIN}\{\vec{\xi_\theta} \mid vars(\theta) = V \wedge (x \mapsto d) \in \theta\} \preceq_{better} \mathcal{MIN}\{\vec{\xi_\theta} \mid vars(\theta) = \{x\} \cup Y \wedge (x \mapsto d) \in \theta\}$.

Proof. Given a set of variables Y, where $y_i \in Y, \forall i \in \{1, \ldots, k-1\}$, an error indicator $\vec{\xi_\sigma}$, where $\sigma = \{x \mapsto d, y_1 \mapsto d_{y_1}, \ldots, y_{k-1} \mapsto d_{y_{k-1}}\}$ for some $d_{y_i} \in D(y_i)$, and an error indicator $\vec{\xi_\gamma} \in \{\vec{\xi_\theta} \mid vars(\theta) = V \wedge \sigma \subseteq \theta\}$. By Lemma 1, $\vec{\xi_\gamma} \preceq_{better} \vec{\xi_\sigma}$. On the R.H.S. of $\preceq_{better}$, the error values of all n-ary ($n > k$) constraints and constraints not involving $\{x\} \cup Y$ must be 0. The combined error values of all k-ary constraints involving $\{x\} \cup Y$ must be smaller (better) than the corresponding error values on the combined error indicator on the L.H.S. Hence, the L.H.S. is worse than the R.H.S. ∎

The following can again be proved using a simple application of Lemma 1.

Lemma 3 Given error indicators $\vec{\xi'}$, $\vec{\xi''}$, and $\vec{\xi'''}$. $(\vec{\xi'} \preceq_{better} \vec{\xi''} \wedge \vec{\xi'} \preceq_{better} \vec{\xi'''}) \rightarrow \vec{\xi'} \preceq_{better} \mathcal{MAX}(\vec{\xi''}, \vec{\xi'''})$.

Given a CH P with variables V. If $x \in V$ and $d \in D(x)$, we define $approx_k(x \mapsto d) = \mathcal{MAX}\{\mathcal{MIN}\{\vec{\xi_\theta} \mid vars(\theta) = \{x\} \cup Y \wedge (x \mapsto d) \in \theta\} \mid Y \subset V - \{x\} \wedge |Y| = k-1\}$, where $k \in \{1, \ldots, |V|\}$. We call it *k-approximation*, which provides estimates of the "goodness" of valuations involving the assignment $x \mapsto d$. Since the error indicators of all valuations involving $x \mapsto d$ might not be comparable, we can only give an approximation, and $approx_{|V|}(x \mapsto d)$ is the best possible approximation (since $\vec{\xi_\theta} \preceq_{better} approx_{|V|}(x \mapsto d)$ for all θ such that $(x \mapsto d) \in \theta$), we call it *best approximation.* However, calculating $approx_{|V|}(x \mapsto d)$ is computationally expensive, and $approx_2(x \mapsto d)$ gives a more practical approximation (most commonly used technique in classical CSPs), we call it *practical approximation.* The following theorem states that $approx_k(x \mapsto d)$ is an approximation of $approx_{|V|}(x \mapsto d)$.

Theorem 1 If P is a CH with variables V, $x \in V$ and $d \in D(x)$, then $approx_{|V|}(x \mapsto d) \preceq_{better} approx_k(x \mapsto d)$, if $k < |V|$.

Proof. By Lemma 2, given any $Y \subset V - \{x\}$ and $|Y| = k - 1$, the combined best case errors $\vec{\xi}_Y$ among the valuations $\{\theta \mid vars(\theta) = \{x\} \cup Y \wedge (x \mapsto d) \in \theta\}$ is better than $approx_{|V|}(x \mapsto d)$. It is thus easy to check that $\mathcal{MAX}\{\vec{\xi}_Y \mid Y \subset V - \{x\} \wedge |Y| = k - 1\}$ must also be better than $approx_{|V|}(x \mapsto d)$ by simply application of Lemma 3. ∎

3.4.2 Constraint Hierarchy k-Consistency

Given a CH P with a constraint set C_u. P is constraint hierarchy k-consistent (CH-k-C) if the associated error indicator of each constraint in C_u explicitly indicates the implicit inconsistency information in P. Formally, we define CH-k-C as follows.

Definition 1 (CH-k-C) *Given P a CH with variables V and C_u the associated constraint set. Let $I_{C_u} = \{\vec{\xi}_c \mid c \in C_u\}$. P is CH-k-C if for all $\vec{\xi}_{x=d} \in I_{C_u}$ such that $approx_{|V|}(x \mapsto d) \preceq_{better} \vec{\xi}_{x=d} \preceq_{better} approx_k(x \mapsto d)$, where $k \in \{1, \ldots, |V|\}$.*

To perform constraint checking on unary and binary constraints is the most commonly used technique for detecting local inconsistency, arc-consistency, in classical CSPs. Therefore, we discuss CHAC (or CH-2-C) and provide a CHAC enforcement algorithm in the following.

The error indicator $\vec{\xi}_{x=d} \in I_{C_u}$ stores the error information for the variable assignment $x \mapsto d$ for P. The definition of CHAC requires that $\vec{\xi}_{x=d}$ must be "between" $approx_{|V|}(x \mapsto d)$ and $approx_2(x \mapsto d)$ for all $x \in V$ and $d \in D(x)$. We would use a simple example to explain the definition in more detail. Given a CH P where $V = \{x, y, z\}$, $D(x) = \{1\}$, $D(y) = \{1,2\}$, $D(z) = \{1,2\}$, $H = \{\emptyset, \{x > y, x = 2\}, \{y = 3, z < y\}, \{z = 1, x + y + z > 4\}\}$, $C_u = \{x = 1, y = 1, y = 2, z = 1, z = 2\}$, and $I_{C_u} = \{\vec{\xi}_{x=1}, \vec{\xi}_{y=1}, \vec{\xi}_{y=2}, \vec{\xi}_{z=1}, \vec{\xi}_{z=2}\}$. Initially, $\vec{\xi}_{x=1} = \vec{\xi}_{y=1} = \vec{\xi}_{y=2} = \vec{\xi}_{z=1} = \vec{\xi}_{z=2} = \langle\langle\rangle, \langle 0,0\rangle, \langle 0,0\rangle, \langle 0,0\rangle\rangle$. P is obviously not CHAC as described, but it becomes CHAC if the error indicators in I_{C_u} are as listed in the third column of Table 3.3. The error indicators $\vec{\xi}_{x=1}$ and $\vec{\xi}_{y=2}$ are equal to the best and practical approximations. The error indicator $\vec{\xi}_{y=1}$ is better than the best approximation and equal to the practical approximation. The error indicator $\vec{\xi}_{z=2}$ is equal to the best approximation and worse than the practical approximation. The error indicator $\vec{\xi}_{z=1}$ is "between" the best and practical approximations.

$v \mapsto d$	$approx_{\lvert V\rvert}(v \mapsto d)$	$\vec{\xi}_{v=d}$	$approx_2(v \mapsto d)$
$x \mapsto 1$	$\langle\langle\rangle, \langle 1,1\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle$	$\langle\langle\rangle, \langle 1,1\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle$	$\langle\langle\rangle, \langle 1,1\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle$
$y \mapsto 1$	$\langle\langle\rangle, \langle 1,1\rangle, \langle 1,1\rangle, \langle 0,\underline{1}\rangle\rangle$	$\langle\langle\rangle, \langle 1,1\rangle, \langle 1,1\rangle, \langle 0,\underline{0}\rangle\rangle$	$\langle\langle\rangle, \langle 1,1\rangle, \langle 1,1\rangle, \langle 0,\underline{0}\rangle\rangle$
$y \mapsto 2$	$\langle\langle\rangle, \langle 1,1\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle$	$\langle\langle\rangle, \langle 1,1\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle$	$\langle\langle\rangle, \langle 1,1\rangle, \langle 1,0\rangle, \langle 0,0\rangle\rangle$
$z \mapsto 1$	$\langle\langle\rangle, \langle \underline{1},1\rangle, \langle 1,0\rangle, \langle 0,\underline{1}\rangle\rangle$	$\langle\langle\rangle, \langle \underline{1},1\rangle, \langle 1,0\rangle, \langle 0,\underline{0}\rangle\rangle$	$\langle\langle\rangle, \langle \underline{0},1\rangle, \langle 1,0\rangle, \langle 0,\underline{0}\rangle\rangle$
$z \mapsto 2$	$\langle\langle\rangle, \langle \underline{1},1\rangle, \langle 1,1\rangle, \langle 1,0\rangle\rangle$	$\langle\langle\rangle, \langle \underline{1},1\rangle, \langle 1,1\rangle, \langle 1,0\rangle\rangle$	$\langle\langle\rangle, \langle \underline{0},1\rangle, \langle 1,1\rangle, \langle 1,0\rangle\rangle$

Table 3.3: A table of error indicators.

3.4.3 A Comparsion between CHAC and PAC

In Section 3.3, we have discussed an arc-consistency technique, the SAC algorithm [10], in SCSPs. The SAC algorithm (or a local consistency algorithm) approximates a complete solution algorithm. When applying the SAC algorithm, the implicit information is explicated by updating the semiring values corresponding to the values in the variable domains of a SCSP. The scheme of SAC is useful in theory, but the SAC algorithm has a heavy complexity for most applications [7]. Therefore, Bistarelli *et. al.* [7] propose a more efficient arc-consistency technique in SCSPs, which is called *partial arc-consistency* (PAC).

A *local consistency rule* is used to explicate the implicit information of a problem. When applying a local consistency rule (or simply a rule) to a problem, the resulting problem is the same as the original problem in terms of the solution set. This is the idea of the SAC algorithm originally. The notion of an *approximate function* ϕ is introduced for a local consistency rule. The idea of an approximation function is to replace the complex computation in the SAC algorithm by a simpler one when we concern arc-consistency. Two particular approximation functions are introduced: ϕ_{best} and ϕ_{worst}. The approximation function ϕ_{best} actually does no approximation. Therefore, the heavy complexity of the SAC algorithm will not be reduced when applying ϕ_{best}, and the resulting rule performs the same computation as in the original SAC algorithm. The approximation function ϕ_{worst} does no domain reduction. This means no implicit information can be explicated when applying ϕ_{worst}. A SCSP is PAC if the implicit information obtained by applying ϕ is "less" than the those obtained by applying ϕ_{best} and is "more" than those obtained by applying ϕ_{worst}.

The idea of approximation functions, ϕ_{best} and ϕ_{worst}, in PAC is similar to the idea of the best and practical approximations, $approx_{|V|}(v \mapsto d)$ and $approx_2(v \mapsto d)$, in our proposal. Both of them avoid calculating the "best" approximation of error information as it is computationally expensive. Instead, they calculate a more "practical" approximation of error information "between" the upper and lower bounds. The subtle difference is in the definitions of the upper and lower bounds. We can get "more" implicit information for a CH when calculating the best approximation $approx_{|V|}(v \mapsto d)$. The best approximation of our proposal is calculated by checking all the n-ary constraints of the problem. We can also get "more" implicit information for a SCSP when applying the approximation function ϕ_{best}. However, the implicit information is calculated by the ordinary SAC algorithm (without approximation). Similarly, we get "less" implicit information for a CH when the practical approximation $approx_2(v \mapsto d)$

of our proposal is calculated. The implicit information is calculated by checking all the unary and binary constraints involving variable v of the problem. But we get "less" implicit information for a SCSP when applying the approximation function ϕ_{worst}, as it does no reduction.

3.4.4 The CHAC Algorithm

The purpose of a CHAC algorithm is thus to explicate and place in C_u the implicit error information in a CH that is otherwise not visible. Such an algorithm is given in Figure 3.6. The subroutines **chnc_pri** and **chac_pri**, in Figures 3.4 and 3.5 respectively, are responsible for ensuring the consistency of unary and binary constraints respectively. After executing lines 1 to 5 in the pseudocode of CHAC algorithm, each error indicators $\vec{\xi}_{x=d} \in I_{C_u}$ should have contained the errors of the unary and binary constraints involving variable x. We use the same example in Section 3.4.2 to illustrate the idea. Before applying CHAC algorithm $\vec{\xi}_{x=1} = \langle\langle\rangle, \langle 0, 0\rangle, \langle 0, 0\rangle, \langle 0, 0\rangle\rangle$. After executing lines 1 to 5 in the pseudocode of CHAC algorithm, $\vec{\xi}_{x=1} = \langle\langle\rangle, \langle \underline{1}, \underline{1}\rangle, \langle 0, 0\rangle, \langle 0, 0\rangle\rangle$ where the underlined values are the error values returned by trivial error function. It is possible that some error values should have been updated according to the definition of CHAC, but they are missed, such as the boxed values in $\vec{\xi}_{x=1} = \langle\langle\rangle, \langle \underline{1}, \underline{1}\rangle, \langle \boxed{0}, 0\rangle, \langle \boxed{0}, 0\rangle\rangle$. In this case, $\vec{\xi}_{x=d}$ cannot capture all the errors of the valuations involving only variables x and $y \in V - \{x\}$. Lines 6 to 11 in the pseudocode of the CHAC algorithm help to recover this missing error information so that the whole of $approx_2(x \mapsto d)$ is computed in C_u. The CHAC algorithm ensures that the error indicator $\vec{\xi}_{x=d}$ are updated to reach at least $approx_2(x \mapsto d)$ for all $(x = d) \in C_u$, and sometimes reveals more error information (thus producing "worse" error indicators). In fact, lines 6 to 11 in the pseudocode of CHAC algorithm also attempts, though not always succeeds, to further update each $\vec{\xi}_{x=d} \in I_{C_u}$ to a "worse" value towards $approx_{|V|}(x \mapsto d)$, but $\vec{\xi}_{x=d}$ will never be worse than $approx_{|V|}(x \mapsto d)$. In this example, $\vec{\xi}_{z=1}$ is strictly in "between" $approx_2(z \mapsto 1)$ and $approx_{|V|}(z \mapsto 1)$, whereas $\vec{\xi}_{z=2}$ is the same as $approx_{|V|}(z \mapsto 2)$ after applying CHAC algorithm.

```
   update($x$, $y$, $c$, $l$, $k$, $D$, $I_{C_u}$)
   begin
1      let $\xi_{min}$ be an error value;
2      for each $d_x \in D(x)$ do
3          $\xi_{min} \leftarrow \infty$;
4          for each $d_y \in D(y)$ do
5              let $\theta = \{x \mapsto d_x, y \mapsto d_y\}$;
6              if $e(c\theta) < \xi_{min}$ then
7                  $\xi_{min} \leftarrow e(c\theta)$;
8          let $\vec{\xi} = \vec{\xi}_{x=d_x} \in I_{C_u}$;
9          if $\xi_k^l < \xi_{min}$ then
10             $\xi_k^l \leftarrow \xi_{min}$;
11     return $I_{C_u}$;
   end
```

Figure 3.3: A subroutine to update error indicators.

3.4.5 Time and Space Complexities of the CHAC Algorithm

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 of the subroutine **chnc_pri** is simply of $O(n_d)$, since the only repeating operations, lines 4 to 7 in Figure 3.4, 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** is of $O({n_d}^2)$, since there exists operations, lines 5 to 7 in Figure 3.3, locating inside a double loop. Therefore, in the worst case, the time complexity of the subroutine **chac_pri** is of $O({n_d}^2)$ as shown

```
chnc_pri($c$, $l$, $k$, $D$, $I_{C_u}$)
begin
    if $|vars(c)| = 1$ then
        let $\{x\} = vars(c)$;
        for each $d \in D(x)$ do
            let $\theta = \{x \mapsto d\}$;
            let $\vec{\xi} = \vec{\xi}_{x=d} \in I_{C_u}$;
            if $\xi^l_k < e(c\theta)$ then
                $\xi^l_k \leftarrow e(c\theta)$;
    return $I_{C_u}$;
end
```

Figure 3.4: A subroutine to check unary constraints.

```
chac_pri($c$, $l$, $k$, $D$, $I_{C_u}$)
begin
    if $|vars(c)| = 2$ then
        let $\{x, y\} = vars(c)$;
        # Update each $\vec{\xi}_{x=d_x} \in I_{C_u}$
        $I_{C_u} \leftarrow$ update($x$, $y$, $c$, $l$, $k$, $D$, $I_{C_u}$);
        # Update each $\vec{\xi}_{y=d_y} \in I_{C_u}$
        $I_{C_u} \leftarrow$ update($y$, $x$, $c$, $l$, $k$, $D$, $I_{C_u}$);
    return $I_{C_u}$;
end
```

Figure 3.5: A subroutine to check binary constraints.

Algorithm 1: The CHAC algorithm.

```
chac($H$, $V$, $D$, $I_{C_u}$)
begin
    for $l \leftarrow 1$ to $n$ do
        for $k \leftarrow 1$ to $|H_l|$ do
            let $c$ be the $k^{th}$ constraint in $H_l$;
            $I_{C_u} \leftarrow$ chnc_pri($c$, $l$, $k$, $D$, $I_{C_u}$);
            $I_{C_u} \leftarrow$ chac_pri($c$, $l$, $k$, $D$, $I_{C_u}$);
    for each $\vec{\xi}_{x=d_x} \in I_{C_u}$ do
        for each $y \in V - \{x\}$ do
            let $\vec{\xi}$ be an error indicator s.t. each $\xi^i_j = \infty$;
            for each $\vec{\xi}_{y=d_y} \in I_{C_u}$ do
                $\vec{\xi} \leftarrow \mathcal{MIN}(\vec{\xi}, \vec{\xi}_{y=d_y})$;
            $\vec{\xi}_{x=d_x} \leftarrow \mathcal{MAX}(\vec{\xi}_{x=d_x}, \vec{\xi})$;
    return $I_{C_u}$;
end
```

Figure 3.6: The CHAC algorithm.

in Figure 3.5. Lines 3 to 5 in the pseudocode of the CHAC algorithm are the operations for checking constraints as shown in Figure 3.6. Since these operations should repeat until all the constraints are considered, the time complexity should be of $O(n_c {n_d}^2)$. Lines 6 to 11 in the pseudocode of the CHAC algorithm help to recover missing error information. Many steps are required for error information recovery, since this operation is required for each error indicator corresponding to the constraint in C_u. The time complexity for lines 6 to 11 in the pseudocode of the CHAC algorithm is of $O({n_d}^2 {n_v}^2)$. Therefore, the worst case time complexity of the CHAC algorithm is of $O((n_c + {n_v}^2){n_d}^2)$.

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 CHAC algorithm depends on the number of error indicators corresponding to the constraints in C_u. Therefore, we require $n_v n_d$ error indicators. The space complexity of the CHAC algorithm is simply of $O(n_v n_d n_c)$ in the worst case.

3.4.6 Correctness of the CHAC Algorithm

Let $\vec{\xi}_{x=d}$ be an error indicator, $approx_2(x \mapsto d)$, where $approx_2(x \mapsto d) = \mathcal{MAX}\{\mathcal{MIN}\{\vec{\xi}_\theta \mid vars(\theta) = \{x, z\} \wedge (x \mapsto d) \in \theta\} \mid z \in V - \{x\}\}$. Let $\vec{\xi'}_{x=d}$ be an error indicator corresponding to constraint $(x = d) \in C_u$, which is computed by a CHAC algorithm as shown in Figure 3.6. By definition, for all constraints $c^i_j \in H_i$, where $i \in \{1, \ldots, n\}$ and n is the number of non-required levels in hierarchy H, the error value $(\xi_{x=d})^i_j$ corresponding to constraint c^i_j is 0 if $|vars(c^i_j)| > 2$ $(vars(c^i_j) \not\subseteq vars(\theta))$. Since the CHAC algorithm only performs constraint check for unary constraints and binary constraints (line 4 to 5 in **chac**), the error value $({\xi'}_{x=d})^i_j$ corresponding to c^i_j will not be modified and equal to 0 if $|vars(c^i_j)| > 2$. Therefore, the computed error value $({\xi'}_{x=d})^i_j$ is equivalent to the error value $(\xi_{x=d})^i_j$ of $approx_2(x \mapsto d)$ if $|vars(c^i_j)| > 2$.

For the case $vars(c^i_j) = \{x\}$, the error value of the practical approximation, $(\xi_{x=d})^i_j$, corresponding to constraint c^i_j is $\max\{\min\{\xi_\theta{}^i_j \mid vars(\theta) = \{x,z\} \wedge (x \mapsto d) \in \theta\} \mid z \in V - \{x\}\}$. Since $vars(c^i_j) = \{x\}$ and $x \mapsto d$ exists in each valuation θ, the error value $(\xi_{x=d})^i_j$ is simply equal to $e(c^i_j(\{x \mapsto d\}))$. The subroutine **chnc_pri** (line 4 in **chac**) updates error value $(\xi'_{x=d})^i_j$ to $e(c^i_j(\{x \mapsto d\}))$ (line 7 in **chnc_pri**) when $vars(c^i_j) = \{x\}$. Therefore, the computed error value $(\xi'_{x=d})^i_j$ is equivalent to the error value $(\xi_{x=d})^i_j$ of $approx_2(x \mapsto d)$ if $vars(c^i_j) = \{x\}$.

For the case $vars(c^i_j) = \{x, y\}$, where $x \neq y$, the error value $(\xi_{x=d})^i_j$ corresponding to constraint c^i_j is $\max\{\min\{\xi_\theta{}^i_j \mid vars(\theta) = \{x,z\} \wedge (x \mapsto d) \in \theta\} \mid z \in V - \{x\}\}$. When an error function is applied to $c^i_j\sigma$, where $\{\sigma | vars(\sigma) = \{x,w\} \wedge (x \mapsto d) \in \sigma \wedge w \in V - \{x,y\}\}$, the error value of $e(c^i_j\sigma) = 0$ $(vars(c^i_j) \not\subseteq vars(\sigma))$. The error value $(\xi_{x=d})^i_j$ is simply equal to $\min\{\xi_\theta{}^i_j \mid vars(\theta) = \{x,y\} \wedge (x \mapsto d) \in \theta\}$. The subroutine **chac_pri** (line 5 in **chac**) updates error value $(\xi'_{x=d})^i_j$ to $\min\{\xi_\theta{}^i_j \mid vars(\theta) = \{x,y\} \wedge (x \mapsto d) \in \theta\}$ (line 3 in **chac_pri**) when $vars(c^i_j) = \{x, y\}$. Therefore, the computed error value $(\xi'_{x=d})^i_j$ is equivalent to the error value $(\xi_{x=d})^i_j$ of $approx_2(x \mapsto d)$ if $vars(c^i_j) = \{x, y\}$.

For the case $vars(c^i_j) = \{y\}$, where $y \neq x$, the error value of the practical approximation $(\xi_{x=d})^i_j$ corresponding to constraint c^i_j is $\max\{\min\{\xi_\theta{}^i_j \mid vars(\theta) = \{x,z\} \wedge (x \mapsto d) \in \theta\} \mid z \in V - \{x\}\}$. For the same reason as the previous case, the error value $(\xi_{x=d})^i_j$ is simply equal to $\min\{\xi_\theta{}^i_j \mid vars(\theta) = \{x,y\} \wedge (x \mapsto d) \in \theta\}$. It is possible to simplify to $\min\{\xi_\theta{}^i_j \mid vars(\theta) = \{y\}\}$ in this case. The pseudocode from line 7 to 11 in **chac** updates the error value $(\xi'_{x=d})^i_j$ to $\min\{(\xi'_{y=d_y})^i_j \mid d_y \in D(y)\}$, where $\vec{\xi'}_{y=d_y}$ is an error indicator corresponding to constraint $(y = d_y) \in C_u$. Since each error value $(\xi'_{y=d_y})^i_j$ is computed by performing constraint check on constraint c^i_j. This implies $\min\{(\xi'_{y=d_y})^i_j \mid d_y \in D(y)\} = \min\{\xi_\theta{}^i_j \mid vars(\theta) = \{y\}\}$. Therefore, the computed error value $(\xi'_{x=d})^i_j$ is equivalent to the error value $(\xi_{x=d})^i_j$ of $approx_2(x \mapsto d)$ if $vars(c^i_j) = \{y\}$.

For the case $vars(c^i_j) = \{u, v\}$, where $u \neq x \wedge v \neq x \wedge u \neq v$, the error value of practical approximation $(\xi_{x=d})^i_j$ corresponding to constraint c^i_j is 0, since the binary constraint c^i_j does not contain variable x. However, the pseudocode from line 7 to 11 in **chac** updates the error value $({\xi'}_{x=d})^i_j$ to $\max\{\min\{({\xi'}_{u=d_u})^i_j \mid d_u \in D(u)\}, \min\{({\xi'}_{v=d_v})^i_j \mid d_v \in D(v)\}\}$, where $\vec{\xi'}_{u=d_u}$ and $\vec{\xi'}_{v=d_v}$ are error indicators corresponding to constraints $(u = d_u)$ and $(v = d_v)$ respectively, $(u = d_u), (v = d_v) \in C_u$. Since each error value $({\xi'}_{u=d_u})^i_j$ (or $({\xi'}_{v=d_v})^i_j$) is computed by performing constraint check on constraint c^i_j. This implies $\max\{\min\{({\xi'}_{u=d_u})^i_j \mid d_u \in D(u)\}, \min\{({\xi'}_{v=d_v})^i_j \mid d_v \in D(v)\}\} \geq 0$. Therefore, the computed error value $({\xi'}_{x=d})^i_j$ is greater than or equal to the error value $(\xi_{x=d})^i_j$ of $approx_2(x \mapsto d)$ if $vars(c^i_j) = \{u, v\}$.

Given any two error indicators $\vec{\xi}_{x=d}$ $(approx_2(x \mapsto d))$ and $\vec{\xi'}_{x=d}$ (an error indicator corresponding to constraint $(x = d) \in C_u$). The previous arguments show that for all $(\xi_{x=d})^i_j \leq ({\xi'}_{x=d})^i_j$, this implies $\vec{\xi'}_{x=d} \preceq_{better} \vec{\xi}_{x=d}$ by Lemma 1. Given any two error indicators $\vec{\xi''}_{x=d}$ $(approx_{|V|}(x \mapsto d))$ and $\vec{\xi'}_{x=d}$ (an error indicator corresponding to constraint $(x = d) \in C_u$). The error value $({\xi'}_{x=d})^i_j$ corresponding to constraint c^i_j such that $|vars(c^i_j)| > 2$ is equal to 0. However, the error value $({\xi''}_{x=d})^i_j$ corresponding to constraint c^i_j such that $|vars(c^i_j)| > 2$ should be greater than or equal to 0, since constraint check is not restricted to unary and binary constraints for the best approximation. Therefore, for all $({\xi'}_{x=d})^i_j \leq ({\xi''}_{x=d})^i_j$, this implies $\vec{\xi''}_{x=d} \preceq_{better} \vec{\xi'}_{x=d}$ by Lemma 1. Hence, our CHAC algorithm is correct in the sense that the computed error indicator $(\vec{\xi'}_{x=d} \in I_{C_u})$ is "between" the best approximation $(approx_{|V|}(x \mapsto d))$ and the practical approximation $(approx_2(x \mapsto d))$.

Chapter 4

A Consistency-Based Finite Domain Constraint Hierarchy Solver

The simplest way to find the solution set of a CH is to construct a complete search tree for the problem, so that we can calculate the error values of each valuation at the leaf nodes and compare all the valuations. However, traversing the complete search tree and comparing all the valuations are tedious and time-consuming. We propose to combine the CHAC and the Branch-and-Bound algorithms so as to prune non-fruitful branches of the search tree and at the same time guarantee that no solutions are missed.

4.1 The Branch-and-Bound CHAC Solver

The backbone of our solver is the ordinary Branch-and-Bound algorithm [42], since CH-solving is an optimization problem. The Branch-and-Bound algorithm always maintains the set of potential best solutions collected so far. The idea is to invoke the CHAC algorithm at each node in the search tree, hoping that the overhead of executing the CHAC algorithm can be more than compensated by the pruning that can take place. At a CHAC 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_{C_u} 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 is shown in Figure 4.3, which is a simple adaptation of the basic Branch-and-Bound solver with the CHAC algorithm. The numbered lines give the backbone of the algorithm, while the unnumbered lines are new additions to enable CHAC enforcement. Note that our algorithm also relies on classical node-consistency and arc-consistency algorithms [37] to perform pruning using the required constraints in H_0 in lines 1 and 2 in the pseudocode of **bb_solv**. Lines 6 to 17 deal with the case of a leaf node. The CHAC algorithm is invoked between lines 17 and 18. Lines 18 to 22 perform the basic variable instantiation (or searching) recursively. The call to the subroutine **go** (between lines 21 and 22) determines whether the error indicator of the variable assignment of the selected branch in I_{C_u} of the current node is not worse than the error indicator of each of the collected solutions so far. The current variable instantiation proceeds only if **go** returns *true*.

4.2 Correctness of the Branch-and-Bound CHAC Solver

The subroutine **cal_error_value** as shown in Figure 4.1 is a function that maps a valuation θ to an error indicator $\vec{\xi_\theta}$ corresponding to θ. Given a hierarchy H and any two valuations σ and θ, the corresponding error indicators are $\vec{\xi_\sigma}$ and $\vec{\xi_\theta}$ respectively. If $\sigma \subset \theta$, it is possible to have a constraint $c_j^i \in H_i$ such that $(vars(c_j^i) \subseteq vars(\theta)) \wedge (vars(c_j^i) \nsubseteq vars(\sigma))$. However, it is not possible to have a constraint $c_j^i \in H_i$ such that $(vars(c_j^i) \nsubseteq vars(\theta)) \wedge (vars(c_j^i) \subseteq vars(\sigma))$. This implies the fact that for all ${\xi_\sigma}_j^i \leq {\xi_\theta}_j^i$ if $\sigma \subset \theta$. By simple application to Lemma 1, $\vec{\xi_\theta} \preceq_{better} \vec{\xi_\sigma}$ if $\sigma \subset \theta$.

To apply the Branch-and-Bound algorithm in CH, it is necessary to define a function f to evaluate valuations and f should be monotonic. Such a function, f, in **bb_solv** is **cal_error_value**. We simply use f to represent **cal_error_value** in the following explanations. The input of f is a valuation θ and the output of f is an error indicator $\vec{\xi_\theta}$, corresponding to the input valuation θ, over comparator $\preceq_{better}$. Since $\vec{\xi_\theta} \preceq_{better} \vec{\xi_\sigma}$ if $\sigma \subset \theta$, f is monotonic such that $f(\theta) \preceq_{better} f(\sigma)$. In addition, it is necessary to define a lower bound B. The bound B in **bb_solv** is an error indicator $\vec{\xi}$ (or a set of error indicators) corresponding to each valuation in the current best valuation set when global comparators are used (or corresponding to the current best incomparable valuations when local comparator is used). For simplicity we assume B is an error indicator in this case. If $\sigma \subset \theta$ and $f(\sigma) \preceq_{better} B$, then by monotonicity of f, $f(\theta) \preceq_{better} B$. Given a set of n valuation $\{\theta_1, \ldots, \theta_n\}$, it is easy to check that for any $\theta_i \in \{\theta_1, \ldots, \theta_n\}$, $f(\theta_i) \preceq_{better} \mathcal{MIN}\{f(\theta_j) \mid j \in \{1, \ldots, n\}\}$ by simple application of Lemma 1. This implies each error indicator of a valuation γ corresponding to a leaf node of the search tree, where $(x \mapsto d) \in \gamma$, is worse than $approx_{|V|}(x \mapsto d)$. If $\vec{\xi}_{x=d}$ is an error indicator corresponding to constraint $(x = d) \in C_u$ (in any internal node of the search tree) and $\vec{\xi}_{x=d} \preceq_{better} B$, then $approx_{|V|}(x \mapsto d) \preceq_{better} B$. Therefore, each valuation γ corresponding to a leaf node of the search tree, where $(x \mapsto d) \in \gamma$, should be worse than the current best valuation set. The subroutine **go** in Figure 4.2 performs such a *check bound* operation [42] in **bb_solv**.

```
cal_error_value($H$, $\theta$, $\vec{\xi}_\theta$)
begin
1    for $l \leftarrow 1$ to $n$ do
2        for $k \leftarrow 1$ to $|H_l|$ do
3            let $c$ be the $k^{th}$ constraint in $H_l$;
4            $\xi^l_{\theta k} \leftarrow e(c\theta)$;
5    return $\vec{\xi}_\theta$;
end
```

Figure 4.1: A subroutine to calculate error values.

```
go($\vec{\xi}_c$, $S_0$, $I_{S_0}$, $\prec_{better}$)
begin
1    for each $\theta \in S_0$ do
2        if $\vec{\xi}_c \prec_{better} \vec{\xi}_\theta$ then
3            return false;
4    return true;
end
```

Figure 4.2: A subroutine to check bound.

Algorithm 2: The Branch-and-Bound CHAC solver.

```
bb_solv($H$, $V$, $D$, $S_0$, in out $I_{S_0}$, $C_u$, $I_{C_u}$, $\prec_{better}$)
begin
       # Any classical node-consistency algorithm
1      $D \leftarrow$ nc_algorithm($H_0$, $D$);
       # Any classical arc-consistency algorithm
2      $D \leftarrow$ ac_algorithm($H_0$, $D$);
3      if $D$ contains an empty variable domain then
4          return $S_0$;
5      else if $D$ contains all singleton variable domain then
6          let $\theta$ be the valuation corresponding to $D$;
7          let $\vec{\xi}_\theta$ be the error indicator corresponding to $\theta$;
8          $\vec{\xi}_\theta \leftarrow$ cal_error_values($H$, $\theta$, $\vec{\xi}_\theta$);
9          for each $\sigma \in S_0$ do
10             if $\vec{\xi}_\sigma \prec_{better} \vec{\xi}_\theta$ then
11                 $S_0 \leftarrow S_0 - \{\sigma\}$;
12                 $I_{S_0} \leftarrow I_{S_0} - \{\vec{\xi}_\sigma\}$;
13             else if $\vec{\xi}_\theta \prec_{better} \vec{\xi}_\sigma$ then
14                 return $S_0$;
15         $S_0 \leftarrow S_0 \cup \{\theta\}$;
16         $I_{S_0} \leftarrow I_{S_0} \cup \{\vec{\xi}_\theta\}$;
17         return $S_0$;
       for each $(x = d) \in C_u$ do
           if $d \notin D(x)$ then
               $C_u \leftarrow C_u - \{x = d\}$;
               $I_{C_u} \leftarrow I_{C_u} - \{\vec{\xi}_{x=d}\}$;
       $I_{C_u} \leftarrow$ chac($H$, $V$, $D$, $I_{C_u}$);
18     choose variable $x \in V$ for which $|D(x)| \geq 2$;
19     let $W$ be a variable domain;
20     $W \leftarrow D(x)$;
21     for each $d \in W$ do
           if go($\vec{\xi}_{x=d}$, $S_0$, $I_{S_0}$, $\prec_{better}$) then
22             $S_0 \leftarrow$ bb_solv($\{H_0 \wedge (x = d), H_1, \ldots, H_n\}$, $V$, $D$, $S_0$, $I_{S_0}$, $C_u$, $I_{C_u}$,
               $\prec_{better}$);
23     return $S_0$;
end
```

Figure 4.3: The Branch-and-Bound CHAC solver.

4.3 An Example Execution Trace

We use the example in Figure 2.11 (applying trivial error function and *locally-better* comparator) to illustrate the functioning of our proposed solver in more detail. Given a constraint hierarchy problem P_A such that $H = \{\emptyset, \{c_1^1, c_2^1\}, \{c_1^2, c_2^2, c_3^2\}\}$, $V = \{x, y, z\}, D(x) = \{1, 2\}, D(y) = \{1, 2\}$, and $D(z) = \{1, 2\}$. $C_u = \{x = 1, x = 2, y = 1, y = 2, z = 1, z = 2\}$ and the associated set of error indicators is $I_{C_u} = \{\vec{\xi}_{x=1}, \vec{\xi}_{x=2}, \vec{\xi}_{y=1}, \vec{\xi}_{y=2}, \vec{\xi}_{z=1}, \vec{\xi}_{z=2}\}$. For each $\vec{\xi}_c$ in I_{C_u} we initialize $\vec{\xi}_c$ to $\langle\langle 0, 0\rangle, \langle 0, 0, 0\rangle\rangle$. We ignore the error values for required constraints, since CHAC concerns only non-required constraints. Figure 4.4 depicts how the complete search is traversed (by following the search nodes in alphabetical order), and when pruning takes place.

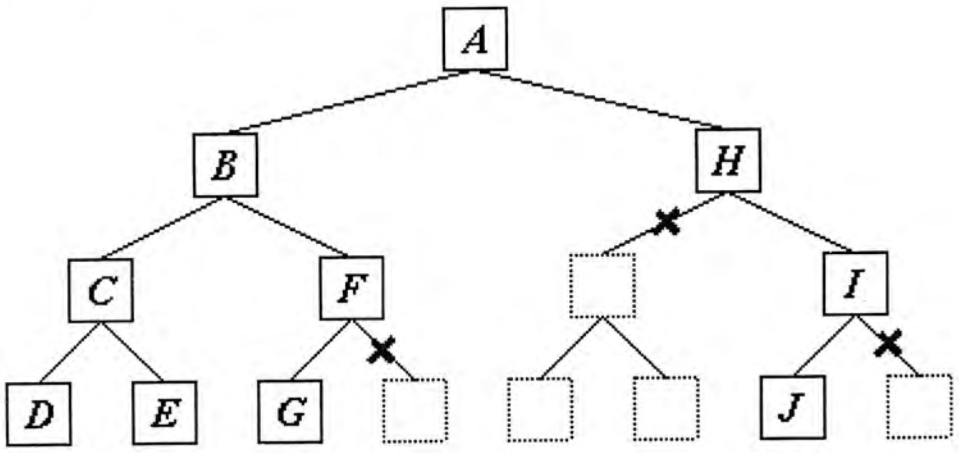


Figure 4.4: A search tree example.

At node A, there being no required constraints in H_0 implies that no domain values are removed as a result of the classical node-consistency and arc-consistency algorithms. However, P_A is not CHAC. After applying the CHAC algorithm on $\langle P_A, C_u\rangle$, I_{C_u} would be updated so that $\vec{\xi}_{x=1} = \vec{\xi}_{x=2} = \vec{\xi}_{y=2} = \vec{\xi}_{z=1} = \langle\langle 0, 0\rangle, \langle 0, 0, 0\rangle\rangle$, $\vec{\xi}_{y=1} = \langle\langle 0, 0\rangle, \langle 0, 1, 0\rangle\rangle$, and $\vec{\xi}_{z=2} = \langle\langle 1, 0\rangle, \langle 0, 0, 0\rangle\rangle$. Assuming that we pick variable x and instantiate it with 1 from $D(x)$, the subroutine **go**($\vec{\xi}_{x=1}$, $\emptyset$, $\emptyset$, $\prec_{l-b}$) returns true. Hence, **bb_solv** is invoked recursively. At B and C, the flow of control is similar, except that both $D(x)$ and $D(y)$

becomes the singleton $\{1\}$.

At node D, since all variables are instantiated to the value 1, the constraint hierarchy becomes P_D such that $H = \{\{x = 1, y = 1, z = 1\}, \{c_1^1, c_2^1\}, \{c_1^2, c_2^2, c_3^2\}\}$, $V = \{x, y, z\}$, $D(x) = \{1\}$, $D(y) = \{1\}$, and $D(z) = \{1, 2\}$. Since there is a required constraint $z = 1$ in H_0, the domain $D(z)$ is updated also to $\{1\}$. Thus D is a leaf node with a complete valuation. S_0 is empty, but would be updated to $\{\theta_1\}$ where $\theta_1 = \{x \mapsto 1, y \mapsto 1, z \mapsto 1\}$ and the associated error indicator $\vec{\xi}_{\theta_1} = \langle\langle 0, 1\rangle, \langle 1, 1, 0\rangle\rangle$.

At node E, P_E has the form $H = \{\{x = 1, y = 1, z = 2\}, \{c_1^1, c_2^1\}, \{c_1^2, c_2^2, c_3^2\}\}$, $V = \{x, y, z\}$, $D(x) = \{1\}$, $D(y) = \{1\}$, and $D(z) = \{1, 2\}$. After instantiating z, node E is also a leaf node with valuation $\theta_2 = \{x \mapsto 1, y \mapsto 1, z \mapsto 2\}$ and the associated error indicator is $\vec{\xi}_{\theta_2} = \langle\langle 1, 0\rangle, \langle 1, 1, 0\rangle\rangle$. Since $\vec{\xi}_{\theta_1}$ and $\vec{\xi}_{\theta_2}$ are incomparable, θ_2 would also be added to S_0.

Search proceeds next to node F, where P_F is defined by $H = \{\{x = 1, y = 2\}, \{c_1^1, c_2^1\}, \{c_1^2, c_2^2, c_3^2\}\}$, $V = \{x, y, z\}$, $D(x) = \{1\}$, $D(y) = \{1, 2\}$, and $D(z) = \{1, 2\}$. The instantiation of y by the required constraint $y = 2$ causes $D(y)$ to become $\{2\}$. However, P_F is not CHAC. The CHAC algorithm would update I_{C_u} for P_F such that $\vec{\xi}_{x=1} = \vec{\xi}_{y=2} = \vec{\xi}_{z=1} = \langle\langle 0, 0\rangle, \langle 0, 0, 0\rangle\rangle$, and $\vec{\xi}_{z=2} = \langle\langle 1, 1\rangle, \langle 0, 0, 0\rangle\rangle$. Then, search would pick variable z and instantiate it to 1 from $D(z)$. Since $\vec{\xi}_{z=1} \not\prec_{l-b} \vec{\xi}_{\theta_1}$ and $\vec{\xi}_{z=1} \not\prec_{l-b} \vec{\xi}_{\theta_2}$, the subroutine **go**($\vec{\xi}_{z=1}$, $\{\theta_1, \theta_2\}$, $\{\vec{\xi}_{\theta_1}, \vec{\xi}_{\theta_2}\}$, $\prec_{l-b}$) would return true, and **bb_solv** is invoked recursively.

Similar to nodes D and E, node G is also a leaf node with valuation $\theta_3 = \{x \mapsto 1, y \mapsto 2, z \mapsto 1\}$ and the associated error indicator is $\vec{\xi}_{\theta_3} = \langle\langle 0, 0\rangle, \langle 0, 0, 0\rangle\rangle$. Since $\vec{\xi}_{\theta_1} \prec_{l-b} \vec{\xi}_{\theta_3}$ and $\vec{\xi}_{\theta_2} \prec_{l-b} \vec{\xi}_{\theta_3}$, this implies $\vec{\xi}_{\theta_1}$ and $\vec{\xi}_{\theta_2}$ would be removed from S_0 and replaced by θ_3. Hence, $S_0 = \{\theta_3\}$. Upon backtracking, F is visited again. This time, we can make use of the error indicators in I_{C_u} computed previously, since node F is already CHAC. Here, $\vec{\xi}_{x=1} = \vec{\xi}_{y=2} = \vec{\xi}_{z=1} = \vec{\xi}_{\theta_3}$ but $\vec{\xi}_{z=2} \prec_{l-b} \vec{\xi}_{\theta_3}$, the subroutine **go**($\vec{\xi}_{z=2}$, $\{\theta_3\}$, $\{\vec{\xi}_{\theta_3}\}$, $\prec_{l-b}$) would return false, and backtracking occurs immediately without visiting the right child of node F. Similarly, the left subtree of node H and the right child of I are pruned as shown in Figure 4.4. In summary, our proposed solver prunes the subtrees of a node only when the estimated error in the current node is already worse than the errors of the potential solutions collected so far. In other words, proceeding further from that node would only increase the error, thus never yielding a better valuation than the ones collected so far.

4.4 Experiments and Results

We implement our proposed Branch-and-Bound CHAC solver (in Figure 4.3) for three reasons. First, we want to test the correctness of the solver. Second, we want to examine the efficiency of the solver. Third, we try to investigate the properties of the solver, in particular the pruning power, the memory requirement, and the overhead of the CHAC algorithm, among the four comparators: *locally-better*, *weighted-sum-better*, *worst-case-better*, and *least-squares-better.*

Since IHCS [39] is not maintained by anyone [15], we cannot get either the program or the benchmark of IHCS for comparison. DeltaStar [23] requires to store all the valuations in S_{i-1} for level H_{i-1}, then it invokes a subroutine *filter* to remove the "worse" valuations in S_{i-1} for level H_i recursively (Section 2.4.4). If no required constraint for a given CH, then S_0 contains all the possible valuations. We found that DeltaStar requires a lot of memories for program execution. For example, we test DeltaStar for a CH with 10 variables and each variable domain with 10 elements. For most of the time, S_0 requires more than 512MB of memories for storing valuations. We encounter failure due to insufficient memories. Therefore, DeltaStar is a theoretical general framework for solution [21]. To use the semiring S_{g-b} (Section 2.4.3) for solving finite domain CH, we need to encode E^n (the set of semiring values A) when applying clp(FD, S). However, the size of A is limited to 32 elements [29]. The reason for this limitation is the fact that an element of A is encoded in a word (32 bits) for efficiency reasons and there is no simple way to extend this size [28]. Therefore, we cannot solve any of our benchmarks according to this limitation.

We compare the performance of our solver and the reified constraint approach by Lua (the Lua's solver hereafter) [36]. 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 and it is simply a Generate-and-Test mechanism, in ILOG Solver 4.4 [35]. In order to provide a (simple) basic Branch-and-Bound solver for comparison, we define an alternative *goal* G_b. The basic Branch-and-Bound solver S_b ("b" stands for "Branch-and-Bound") is obtained by implementing additional comparators in G_b. 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 accumulated errors are calculated by performing constraint checking at each search node. For example, a given CH with three variables $\{x, y, z\}$ and two strong constraints $\{x + y = 3, x - z = 1\}$. Suppose that variable x has been instantiated to 1, variable y is being instantiated to 1 at the moment, and variable z has not been instantiated yet. Then, the accumulated errors are 1 and 0 corresponding to constraints "$x + y = 3$" and "$x - z = 1$" if trivial error function is applied. Since the values assigned to variables x and y have been known and the constraint "$x + y = 3$" is being violated at the moment, the error of constraint "$x + y = 3$" is 1. However, the error of constraint "$x - z = 1$" is unknown, as variable z has not been instantiated. Hence, the error is 0 meaning that no error can be calculated at the moment. 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 "CHAC") 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 CHAC 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* (Section 2.4.2). 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* (Section 2.4.2). 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. This implementation design ensures "fairness" in our comparisons, since all the solvers share the same backbone.

4.4.1 Experimental Setup

We benchmark the performance of our solver S_c by conducting two different experiments. In the first experiment, we want to examine the efficiency, the memory requirement, and the pruning power of our solver. We also want to know the overhead of the CHAC algorithm. We use different problem instances, which are randomly generated, for different comparators. In the second experiment, we want to investigate the performance, in terms of execution time, of our solver for different comparators. Therefore, we use same problem instances for different comparators. For simplicity reason, we apply trivial error function to test our solver 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 local comparator. Since there is a lack of benchmarks for finite domain CH [20, 15, 4, 52], we randomly generate CHs for our testing. For each comparator, we benchmark the performance of S_c in three different ways. First, we set up an experiment that consists of 4 sets of randomly generated CHs: 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 $i \in \{1, 2, 3, 4\}$. All problems do not have any required constraints to make them more "difficult" to solve.

Second, we set up an experiment that consists of 4 sets of randomly generated CHs: P''_1, P''_2, P''_3, and P''_4, each of which contains 15 problem instances. 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 $i \in \{1, 2, 3, 4\}$.

Third, we set up an experiment that consists of 4 sets of randomly generated CHs: P'''_1, P'''_2, P'''_3, and P'''_4, each of which contains 15 problem instances. 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 $\forall j \in \{1, \ldots, i+1\}, |H_j| = 5$.

Our experiments are conducted on Sun Ultra 5/400 workstations with 256MB RAM. We collect the following information of solver S_i (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
- The overhead (for enforcing the CHAC algorithm) O_c of solver S_c
- The number of failures (or backtracking) $\#F_i$ in searching

In Lua's reified constraint approach, backtracking will be performed when there exists an empty variable domain (after applying local consistency algorithm) or the values in error vector are "worse" than the bound during search (Section 2.4.2). We use a simple example as shown in Figure 4.5 to illustrate backtracking in Lua's solver. If values in error vector are "worse" than the bound during search (case I in Figure 4.5), then failure will be detected in S_r and the counter $\#F_r$ will be incremented by 1. Suppose the variable domain $D(y)$ is an empty domain after applying local consistency algorithm at node "$x \mapsto 2$" (case II in Figure 4.5), then backtracking should be performed and the counter $\#F_r$ will be incremented by 1. However, there is no failure or backtracking if $3 \in D(x)$ is removed after applying local consistency algorithm in S_r (case III in Figure 4.5). Hence, the counter $\#F_r$ remains unchanged ($\#F_r = 2$).

Since values in variable domains will not be removed (after applying the CHAC algorithm) in our approach, backtracking will be performed only when the check bound operation (**go** in Figure 4.3) returns *false*. We use a simple example as shown in Figure 4.6 to illustrate backtracking in our solver. For all

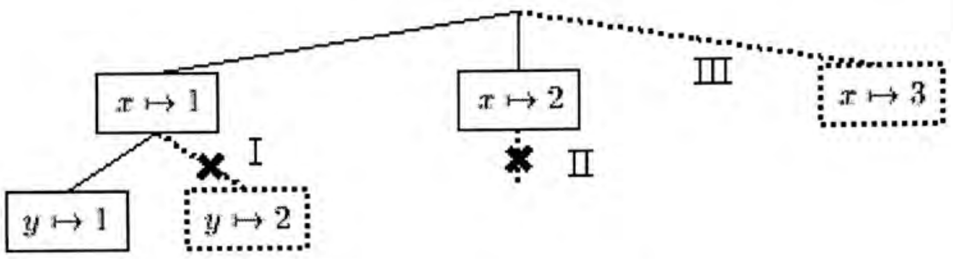


Figure 4.5: Backtracking of Lua's solver.

the cases (I, II, and III), the failures are detected by performing the check bound operation in S_c. When the subroutine **go** returns *false*, the counter $\#F_c$ will be incremented by 1. Suppose the number of internal nodes and leaf nodes visited by S_r and S_c are the same, the number of failures $\#F_r$ and $\#F_c$ may not be the same ($\#F_r = 2$, but $\#F_c = 4$ in this simple example).

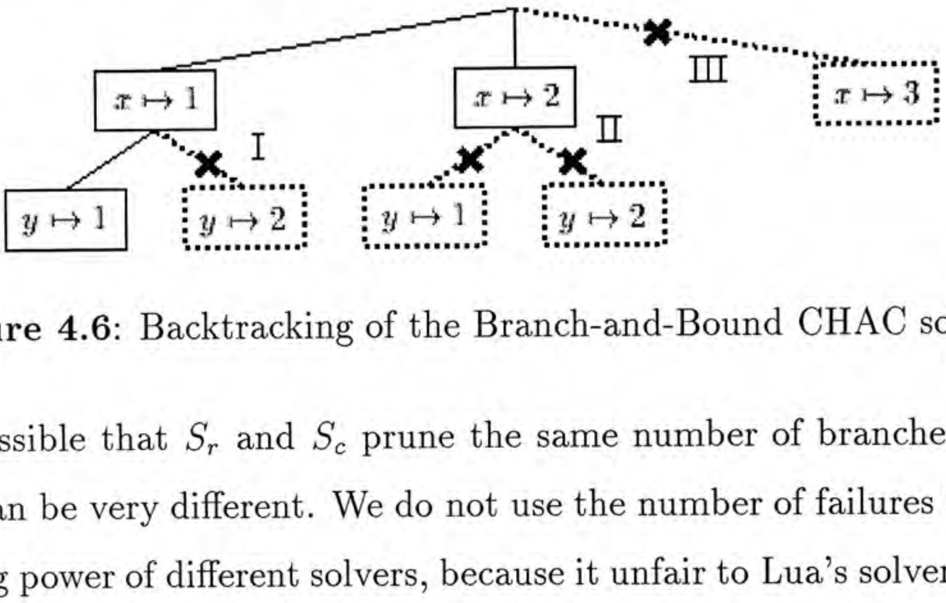


Figure 4.6: Backtracking of the Branch-and-Bound CHAC solver.

It is possible that S_r and S_c prune the same number of branches, but $\#F_r$ and $\#F_c$ can be very different. We do not use the number of failures to compare the pruning power of different solvers, because it unfair to Lua's solver. However, we also report such information.

4.4.2 The First Experiment

We randomly generate 720 problem instances for the first experiment. We benchmark the performance of each comparator (*locally-better*, *weighted-sum-better*, *worst-case-better*, and *least-squares-better*) using 180 problem instances. We want to examine the efficiency, the memory requirement, and the pruning power of our solver, as well as the overhead of the CHAC algorithm, from this large-scale experiment.

Experiments for the *locally-better* comparator

We randomly generate 180 problem instances (3 experiments $\times$ 4 sets of CHs $\times$ 15 instances) as the benchmark problems for the *locally-better* comparator. We record the results in three different ways: varying the size of variable domains (P'_1, P'_2, P'_3, and P'_4), varying the number of variables (P''_1, P''_2, P''_3, and P''_4), and varying the number of hierarchies (P'''_1, P'''_2, P'''_3, and P'''_4). We use the same experimental setup for other comparators (*weighted-sum-better*, *worst-case-better*, and *least-squares-better*).

We use the ratio T_g/T_c as a measurement of efficiency of our solver S_c corresponding to solver S_g. Similarly, we use the ratio T_b/T_c (or T_r/T_c) as a measurement of efficiency of our solver corresponding to the basic Branch-and-Bound solver S_b (or Lua's solver S_r). The larger the value of the ratio T_i/T_c ($i \in \{g, b, r\}$) is, the better the performance of our solver S_c corresponding to solver S_i in terms of execution time. For example, solver S_c is faster than solver S_g, with a factor 3.2 on average and with a factor 1.56 in terms of median, for the set of CHs in P'_1 as shown in Table 4.1. If the mean ratio of T_i/T_c is larger than the median ratio of T_i/T_c, then this implies our solver is faster than solver S_i with a relatively large factor for some problem instances.

The ratio M_i/M_c is used to compare the memory requirement of solver S_i and our solver S_c. For example, solver S_g requires less memory than our solver as the mean ratio of M_g/M_c is less than or equal to 1 as shown in Tables 4.1, 4.3, and 4.5. Our solver requires more memories than solver S_g, since extra memories are required to store the consistency information in solver S_c.

A choice point corresponds to a branching node in a search tree. If the number of choice points is small, then the number of branching nodes is also small. We use the ratios $\#L_i/\#L_c$ and $\#C_i/\#C_c$ as a measurement of pruning power of solver S_c corresponding to solver S_i. The larger the values of $\#L_i/\#L_c$ and $\#C_i/\#C_c$ are, the better the pruning power of S_c.

We use the ratio O_c/T_c as a measurement of the overhead for enforcing CHAC in solver S_c. The larger the value of O_c/T_c is, the more the proportion of time is required to enforce the CHAC algorithm. Experimental results show that the mean ratio of O_c/T_c is very close to the median ratio of O_c/T_c as shown in Tables 4.1, 4.3, and 4.5. We can observe that our solver requires more than half of the execution time in enforcing the CHAC algorithm. The mean (or median) ratio of O_c/T_c almost remains constant, even the problem size increases as shown in Tables 4.1and 4.3. However, the mean (or median) ratio of O_c/T_c strictly increases when the number of hierarchies increases as shown in Table 4.5.

Experimental results show that our solver can produce more pruning when the size of the problem instances are getting larger. The mean ratios of $\#L_g/\#L_c$ (or $\#L_b/\#L_c$) and $\#C_g/\#C_c$ (or $\#C_b/\#C_c$) strictly increases when the size of variable domains increases as shown in Table 4.1 (or Table 4.2). Therefore, our solver can solve a "large" problem more efficient than solvers S_g and S_b. Results also show that the mean ratio of T_g/T_c (or T_b/T_c) strictly increases when the size of variable domains increases as shown in Table 4.1 (or Table 4.2). When the size of the problem instances increases in terms of increasing the number of variables, we observe similar results as those of increasing the size of the variable domains. The mean ratios of T_g/T_c and T_b/T_c increase as shown in Tables 4.3 and 4.4. Although they do not strictly increase, results show that the pruning power of our solver is guaranteed as the mean ratios of $\#L_g/\#L_c$ and $\#L_b/\#L_c$ strictly increase when the number of variables increases as shown in Tables 4.3 and 4.4. However, no particular pattern of the ratio T_g/T_c (or T_b/T_c) can be observed when the number of hierarchies increases as shown in Table 4.5 (or Table 4.6).

Mean					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'_1	3.20	0.92	20.65	5.27	0.53
P'_2	50.60	0.81	661.46	72.48	0.59
P'_3	289.66	0.79	5518.85	416.12	0.57
P'_4	1114.78	0.75	6676.14	1453.42	0.57
Median					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'_1	1.56	0.92	9.42	2.98	0.50
P'_2	30.09	0.81	219.90	54.99	0.61
P'_3	32.20	0.79	876.56	48.91	0.57
P'_4	130.38	0.74	1583.67	181.71	0.57

Table 4.1: A comparison between S_g and S_c by varying the size of variable domains for *l-b*.

Mean				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'_1	4.55	0.92	11.24	4.51
P'_2	32.98	0.81	266.86	27.98
P'_3	139.04	0.79	408.63	130.7
P'_4	150.00	0.75	524.82	524.82
Median				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'_1	2.52	0.92	7.46	2.63
P'_2	3.87	0.81	50.90	9.68
P'_3	6.21	0.79	45.30	11.02
P'_4	5.76	0.74	65.50	13.45

Table 4.2: A comparison between S_b and S_c by varying the size of variable domains for *l-b*.

Mean					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P''_1	2.09	1.00	13.85	3.76	0.55
P''_2	2.91	1.00	16.80	4.95	0.57
P''_3	4.62	0.92	24.34	8.06	0.55
P''_4	16.06	0.95	58.33	26.86	0.51
Median					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P''_1	2.00	1.00	8.80	2.26	0.50
P''_2	1.81	1.00	10.35	3.32	0.58
P''_3	2.01	0.92	12.00	3.46	0.55
P''_4	4.18	0.92	13.66	7.02	0.51

Table 4.3: A comparison between S_g and S_c by varying the number of variables for *l-b*.

Mean				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P''_1	2.44	1.00	6.74	2.24
P''_2	4.62	1.00	15.12	4.46
P''_3	4.28	0.92	15.73	4.17
P''_4	11.91	0.95	26.76	11.33
Median				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P''_1	1.50	1.00	3.88	1.87
P''_2	2.27	1.00	7.10	2.86
P''_3	1.65	0.92	5.53	2.43
P''_4	3.35	0.92	6.55	4.37

Table 4.4: A comparison between S_b and S_c by varying the number of variables for *l-b*.

Mean					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'''_1	136.46	0.87	1897.50	186.89	0.57
P'''_2	15.54	0.85	198.15	19.38	0.61
P'''_3	339.96	0.88	4521.15	400.76	0.65
P'''_4	140.29	0.88	2706.92	160.78	0.66
Median					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'''_1	17.37	0.87	258.50	21.76	0.57
P'''_2	8.81	0.88	59.74	14.06	0.58
P'''_3	67.14	0.88	657.62	82.68	0.61
P'''_4	67.55	0.88	634.17	50.74	0.66

Table 4.5: A comparison between S_g and S_c by varying the number of hierarchies for *l-b*.

Mean				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'''_1	38.40	0.87	295.72	36.78
P'''_2	11.68	0.85	59.26	10.80
P'''_3	116.67	0.88	624.67	52.43
P'''_4	34.31	0.88	325.14	23.86
Median				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'''_1	10.33	0.87	28.94	10.21
P'''_2	7.46	0.85	35.90	4.77
P'''_3	19.84	0.88	42.38	6.32
P'''_4	18.65	0.88	69.54	10.18

Table 4.6: A comparison between S_b and S_c by varying the number of hierarchies for *l-b*.

Experiments for the *weighted-sum-better* comparator

Experimental results show that the performance of our solver is getting better when the size of the problem instances increases. The mean ratio of T_i/Tc ($i \in \{g, b, r\}$) strictly increases when the size of the problem instances (in terms of the size of variable domains or the number of variables) increases, since the mean ratios of $\#L_i/\#L_c$ and $\#C_i/\#C_c$ also strictly increase as shown in Tables 4.7, 4.8, 4.9, 4.10, 4.11, and 4.12. However, no particular pattern of the mean ratio of T_i/T_c can be observed, except the mean ratio of T_r/T_c strictly increases, when the number of hierarchies increases as shown in Tables 4.13, 4.14, and 4.15. In addition, results show that the mean ratio of T_i/T_c is always larger than the corresponding median ratio for all the cases.

Experimental results show that our solver requires less memories than Lua's solver as shown in Tables 4.9, 4.12, and 4.15, except for the set of CHs in P'_4 as shown in Table 4.9. Experimental results again show that the mean ratio of O_c/T_c is very close to the median ratio of O_c/T_c for *weighted-sum-better.* The mean ratio of O_c/T_c almost remains constant when the size of variable domains increases as shown in Table 4.7, but decreases when the number of variable increases as shown in Table 4.10. However, the mean ratio of O_c/T_c strictly increases when the number of hierarchies increases as shown in Table 4.13. The comparison between $\#F_r$ and $\#F_c$ is listed in Tables 4.9, 4.12, and 4.15 for a more complete comparison. However, we do not use this ratio as a measurement of pruning power.

Experiments for the *worst-case-better* comparator

Experimental results show that the performance of our solver is also getting better when the size of the problem instances increases for *worst-case-better.* The mean ratio of T_i/Tc ($i \in \{g, b, r\}$) strictly increases when the size of the problem

Mean					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'_1	6.86	0.92	59.61	11.04	0.58
P'_2	41.82	0.87	430.34	67.03	0.58
P'_3	602.42	0.83	9363.08	729.50	0.60
P'_4	1233.76	0.76	15644.83	1525.96	0.59
Median					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'_1	3.59	0.92	34.01	5.83	0.59
P'_2	18.30	0.87	226.63	18.96	0.58
P'_3	41.75	0.83	270.17	57.11	0.59
P'_4	120.58	0.77	1880.24	156.08	0.59

Table 4.7: A comparison between S_g and S_c by varying the size of variable domains for *w-s-b*.

Mean				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'_1	5.47	0.92	33.84	8.06
P'_2	28.72	0.87	149.97	23.18
P'_3	264.63	0.83	5524.71	131.24
P'_4	1066.57	0.76	8007.48	1114.50
Median				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'_1	3.42	0.92	17.94	4.23
P'_2	6.74	0.87	26.30	6.53
P'_3	9.48	0.83	128.40	8.46
P'_4	6.27	0.77	41.77	9.96

Table 4.8: A comparison between S_b and S_c by varying the size of variable domains for *w-s-b*.

Mean					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P'_1	4.54	1.26	29.63	5.56	0.02295
P'_2	10.13	1.12	94.56	13.69	0.00116
P'_3	111.75	1.04	5005.39	92.16	0.00115
P'_4	828.31	0.91	7342.58	930.08	0.00020
Median					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P'_1	2.77	1.25	9.94	3.56	0.000368
P'_2	3.04	1.13	20.29	3.46	0.000055
P'_3	5.39	1.06	84.67	5.09	0.000079
P'_4	4.60	0.91	25.46	5.70	0.000031

Table 4.9: A comparison between S_r and S_c by varying the size of variable domains for *w-s-b*.

Mean					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P''_1	1.04	0.99	4.61	1.74	0.62
P''_2	3.63	0.99	17.32	6.13	0.57
P''_3	15.17	0.92	93.54	22.15	0.53
P''_4	17.85	0.94	182.62	27.07	0.53
Median					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P''_1	0.83	1.00	3.13	1.44	0.67
P''_2	2.30	1.00	13.30	3.92	0.57
P''_3	3.84	0.92	22.34	5.15	0.52
P''_4	3.18	0.92	8.02	6.21	0.53

Table 4.10: A comparison between S_g and S_c by varying the number of variables for *w-s-b*.

Mean				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P''_1	1.22	0.99	2.64	1.33
P''_2	3.90	0.99	9.76	3.89
P''_3	12.52	0.92	58.55	12.49
P''_4	15.61	0.94	132.01	16.74
Median				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P''_1	1.17	1.00	2.49	1.19
P''_2	2.83	1.00	5.29	3.35
P''_3	2.64	0.92	8.50	2.59
P''_4	2.37	0.92	5.18	4.24

Table 4.11: A comparison between S_b and S_c by varying the number of variables for *w-s-b*.

Mean					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P''_1	0.72	1.40	1.59	0.88	0.16327
P''_2	1.36	1.33	5.32	1.74	0.01145
P''_3	8.86	1.26	44.67	10.95	0.00085
P''_4	12.58	1.22	118.84	15.92	0.00090
Median					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P''_1	0.67	1.40	1.63	0.60	0.054393
P''_2	1.06	1.36	3.39	1.71	0.006000
P''_3	1.79	1.25	6.79	2.59	0.000240
P''_4	1.51	1.23	3.98	2.97	0.000015

Table 4.12: A comparison between S_r and S_c by varying the number of variables for *w-s-b*.

Mean					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'''_1	124.29	0.87	2566.32	150.96	0.60
P'''_2	26.06	0.86	372.62	33.35	0.65
P'''_3	127.96	0.88	2986.32	136.65	0.67
P'''_4	271.32	0.88	6066.67	276.83	0.71
Median					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'''_1	64.15	0.87	603.77	73.29	0.56
P'''_2	14.16	0.88	154.90	17.65	0.64
P'''_3	41.47	0.88	731.43	36.72	0.66
P'''_4	43.74	0.88	514.80	45.15	0.69

Table 4.13: A comparison between S_g and S_c by varying the number of hierarchies for *w-s-b*.

Mean				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'''_1	16.74	0.87	159.75	11.20
P'''_2	33.31	0.86	137.20	20.28
P'''_3	52.25	0.88	452.63	26.43
P'''_4	34.90	0.88	492.53	36.60
Median				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'''_1	9.45	0.87	42.67	5.11
P'''_2	6.72	0.88	73.04	4.93
P'''_3	9.84	0.88	119.98	17.23
P'''_4	11.10	0.88	149.37	8.93

Table 4.14: A comparison between S_b and S_c by varying the number of hierarchies for *w-s-b*.

Mean					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P'''_1	8.15	1.13	101.70	8.06	0.0016
P'''_2	11.39	1.21	96.75	11.52	0.0043
P'''_3	22.72	1.32	435.72	19.03	0.3527
P'''_4	27.06	1.49	426.83	21.41	0.9804
Median					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P'''_1	5.96	1.13	35.70	4.85	0.00051
P'''_2	3.98	1.19	60.71	3.85	0.00016
P'''_3	8.90	1.31	97.66	8.00	0.00021
P'''_4	5.37	1.44	111.48	4.69	0.00044

Table 4.15: A comparison between S_r and S_c by varying the number of hierarchies for *w-s-b*.

instances (in terms of the size of variable domains or the number of variables) increases, since the corresponding mean ratio of $\#C_i/\#C_c$ also strictly increases as shown in Tables 4.16, 4.17, 4.18, 4.19, 4.20, and 4.21. However, no particular pattern of the mean ratio of T_i/T_c can be observed, except the mean ratio of T_g/T_c strictly increases as shown in Tables 4.22, when the number of hierarchies increases as shown in Tables 4.23, and 4.24.

Experimental results show that our solver requires less memories than Lua's solver as shown in Tables 4.18, 4.21, and 4.24, except for the set of CHs in P'_4 as shown in Table 4.18. The mean ratio of O_c/T_c almost remains constant when the size of variable domains increases as shown in Table 4.16. However, the mean ratio of O_c/T_c increases when the number of variables (or hierarchies) increases as shown in Table 4.19 (or Table 4.22).

Mean					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'_1	7.87	0.92	73.55	13.49	0.53
P'_2	19.64	0.86	189.48	32.80	0.56
P'_3	33.94	0.83	122.76	60.39	0.55
P'_4	363.38	0.76	2802.27	546.22	0.56
Median					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'_1	0.88	0.92	9.09	1.83	0.50
P'_2	6.37	0.87	33.34	9.63	0.55
P'_3	3.23	0.83	12.12	7.46	0.52
P'_4	19.74	0.77	279.57	39.23	0.53

Table 4.16: A comparison between S_g and S_c by varying the size of variable domains for *w-c-b*.

Mean				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'_1	6.22	0.92	37.06	5.92
P'_2	19.81	0.86	179.52	27.79
P'_3	32.27	0.83	115.82	51.78
P'_4	261.77	0.76	1626.16	348.81
Median				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'_1	1.28	0.92	5.92	1.75
P'_2	6.69	0.87	16.70	7.11
P'_3	2.16	0.83	9.94	5.84
P'_4	22.52	0.77	169.07	26.47

Table 4.17: A comparison between S_b and S_c by varying the size of variable domains for *w-c-b*.

Mean					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P'_1	4.41	1.25	35.66	5.59	0.3215
P'_2	19.15	1.12	163.82	24.49	0.0017
P'_3	30.34	1.03	97.69	43.10	0.1334
P'_4	231.96	0.93	1420.50	318.86	0.1330
Median					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P'_1	1.15	1.25	3.11	1.72	0.007203
P'_2	5.32	1.13	9.84	6.09	0.000061
P'_3	1.69	1.06	6.62	3.15	0.000041
P'_4	20.16	0.91	139.40	20.15	0.000029

Table 4.18: A comparison between S_r and S_c by varying the size of variable domains for *w-c-b*.

Mean					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P''_1	1.60	0.98	7.75	2.78	0.46
P''_2	2.92	1.00	13.27	4.95	0.52
P''_3	5.50	0.92	16.89	11.46	0.54
P''_4	18.72	0.94	106.82	27.59	0.54
Median					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P''_1	1.25	1.00	4.17	1.93	0.50
P''_2	0.74	1.00	1.96	1.77	0.53
P''_3	1.64	0.92	4.34	3.48	0.53
P''_4	1.85	0.92	5.73	3.92	0.53

Table 4.19: A comparison between S_g and S_c by varying the number of variables for *w-c-b*.

Mean				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P''_1	3.12	0.98	6.16	2.54
P''_2	3.59	1.00	12.21	4.63
P''_3	6.50	0.92	13.57	10.02
P''_4	21.11	0.94	103.59	26.67
Median				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P''_1	1.56	1.00	3.11	1.41
P''_2	0.97	1.00	1.88	1.67
P''_3	1.58	0.92	3.35	2.97
P''_4	1.78	0.92	5.21	2.91

Table 4.20: A comparison between S_b and S_c by varying the number of variables for *w-c-b*.

Mean					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P''_1	1.69	1.32	5.14	2.01	0.178
P''_2	3.08	1.27	11.76	4.35	0.341
P''_3	4.03	1.23	11.66	8.23	0.075
P''_4	19.72	1.19	102.91	26.05	0.067
Median					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P''_1	1.00	1.30	2.27	1.33	0.0216718
P''_2	0.87	1.27	1.84	1.62	0.0066079
P''_3	1.23	1.25	3.28	2.57	0.0001734
P''_4	1.53	1.15	5.00	2.38	0.0000061

Table 4.21: A comparison between S_r and S_c by varying the number of variables for *w-c-b*.

Mean					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'''_1	13.33	0.87	125.78	21.95	0.58
P'''_2	48.07	0.85	749.52	57.24	0.65
P'''_3	59.36	0.88	485.75	91.43	0.69
P'''_4	78.49	0.88	1510.83	93.41	0.69
Median					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'''_1	5.55	0.87	21.08	12.62	0.53
P'''_2	23.14	0.88	142.58	31.25	0.65
P'''_3	13.59	0.88	170.29	15.82	0.69
P'''_4	19.05	0.88	187.35	22.02	0.68

Table 4.22: A comparison between S_g and S_c by varying the number of hierarchies for *w-c-b*.

Mean				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'''_1	13.53	0.87	108.44	16.30
P'''_2	44.35	0.85	550.01	47.90
P'''_3	58.48	0.88	379.01	68.35
P'''_4	24.43	0.88	355.37	20.29
Median				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'''_1	2.40	0.87	14.18	5.90
P'''_2	19.23	0.88	128.04	21.23
P'''_3	10.86	0.88	80.27	10.87
P'''_4	11.29	0.88	57.55	6.55

Table 4.23: A comparison between S_b and S_c by varying the number of hierarchies for *w-c-b*.

Mean					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P'''_1	11.12	1.12	98.99	14.28	0.134
P'''_2	41.07	1.17	512.50	38.06	0.333
P'''_3	54.68	1.27	307.29	65.37	0.067
P'''_4	19.51	1.38	329.98	16.49	0.067
Median					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P'''_1	2.10	1.13	11.92	3.31	0.0002679
P'''_2	17.37	1.19	116.07	17.13	0.0001983
P'''_3	8.20	1.25	79.21	7.55	0.0000335
P'''_4	4.84	1.37	41.44	4.33	0.0000833

Table 4.24: A comparison between S_r and S_c by varying the number of hierarchies for *w-c-b*.

Experiments for the *least-squares-better* comparator

Experimental results show that the mean ratio of T_g/T_c strictly increases when the size of variables domain increases, since the mean ratio of $\#C_g/\#C_c$ also strictly increases as shown in Table 4.25. Although the mean ratios of $\#L_b/\#L_c$ and $\#C_b/\#C_c$ do not strictly increase, the mean ratio of T_b/T_c also strictly increases as shown in Table 4.26. However, the mean ratio of T_r/T_c does not strictly increase, even the size of variable domains increases as shown in Table 4.27. The mean ratio of T_i/T_c ($i \in \{g, b, r\}$) increase, but not strictly increasing, as the number of the variables increases as shown in Tables 4.28, 4.29, and 4.30. However, no particular pattern for the mean ratio of T_i/T_c when the number of hierarchies increases as shown in Tables 4.31, 4.32, and 4.33. We can only observe that the median ratio of $\#L_r/\#L_c$ increases when comparing our solver to the Lua's solver as shown in Table 4.33.

The memory requirement of our solver is less than that of the Lua's solver for most of the cases as shown in Tables 4.27, 4.30, and 4.33. However, our solver always requires more memories than solvers S_g and S_b. Results show that the mean ratio of O_c/T_c almost remains constant when the size of variable domains increases as shown in Table 4.25. However, the mean ratio of O_c/T_c decreases (increases) when the number of variables (hierarchies) increases as shown in Table 4.28 (Table 4.31).

Mean					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'_1	3.04	0.92	17.02	4.88	0.54
P'_2	32.13	0.87	440.84	45.03	0.57
P'_3	214.58	0.83	7853.79	248.74	0.58
P'_4	385.12	0.75	2618.29	526.99	0.57
Median					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'_1	3.84	0.92	15.17	4.60	0.52
P'_2	11.95	0.87	65.98	18.26	0.55
P'_3	15.94	0.83	58.01	24.58	0.56
P'_4	11.55	0.74	24.77	16.91	0.56

Table 4.25: A comparison between S_g and S_c by varying the size of variable domains for *l-s-b*.

Remarks

Our proposed solver S_c is faster than the Generate-and-Test solver S_g, the basic Branch-and-Bound solver S_b, and the Lua's solver S_r for most of the time. For example, S_c is faster than S_g, S_b, and S_r respectively 88%, 79%, and 73% of the time for *weighted-sum-better* as shown in Table 4.34. Since the CHAC algorithm incurs overhead in the branch-and-bound search, this is the main reason for S_c to be slower than others for some problem instances. The inferiority of solver S_c for "small" problems, in terms of the size of variable domains and the number of variables, is expected since the pruning obtained is outweighed by the overhead. For the larger problems the extra effort paid by the CHAC algorithm at each search node is demonstrated worthwhile. This result is in line with the behav-

Mean				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'_1	3.33	0.92	13.11	3.03
P'_2	11.54	0.87	64.09	11.08
P'_3	13.32	0.83	148.52	34.15
P'_4	25.17	0.75	72.86	32.74
Median				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'_1	1.79	0.92	6.85	2.77
P'_2	2.71	0.87	15.46	2.66
P'_3	2.08	0.83	13.32	6.61
P'_4	8.21	0.74	21.64	5.38

Table 4.26: A comparison between S_b and S_c by varying the size of variable domains for *l-s-b*.

Mean					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P'_1	1.99	1.26	8.10	2.68	0.02573
P'_2	5.53	1.10	45.11	7.41	0.00134
P'_3	12.38	1.01	128.78	15.60	0.00181
P'_4	10.21	0.91	66.82	13.01	0.00015
Median					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P'_1	1.61	1.25	6.60	1.89	0.0011610
P'_2	0.98	1.07	8.17	1.09	0.0001663
P'_3	1.87	1.00	12.91	3.50	0.0000173
P'_4	3.90	0.87	19.23	4.97	0.0000043

Table 4.27: A comparison between S_r and S_c by varying the size of variable domains for *l-s-b*.

Mean					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P''_1	1.12	1.00	6.03	2.24	0.64
P''_2	2.51	1.00	10.77	4.02	0.55
P''_3	17.79	0.92	86.07	29.22	0.54
P''_4	11.11	0.95	45.23	18.94	0.54
Median					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P''_1	1.00	1.00	4.11	2.05	0.60
P''_2	1.45	1.00	4.87	2.73	0.52
P''_3	4.68	0.92	15.52	10.42	0.54
P''_4	4.44	0.92	20.29	7.99	0.53

Table 4.28: A comparison between S_g and S_c by varying the number of variables for *l-s-b*.

Mean				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P''_1	2.32	1.00	2.75	1.25
P''_2	2.21	1.00	6.37	2.96
P''_3	9.16	0.92	19.39	10.36
P''_4	6.03	0.95	21.58	8.63
Median				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P''_1	1.14	1.00	2.30	1.16
P''_2	1.37	1.00	2.73	1.83
P''_3	4.09	0.92	5.49	5.45
P''_4	1.62	0.92	4.53	4.25

Table 4.29: A comparison between S_b and S_c by varying the number of variables for *l-s-b*.

Mean					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P''_1	0.77	1.38	2.55	0.99	0.11417
P''_2	1.23	1.28	5.13	1.82	0.07130
P''_3	5.37	1.24	16.80	9.11	0.00507
P''_4	4.43	1.20	15.65	7.42	0.00016
Median					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P''_1	0.67	1.40	1.76	0.79	0.070588
P''_2	0.69	1.27	1.84	1.00	0.005418
P''_3	1.47	1.25	5.28	2.14	0.000753
P''_4	1.32	1.23	3.47	2.59	0.000013

Table 4.30: A comparison between S_r and S_c by varying the number of variables for *l-s-b*.

Mean					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'''_1	347.25	0.86	11461.62	414.69	0.59
P'''_2	15.61	0.85	229.15	18.94	0.62
P'''_3	77.70	0.88	1820.66	81.70	0.69
P'''_4	146.77	0.88	4133.90	161.91	0.72
Median					
CHs	T_g/T_c	M_g/M_c	$\#L_g/\#L_c$	$\#C_g/\#C_c$	O_c/T_c
P'''_1	24.62	0.87	206.87	24.22	0.60
P'''_2	6.79	0.88	156.64	9.40	0.61
P'''_3	24.46	0.88	799.80	27.68	0.70
P'''_4	39.70	0.88	493.45	32.55	0.72

Table 4.31: A comparison between S_g and S_c by varying the number of hierarchies for *l-s-b*.

Mean				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'''_1	57.91	0.86	283.60	41.17
P'''_2	9.31	0.85	62.88	8.22
P'''_3	50.84	0.88	899.27	40.77
P'''_4	27.72	0.88	223.75	8.05
Median				
CHs	T_b/T_c	M_b/M_c	$\#L_b/\#L_c$	$\#C_b/\#C_c$
P'''_1	13.34	0.87	28.75	10.93
P'''_2	5.50	0.88	28.08	5.78
P'''_3	18.05	0.88	29.01	4.51
P'''_4	16.99	0.88	63.01	7.07

Table 4.32: A comparison between S_b and S_c by varying the number of hierarchies for *l-s-b*.

Mean					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P'''_1	31.51	1.12	204.87	37.87	0.11031
P'''_2	7.27	1.17	48.87	7.65	0.00132
P'''_3	37.11	1.27	754.02	34.57	0.00797
P'''_4	8.55	1.37	179.29	7.62	0.00071
Median					
CHs	T_r/T_c	M_r/M_c	$\#L_r/\#L_c$	$\#C_r/\#C_c$	$\#F_r/\#F_c$
P'''_1	3.51	1.13	21.95	3.27	0.000378
P'''_2	2.37	1.19	24.15	2.06	0.000074
P'''_3	5.18	1.25	26.67	3.93	0.000457
P'''_4	2.25	1.37	44.90	2.16	0.000183

Table 4.33: A comparison between S_r and S_c by varying the number of hierarchies for *l-s-b*.

ior of embedding classical consistency techniques in basic tree search in solving classical CSPs.

Since the consistency information (or error indicators) may be recomputed for the tradeoff between time and space. The experimental results show that our implementation of S_c requires more memories than S_r at most 14% of the time for global comparators as shown in Table 4.34. Results also show that our solver always requires more memories than S_g or S_b. However, recomputation of consistency information incurs a larger proportion of overhead. At least 50% of the execution time is used to calculate consistency information in S_c on average.

Experimental results show that the number of leaf nodes visited by solver S_c is less than that of S_g, S_b, and S_r at least 91% of the time for *worst-case-better*. The number of choice points visited by solver S_c is also less than that of S_g, S_b, and S_r at least 89% (for *w-c-b*), 78% (for *l-s-b*), and 72% (for *l-s-b*) of the time respectively. S_r relies on classical constraint propagation to enforce the semantics and the operations of the comparators via reified constraints. While the approach, based on existing technology, is clever and clean, the pruning power of reified constraints is relatively weak. On the other hand, S_c executes a dedicated algorithm for maintaining CHAC to help pruning and solution filtering, thus attaining a higher efficiency. Experimental results show that our solver can prune better than Lua's reified constraint approach in practice. Therefore, the execution time of our solver can be further reduced if the implementation of the CHAC algorithm is optimized.

The detail comparisons for each comparator, in terms of the mean and median ratios, are shown in Tables 4.35 and 4.36. The ratios for each comparator are calculated by using 180 problem instances. Results show that the mean is larger than the median in terms of the time, the number of leaf nodes visited, and the number of choice points in the search. This implies that our solver can perform extremely well for some problem instances. Our solver is efficient in terms of the execution time, since it visits less number of leaf nodes and number of choice points than Lua's solver on average. Besides, it requires less memory than Lua's solver. However, at least 50% of the execution time is used to calculate consistency information by the CHAC algorithm on average.

Comparator	Time	Memory	Leaf Node	Choice Point
Comparison between S_g and S_c				
l-b	94%	100%	100%	98%
w-s-b	88%	100%	100%	94%
w-c-b	73%	100%	91%	89%
l-s-b	83%	100%	99%	94%
Comparison between S_b and S_c				
l-b	79%	100%	93%	79%
w-s-b	79%	100%	97%	84%
w-c-b	73%	100%	91%	89%
l-s-b	66%	100%	96%	78%
Comparison between S_r and S_c				
w-s-b	73%	11%	97%	81%
w-c-b	71%	12%	91%	88%
l-s-b	63%	14%	95%	72%

Table 4.34: A summary of the performance of S_c.

4.4.3 The Second Experiment

We randomly generate 180 problem instances (3 experiments × 4 sets of CHs × 15 instances) as the benchmark problems for the second experiment. However, we only focus on the execution time in this experiment. Similar to the first experiment, we record the results in three different ways: varying the size of variable domains (P'_1, P'_2, P'_3, and P'_4), varying the number of variables (P''_1, P''_2, P''_3, and P''_4), and varying the number of hierarchies (P'''_1, P'''_2, P'''_3, and P'''_4). We benchmark the performance of our solver on different comparators (*locally-better*, *weighted-sum-better*, *worst-case-better*, and *least-squares-better*)

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	0.88	1860	230
w-s-b	206	0.89	3149	249
w-c-b	54	0.89	517	80
l-s-b	105	0.89	2394	130
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	0.88	215	70
w-s-b	128	0.89	1263	117
w-c-b	41	0.89	291	53
l-s-b	18	0.89	152	17
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	1.22	1142	94
w-c-b	37	1.19	258	47
l-s-b	11	1.19	123	12

Table 4.35: A summary of the mean performance of S_c.

Median				
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	9	0.88	70	13
w-s-b	9	0.88	86	13
w-c-b	4	0.88	22	7
l-s-b	5	0.88	33	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	7	0.88	31	7
w-s-b	6	0.88	22	6
w-c-b	3	0.88	13	5
l-s-b	3	0.88	16	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	3	1.25	18	3
w-c-b	3	1.23	11	3
l-s-b	2	1.25	11	2

Table 4.36: A summary of the median performance of S_c.

using same problem instances. Therefore, we can fairly compare the performance of our solver on different comparators.

Varying the Size of Variable Domains

We randomly generate 60 problem instances (4 sets of CHs $\times$ 15 instances) to benchmark the performance of our solver when the size of variable domains increases. We compare the performance between the solver S_g and our solver S_c first as shown in Table 4.37. Experimental results show that the mean ratios of T_g/T_c on *weighted-sum-better*, *worst-case-better*, and *least-squares-better* strictly increase when the size of variable domains increases. Results also show that the mean and median ratios of T_g/T_c on *weighted-sum-better* and *least-squares-better* are very close. The mean and median ratios of T_g/T_c on the *worst-case-better* comparator are almost the smallest. This implies the performance of our solver on the *worst-case-better* comparator is the worst.

	T_g/T_c (Mean)				T_g/T_c (Median)			
CHs	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*
P'_1	10	8	5	7	4	4	1.5	4
P'_2	13	36	15	37	3	7	0.6	7
P'_3	171	267	67	261	22	74	6	72
P'_4	76	385	72	342	12	13	5	13

Table 4.37: A comparison between S_g and S_c by varying the size of variable domains.

When comparing the performance of our solver to the basic Branch-and-Bound solver S_b and Lua's solver S_r, we can only observe that the mean and median ratios of T_b/T_c (or T_r/T_c) on *weighted-sum-better* and *least-squares-better* are very close as shown in Tables 4.38 and 4.39.

	T_b/T_c (Mean)				T_b/T_c (Median)			
CHs	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*
P'_1	7	6	4	6	2	2	1.3	2
P'_2	9	18	22	19	1.3	6	0.6	6
P'_3	31	121	47	123	4	2	5	2
P'_4	23	37	35	39	3	5	3	5

Table 4.38: A comparison between S_b and S_c by varying the size of variable domains.

	T_r/T_c (Mean)			T_r/T_c (Median)		
CHs	*w-s-b*	*w-c-b*	*l-s-b*	*w-s-b*	*w-c-b*	*l-s-b*
P'_1	5	4	5	1	0.8	0.9
P'_2	9	19	9	5	0.6	5
P'_3	113	42	115	0.9	5	0.9
P'_4	17	27	18	2	2	2

Table 4.39: A comparison between S_r and S_c by varying the size of variable domains.

Varying the Number of Variables

We also randomly generate 60 problem instances (4 sets of CHs $\times$ 15 instances) to benchmark the performance of our solver on different comparators when the number of variables increases. When comparing the performance between the solver S_g and our solver S_c as shown in Table 4.40, the results show that the mean ratios of T_g/T_c on *weighted-sum-better*, *worst-case-better*, and *least-squares-better* increase if the problem size increases in terms of the number of variables. The mean and median ratios of T_g/T_c on *weighted-sum-better* and *least-squares-better* are very close. Again, the performance of our solver for solving the *worst-case-better* comparator is the worst.

For the comparison between the basic Branch-and-Bound solver S_b and our solver, the results show that the mean and median ratios of T_b/T_c on *weighted-*

	T_g/T_c (Mean)				T_g/T_c (Median)			
CHs	*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	1.2	0.9	1.3	1	0.8	0.6	1
P''_2	5	6	3	6	2	3	1.4	3
P''_3	4	7	3	7	2	3	1.5	2
P''_4	26	24	8	24	2	4	1.4	3

Table 4.40: A comparison between S_g and S_c by varying the number of variables.

sum-better and *least-squares-better* are very close as shown in Table 4.41. The results, in particular for the median ratio, also show that our solver performs the worst for the *worst-case-better* comparator.

	T_b/T_c (Mean)				T_b/T_c (Median)			
CHs	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*
P''_1	1.4	1.2	1.3	1.5	1.3	1.3	1	1.2
P''_2	4	5	3	5	1.5	2	1.4	2
P''_3	3	5	4	5	1.4	2	1.4	2
P''_4	5	3	7	3	1.4	3	0.7	3

Table 4.41: A comparison between S_b and S_c by varying the number of variables.

In the comparison between Lua's solver S_r and our solver, we can only observe that the mean and median ratios of T_r/T_c on *weighted-sum-better* and *least-squares-better* are very close as shown in Table 4.42. The results are obtained by increasing the number of variables in this experiment, but we obtain results by increasing the size of variable domains in previous experiment. We obtain similar results in both experiments, since the number of hierarchies is kept constant.

Varying the Number of Hierarchies

Lastly, we benchmark the performance of our solver on different comparators when the number of hierarchies increases. Experimental results show that the

	T_r/T_c (Mean)			T_r/T_c (Median)		
CHs	*w-s-b*	*w-c-b*	*l-s-b*	*w-s-b*	*w-c-b*	*l-s-b*
P''_1	1.1	1.1	1.4	1.2	0.8	1
P''_2	5	3	5	1.4	1.4	1.3
P''_3	4	4	4	2	1.4	1.7
P''_4	1.4	6	1.4	1	0.6	1

Table 4.42: A comparison between S_r and S_c by varying the number of variables.

mean and median ratios, corresponding to T_g/T_c, T_b/T_c, and T_r/T_c, on *weighted-sum-better* and *least-squares-better* are very close as show in Tables 4.43, 4.44, and 4.45. These are similar results as those obtained in previous experiments. However, the results show that our solver can perform very well, in particular for the *worst-case-better* comparator, when problem instances with more constraints. The mean ratio of T_b/T_c (or T_r/T_c) on the *worst-case-better* comparator strictly increases when the number of hierarchies increases. Results show that the performance of our solver on the *worst-case-better* comparator is the worst only when problem instances consist "less" constraints.

	T_g/T_c (Mean)				T_g/T_c (Median)			
CHs	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*
P'''_1	122	146	108	151	27	24	6	26
P'''_2	116	209	130	212	29	52	12	54
P'''_3	50	232	168	219	10	63	29	70
P'''_4	75	122	154	124	15	44	52	47

Table 4.43: A comparison between S_g and S_c by varying the number of hierarchies.

Remarks

Experiment results show that the performances of our solver on *weighted-sum-better* and *least-squares-better* are very close. However, the performance of our

	T_b/T_c (Mean)				T_b/T_c (Median)			
CHs	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*	*l-b*	*w-s-b*	*w-c-b*	*l-s-b*
P'''_1	32	44	44	44	4	3	4	3
P'''_2	34	51	116	50	5	6	10	6
P'''_3	21	42	121	44	3	6	8	5
P'''_4	26	58	132	60	4	11	53	11

Table 4.44: A comparison between S_b and S_c by varying the number of hierarchies.

	T_r/T_c (Mean)			T_r/T_c (Median)		
CHs	*w-s-b*	*w-c-b*	*l-s-b*	*w-s-b*	*w-c-b*	*l-s-b*
P'''_1	37	39	39	2	4	2
P'''_2	38	104	39	6	9	6
P'''_3	31	113	29	5	6	5
P'''_4	51	128	52	9	51	9

Table 4.45: A comparison between S_r and S_c by varying the number of hierarchies.

solver on the *worst-case-better* comparator is the worst for most of the cases. The aggregated error value in each level of a valuation is calculated by **max** in *worst-case-better*, but $\sum$ in *weighted-sum-better* (or *least-squares-better*). It is possible that a valuation θ is *weighted-sum-better* than another valuation σ, but they may be incomparable if *worst-case-better* is used. For example, there are three constraints in a particular hierarchy level H_i. The error values in level i of valuations θ and σ are $\{10, 0, 0\}$ and $\{10, 9, 8\}$ respectively. It is clear that we cannot compare valuations θ and σ at level i using *worst-case-better*, since they have the same aggregated error value 10 at this level. However, valuation θ is *weighted-sum-better* than valuation σ, since they have different aggregated error values 10 and 27 corresponding to θ and σ at this level. The same argument is also applicable to the *least-squares-better* and the *locally-better* comparators. No pruning can be produced if errors are incomparable. Thus, the performance on the *worst-case-better* comparator is low for our solver as well as Lua's solver.

Given a problem instance with more number of constraints, the performance of Lua's solver, comparing to our solver, on the *worst-case-better* comparator is relatively low. The main reason is the requirement of extra variables. The extra variables include variables associated with the reified constraints, as well as the error combining constraints. This implies that Lua's solver searches a larger search tree than our solver. Furthermore, the error combining constraint is implemented using the **IlcMax** constraint in ILOG Solver 4.4, which is again weak in propagation.

Chapter 5

Concluding Remarks

5.1 Summary and Contributions

We have discussed the deficiencies of existing techniques [23, 39, 8, 36] in solving finite domain CH in the thesis. DeltaStar [23] fails to solve even small problems in practice [21]. Current status of clp(FD, S) cannot solve any practical finite domain CH [28], because there exists a limitation of the size of the set of semiring values. Lua's solver [36] is based on combining reified constraint propagation and the ordinary Branch-and-Bound algorithm [42]. Our experiments have shown that reified constraint propagation is a relatively weak propagation. We propose to adopt the general notion of local consistency in SCSP [9] to finite domain CH. Since this general notion of local consistency can handle soft constraints, it is possible to adopt it to CH. This idea is realized by reformulating CH with the notion of error indicators. An error indicator is defined as useful consistency information to indicate the "goodness" of values in variable domains. We define constraint hierarchy k-consistency (CH-k-C) in finite domain CH, which is based on error indicators. Since arc-consistency algorithm is a common technique to detect local inconsistency in classical CSPs [6, 30], we design and implement an algorithm to enforce CH-2-C, which we also called constraint hierarchy arc-consistency (CHAC). The CHAC algorithm is derived and it is used to transform a finite domain CH from arc-inconsistent to arc-consistent. We propose to combine the CHAC algorithm and the ordinary Branch-and-Bound algorithm to facilitate pruning in a search tree. This technique is implemented in the Branch-and-Bound CHAC solver. Our large-scale experiments show that our proposed solver is practical and produce more pruning than Lua's solver.

The contribution of our work is two-fold. First, we reformulate CH with the notion of error indicators. By using error indicators and adopting the general notion of local consistency in SCSP, we define k-consistency in finite domain CH. We derive the CHAC algorithm, which maintains a finite domain CH to be arc-consistent by performing constraint check of unary and binary constraints. The correctness of the CHAC algorithm is established.

Second, we show how the CHAC algorithm can be incorporated into the ordinary Branch-and-Bound algorithm to provide a general finite domain CH solver (Branch-and-Bound CHAC solver). Our solver can find solutions with respect to *locally-better*, *weighted-sum-better*, *worst-case-better*, and *least-squares-better* comparators. In addition, our solver is designed to handle arbitrary error functions. Pruning is realized in our solver by invoking CHAC algorithm in each step of traversing a search tree and comparing the current bound [42] with error indicators. The correctness of the Branch-and-Bound CHAC solver is also established. Our experiments confirm the efficiency, pruning power, and robustness of our solver, which brings us one step towards practical finite domain CH solving.

5.2 Future Work

There is room for future research. First, we realize that inconsistent values in variable domains can be removed in Lua's reified constraint approach. Our experiments show that pruning heavily relies on removing inconsistent values from variable domains in Lua's solver. A relatively small proportion of pruning relies on comparing the bound in Branch-and-Bound algorithm. It would be interesting to study how to integrate reified constraint propagation and CHAC algorithm to produce more pruning.

Second, finite domain CH is an optimization problem. The efficiency of the Branch-and-Bound algorithm can be sensitive to variable and value orderings. If the "best" solutions are located nearer to the left hand side of a search tree, then the Branch-and-Bound algorithm can find the "best" solutions in an earlier time. This implies more pruning can be produced. Therefore, it is worthwhile to investigate good ordering heuristics.

Concerning implementations, our implementation and even the CHAC algorithm are hardly optimized. Experiments show that our solver spends more than 50% of the execution time to enforce CHAC algorithm. It is impractical to store all consistency information during searching, as too many memories are required. Hence, some consistency information is recomputed resulting in such a large overhead. Our experiments show that the current implementation of our solver requires fewer memories with respect to Lua's solver. It would be interesting to study a way to make the tradeoff between time and space in order to reduce the overhead of the CHAC algorithm.

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 regional comparator [51], *regionally-better* comparator. The existing comparators, although rigorously and mathematically defined, might be too general for a specific real-life situation [16]. It would be interesting to allow user-defined comparators, tailored for individual situations. It would be interesting to introduce new comparators that should be of particular relevance to real-life problems and applicable to our solver.

Last but not the least, in order to establish the practicality of our work, we need to experiment on more structured problems and real-life problems as we test our solver only on random problems. It would be also interesting to study whether our solver can have better pruning when applying metric error function.

Bibliography

[1] B. Abramson and M. Yung. Divide and conquer under global constraints: A solution to the n-queens problem. *Journal of Parallel and Distributed Computing*, 6:649–662, 1989.

[2] 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.

[3] R. Barták. A plug-in architecture of constraint hierarchy solvers. In *Proceedings of PACT97*, pages 359–371, 1997.

[4] R. Barták. Benchmark for finite domain constraint hierarchies. Private Communication, May 2002.

[5] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.

[6] C. Bessière, E.C. Freuder, and J.C. Régin. Using inference to reduce arc consistency computation. In *Proceedings of IJCAI95*, pages 592–598, 1995.

[7] S. Bistarelli, P. Codognet, Y. Georget, and F. Rossi. Labeling and partial local consistency for soft constraint programming. In *Proceedings of the 2nd International Workshop on Practical Aspects of Declarative Languages*, pages 230–248, 2000.

[8] S. Bistarelli, Y. Georget, and J.H.M. Lee. Capturing (fuzzy) constraint hierarchies in semiring-based constraint satisfaction. Unpublished Manuscript, 1999.

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

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

[11] 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.

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

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

[14] 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.

[15] P. Codognet. Benchmark for finite domain constraint hierarchies. Private Communication, May 2002.

[16] P. Codognet. Semantic of the existing comparators. Private Communication, July 2002.

[17] P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3), 1996.

[18] P. Codognet and D. Diaz. Yet another adaptive search method for constraint solving. In *The 1st Symposium on Stochastic Algorithms, Foundations and Applications*, 2001.

[19] B. Freeman-Benson. Converting an existing user interface to use constraints. In *Proceedings of the 1993 ACM Symposium on User Interface Software and Technology*, pages 207–215, 1993.

[20] B. Freeman-Benson. Benchmark for finite domain constraint hierarchies. Private Communication, May 2002.

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

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

[23] 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.

[24] E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 12:958–966, 1978.

[25] E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[26] P. Galinier and J.K. Hao. A general approach for constraint solving by local search. In *Proceedings of the 2nd International Workshop on Integration*

of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pages 57–69, 2000.

[27] J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. In *Proceedings of IJCAI77*, page 457, 1977.

[28] Y Georget. Using clp(FD,S) for solving finite domain constraint hierarchies. Private Communication, Feb 2002.

[29] Y. Georget and P. Codognet. Compiling semiring-based constraints with clp(FD,S). In *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming*, 1998.

[30] S.A. Grant and B.M. Smith. The phase transition behavior of maintaining arc consistency. In *Proceedings of ECAI96*, pages 175–179, 1996.

[31] P.V. Hentenryck. Tutorial on the CHIP systems and applications. In *Workshop of Constraint Logic Programming*, 1988.

[32] P.V. Hentenryck, Y. Deville, and C. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[33] 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.

[34] H. Hosobe, K. Miyashita, S. Takahashi, S. Matsuoka, and A. Yonezawa. Locally simultaneous constraint satisfaction. In *Proceedings of PPCP94*, pages 51–62, 1994.

[35] ILOG. *ILOG Solver 4.4 Reference Manual*, 1999.

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

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

[38] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

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

[40] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[41] B.A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[42] P.M. Narendra and K. Fukunaga. A branch and bound algorithm for feature subset selection. *IEEE Transactions on Computers*, 26(9):917–922, 1977.

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

[44] Zs. Ruttkay. Fuzzy constraint satisfaction. In *Proceedings of the 3rd IEEE International Conference on Fuzzy Systems*, pages 1263–1268, 1994.

[45] M. Sannella. SkyBlue: A multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 1994 ACM Symposium on User Interface Software and Technology*, pages 137–146, 1994.

[46] 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.

[47] T. Schiex. Possibilistic constraint satisfaction problems or "how to handle soft constraints?". In *Proceedings of the 8th International Conference on Uncertainty in Artificial Intelligence*, pages 269–275, 1992.

[48] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: hard and easy problems. In *Proceedings of IJCAI95*, pages 631–637, 1995.

[49] SICStus. *SICStus Prolog User's Manual*, 2002.

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

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

[52] A. Wolf. Benchmark for finite domain constraint hierarchies. Private Communication, May 2002.