INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Global Optimization Using Interval Constraints

by

Huaimo Chen
B.Sc., Shenyang Institute of Aeronautical Technology, 1982
M.Sc., Changsha Institute of Technology, 1987

A Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. M.H. van Emden, Supervisor (Department of Computer Science)

Dr. M.H.M. Cheng, Department Member (Department of Computer Science)

Dr. M.R. Levy, Department Member (Department of Computer Science)

Dr. Z. Dong, Outside Member (Department of Mechanical Engineering)

Dr. T. Hickey, External Examiner (Department of Computer Science, Brandeis University)

Supervisor: Dr. M.H. van Emden

Abstract

Global optimization methods can be classified into two non-overlapping classes with respect to accuracy: those with guaranteed accuracy and those without. The former are called *bounding methods*, the latter *point methods*. Bounding methods compute lower and upper bounds of function over a box and give a lower bound and an upper bound for the minimum. Point methods compute function values at points and output as the minimum the function value at a point.

R. E. Moore was the first to propose the bounding method using interval arithmetic for unconstrained global optimization. The first bounding method using interval arithmetic for constrained global optimization was due to E. R. Hansen and S. Sengupta. These methods are the well known bounding methods. Since these methods use interval arithmetic, we call them interval arithmetic methods. This dissertation studies the new bounding methods that use interval constraints, which is called interval constraint methods.

We prove that interval constraints is a generalization of interval arithmetic, computing an interval function in interval constraints gives the same result as in interval arithmetic. We propose a hypernarrowing algorithm using interval constraints. This algorithm produces a smaller interval result for the range of function f over a given domain than interval arithmetic. We present a generic Branch-and-Bound algorithm for unconstrained global optimization, prove the properties of the algorithm, and propose improvements on the algorithm. From this algorithm, we can obtain its interval arithmetic version and interval constraint version. We investigate the role of interval

constraints in global optimization and discuss the performance and characteristics of interval arithmetic methods and interval constraint ones.

Based on the Branch-and-Bound algorithm for unconstrained global optimization, we present a generic Branch-and-Bound algorithm for constrained global optimization, study the effect of Fritz-John conditions as redundant constraints and compare the interval arithmetic method for constrained optimization with the interval constraint one.

Examiners:

Dr. M.H. van Emden, Supervisor (Department of Computer Science)

Dr. M.H.M. Cheng, Department Member (Department of Computer Science)

Dr. M.R. Levy, Department Member (Department of Computer Science)

Dr. Z. Dong, Outside Member (Department of Mechanical Engineering)

Dr. T. Hickey, External Examiner (Department of Computer Science, Brandeis University)

Contents

List of Symbols

$f(x)$	objective function.
$p_i(x)$	inequality constraint functions, $i = 1, \ldots, m$.
$q_j(x)$	equality constraint functions, $j = 1, \ldots, r$.
f^*	the global minimum of the function f subject to $p_i(x) \leq 0$ $(i = 1, \ldots, m)$ and $q_j(x) = 0$ $(j = 1, \ldots, r)$, i.e., $f^* = \min \{f(x) \mid p_i(x) \leq 0 \ (i = 1, \ldots, m), q_j(x) = 0 \ (j = 1, \ldots, r)\}$.
x^*	a global minimizer, $f(x^*) = f^*$.
f^*_{ub}	the lowest upper bound of the global minimum f^* obtained so far.
R	the set of real numbers.
$\mathcal{F}$	a finite subset of R. Typically, $\mathcal{F}$ is the set of floating-point numbers of a computer.
$I(\mathcal{F})$	the set of floating-point intervals, i.e., the set of all $[a, b]$, where $a, b \in \mathcal{F}$.
$\mathcal{F}_n$	$\mathcal{F}_n = \{I_1 \times \cdots \times I_n \mid I_i \in I(\mathcal{F}) \text{ for } i = 1, \ldots, n\}$.
B	floating-point box $B \in \mathcal{F}_n$.
X	floating-point box $X \in \mathcal{F}_n$.
$\square f(X)$	$\square f(X)$ is the smallest interval containing the range of f over X.
lb	lower bound function $lb([a, b]) = a$ for any $[a, b] \in I(\mathcal{F})$.

ub upper bound function $ub([a,b]) = b$ for any $[a,b] \in I(\mathcal{F})$.

$|B|$ the width of box B. For any $B = [a,b]$, $|B| = b - a$;
for any $B = X_1 \times \cdots \times X_n$, $|B| = \max\{|X_i| \mid i = 1, \ldots, n\}$.

$\mathrm{mid}(B)$ the midpoint of box B. For any $B = X_1 \times \cdots \times X_n$,
$\mathrm{mid}(B) = (c_1, \ldots, c_n)$, where c_i is the midpoint of the interval X_i.

$\pi_i(r)$ the i-th projection of relation r. For any n-ary relation $r \subset R^n$,
the i-th projection of r
$\pi_i(r) = \{x_i \in R \mid \exists x_1 \cdots \exists x_{i-1} \exists x_{i+1} \cdots \exists x_n$ such that $(x_1, \ldots, x_n) \in r\}$.

$g_i(x)$ $g_i(x) = \partial f(x)/\partial x_i (i = 1, \ldots, n)$, i.e., the i-th projection of gradient of f.

$\nabla f(x)$ $\nabla f(x) = (g_1(x), \ldots, g_n(x))^T$, i.e., the gradient of f.

$h_{ij}(x)$ $h_{ij}(x) = \partial^2 f(x)/\partial x_i \partial x_j \ \ (i,j = 1, \ldots, n)$, i.e., the element of Hessian.

$J_{ij}(x,X)$ $J_{ij}(x,X) = h_{ij}(X_1, \ldots, X_j, x_{j+1}, \ldots, x_n)$ $(i,j = 1, \ldots, n)$,
i.e., the element of Jacobian.

$J(x,X)$ the Jacobian of f, i.e., $J(x,X) = [J_{ij}(x,X)], \ \ i,j = 1, \ldots, n$.

BBUGO the Branch-and-Bound algorithm for Unconstrained Global Optimization.

BBCGO the Branch-and-Bound algorithm for Constrained Global Optimization.

IAU the Interval Arithmetic version of the Branch-and-Bound algorithm
for Unconstrained Global Optimization.

ICU the Interval Constraint version of the Branch-and-Bound algorithm
for Unconstrained Global Optimization.

$f_b(B)$ $f_b(B) = \begin{cases} ub(f(c)) & \text{if no equalities and} \\ & \quad c \in B, ub(p_i(c)) \leq 0 \quad \text{for all } i = 1, \ldots, m \\ ub(f(B)) & \text{if equalities exist and B contains a feasible point} \\ +\infty & \text{otherwise.} \end{cases}$

IAC the Interval Arithmetic version of the Branch-and-Bound algorithm for Constrained Global Optimization.

ICC the Interval Constraint version of the Branch-and-Bound algorithm for Constrained Global Optimization.

List of Tables

List of Figures

Acknowledgements

First and foremost, I would like to express my deep appreciation to my supervisor, Professor Maarten van Emden, who guided me with extreme patience during the course of my study. He introduced me to the beautiful world of constraint logic programming and global optimization. His insights and careful guidance has made this dissertation possible. Last but not least, Maarten provided me with adequate financial support when I needed money most.

I would like to thank Professors T. Hickey, Z. Dong, M.H.M. Cheng and M.R. Levy for their conscientious reading of the dissertation, their comments and suggestions.

I would also like to thank all members of the Department of Computer Science at University of Victoria for the friendly and pleasant working environment, especially members of logical and functional programming group in the department.

Chapter 1

Introduction

1.1 Motivation

Global optimization is concerned with the determination of the global optimum (maximum or minimum) of a function. Many practical engineering applications can be formulated as global optimization problems, which are difficult to solve. For quite a long time, it has been held that no numerical method could guarantee a global optimum to a general nonlinear global optimization problem [81].

Interval arithmetic provides an upper bound and a lower bound for the range of the values of a function over a given domain [64, 61, 74, 36]. The global optimization methods using interval arithmetic can produce an upper bound and a lower bound for the global optimum of a given function [61, 78, 34, 35, 36], which are the well known methods that produce results with guaranteed accuracy [81]. These methods are called interval arithmetic methods. However, they are not efficient enough for practical uses.

This dissertation will study a new global optimization method which uses interval constraints. This new method inherits the advantages of the interval arithmetic methods. More importantly, it is more declarative and faster than interval arithmetic methods.

1.2 Global Optimization

Definition 1.1:

The global optimization problem is defined as finding

$$f^* = \min \{ f(x) \mid$$

$$p_i(x) \leq 0 \ (i = 1, \ldots, m) \ \text{and} \tag{1.1}$$

$$q_j(x) = 0 \ (j = 1, \ldots, r) \tag{1.2}$$

$$\}$$

where f, p_i $(i = 1, \ldots, m)$ and q_j $(j = 1, \ldots, r)$ are scalar functions of a vector x of n components and f^* is the global minimum.

Without loss of generality, we restrict our attention to minimization. Since minimizing $f(x)$ is equivalent to maximizing $-f(x)$, we can find the maximum for $f(x)$ by finding the minimum of $-f(x)$. If $m = 0$ and $r = 0$, we get an unconstrained global optimization problem; otherwise, we get a constrained one.

The global optimization is to find the global minimum f^* as well as global minimizers or locations at which this minimum value occurs. Formally, we say that a

point x^* is a global minimizer if

$$f(x^*) = f^*$$

and x^* satisfying the constraints in 1.1 and 1.2.

Contrary to the global optimization, local optimization is to find local minima and local minimizers. A point x^* is a local minimizer if there exists a number $\epsilon > 0$ such that

$$f(x^*) \leq f(x)$$

for all x satisfying

$$||x - x^*|| < \epsilon$$

and the constraints in 1.1 and 1.2; where $x - x^*$ is the result of vector subtraction, the *norm* $||v||$ of a vector v with n components is a measure of the size of v [28, 81]. One common norm is defined as

$$||v|| = \max_{1 \leq i \leq n} |v_i|.$$

Rinnooy Kan and Timmer [45] claim that the global optimization problem as stated above is unsolvable in a finite number of steps. However, for a given tolerance $\epsilon > 0$, one can, in a finite amount of time, find a point x' such that the difference between $f(x')$ and the global optimum is within ϵ [62]. Such a point x' is called an ϵ-global minimizer. Formally, an ϵ-global minimizer is defined as a point, x', such that

$$| f(x') - f^* | \leq \epsilon$$

and

$$p_i(x') \leq \epsilon \ (i = 1, \ldots, m) \tag{1.3}$$

$$|q_j(x')| \leq \epsilon \ (j = 1, \ldots, r) \tag{1.4}$$

where $f(x')$ is called an ϵ-global minimum.

In the following, the terms *minimum* and *minimizer* will mean *global minimum* and *global minimizer* respectively if we do not specify otherwise.

1.3 Methods for Solving the Global Optimization Problem

Among many kinds of classifications of methods for solving the global optimization problem [81, 88], we are interested in dividing the methods into the following two categories:

- point methods, and
- bounding methods.

Point methods compute function values at points and output as the minimum the function value at a point. Bounding methods compute lower and upper bounds of function over a box and give a lower bound and an upper bound for the minimum. Point methods are incapable of reliably solving the global optimization problem. Bounding methods produce correct global optimization solutions even in the presence of round-off errors. The strength of the point methods is their efficiency.

1.3.1 Point Methods

Point methods are iterative. From a starting point x_i (initially $i = 0$), a point method tries to converge to a minimum by finding another point x_{i+1} such that $f(x_{i+1}) < f(x_i)$

and using x_{i+1} as a new starting point.

Point methods may converge to a local minimum. Through starting the iteration at a number of randomly selected points or at the points of a grid, we increase the probability of finding all of the local minima. Thus it is possible that the smallest local minimum among all the local minima found is the global minimum.

A point method gives an upper bound for the global minimum without an indication of how close this bound is. Without additional information about the objective function, point methods can not even guarantee that an ϵ-global minimizer has been found [74].

1.3.2 Bounding Methods

Starting with a given box, a Cartesian product of intervals, bounding methods produce: (1) a lower bound and an upper bound for the global minimum of the function f, and (2) a list of small boxes. The union of these boxes will contain all global minimizers.

All bounding methods, such as Lipschitzian methods [81] and interval arithmetic methods [81], consist of the following three steps:

(1) **partitioning** the initial box into smaller boxes,

(2) **bounding** the function (and possibly its derivatives) over the boxes and the global minimum of the function, and

(3) **rejecting** (by using the bounds calculated in step 2) those boxes which do not contain a global minimizer.

The above three steps will be repeatedly executed until a certain termination condition is satisfied. The union of the remaining boxes will contain all global minimizers.

The bounding algorithm searches for the global minimum by exhaustively partitioning and "pruning" all of the feasible space, which do not contain a global minimizer.

Step 1 (i.e., the partitioning step) usually splits the initial domain, a given box, into smaller boxes. Choosing boxes as the geometric shape to be split has the following advantages:

- boxes are easily partitioned,
- boxes cover the feasible region without overlap,
- lower and upper bounds of the function f over boxes are easily computed.

If the feasible space itself is not a box, then an initial box containing it is used as the initial search domain. The equalities and inequalities which are used to characterize the true feasible space are then additionally used in the rejection phase of the algorithm to eliminate those subboxes lying in the initial box, but outside of the feasible region.

Step 2 (i.e., the bounding step) computes the lower bounds of the function f over boxes and finds an upper bound of the global minimum. Initially, the upper bound of the minimum is set to $+\infty$. At each iteration, it may be improved by the value of the function f at a point in the feasible region or a local minimum. For a given box B, the lower bound of f can be computed in one of the following ways:

- **Lipschitzian Approach.** It is based on the assumption that a Lipschitz constant, L, exists such that for any $u, v \in B$, we have

$$|f(u) - f(v)| \leq L||u - v||.$$

If the value of function f is known at point u, then a lower bound on the function value for all x between u and v can be determined by the following formula

$$f(u) - L||u - v||.$$

- **Interval Arithmetic.** Lower and upper bounds of the function f over the given box can be calculated by using interval arithmetic. An "*interval extension*" (to be defined later) F for f can be obtained easily. Such an interval function operates over a given box and returns an interval result bounding the range of the function over the given box.

- **Interval Constraints.** For the function f and box B, the interval constraints approach produces the same interval result as interval arithmetic if the initial interval for y is $(-\infty, +\infty)$, where y is the variable for the value of the function f. More importantly, when various additional conditions are used as constraints, not only can it produce a smaller interval result than interval arithmetic, but also it can make the given box B become smaller. This will be explained in more detail later.

Among the lower bounds of the function f over all the unrejected boxes, the lowest lower bound is a lower bound of the global minimum.

Step 3 (i.e., the rejecting step) rejects boxes by using various conditions. The following are some of commonly used conditions.

- rejecting boxes over which the lower bound of the function f is greater than an upper bound on the global minimum known so far;
- rejecting boxes which do not intersect with the feasible space (i.e., in which any point does not satisfy the conditions 1.1 and 1.2);
- rejecting boxes B not on the border of the initial box for which $0 \notin \{g(x) \mid x \in B\}$, where g is the gradient of the objective function, f;
- rejecting boxes for which the function is not concave anywhere within the boxes.

1.4 Related Work

Applications of BNR-Prolog. BNR-Prolog [10, 67, 7] is an interval constraint logic programming language. W. J. Older explored in BNR-Prolog the applications of interval constraints in several areas [73, 69, 65, 72, 70]. In global optimization, Older solved a global optimization problem from [29] by using a branch and bound algorithm and the Kuhn-Tucker conditions [72]. But he did not study the properties of the branch and bound algorithm for solving the global optimization problem.

UniCalc Solver. The UniCalc solver [3] was designed to solve systems of nonlinear equalities and inequalities. A technique analogous to interval constraints, called *subdefinite computations method* [76], is used in the UniCalc solver. It gives bounds on solutions. But it does not contain special methods for solving global optimization problems. To find the global minimum of a second order differentiable function without constraints, one has to follow the following three steps:

(1) create a system of equalities and inequalities from the known necessary conditions for a point x to be a local minimizer: the first order partial derivative of the objective function must be zero, and the diagonal elements of the Hessian matrix must be greater than or equal to zero.

(2) apply the UniCalc solver to the system, and thus find the intervals for the variables in the system.

(3) separate minimizers by using the root locating tool.

Numerica. Recently, P. Van Hentenryck and L. Michel reported their Numerica system for global optimization in [37, 38]. They use interval constraints, therefore obtaining the same advantages over interval arithmetic as reported in [16, 84, 17]. Numerica is built on Newton [5], an interval constraint logic programming language. An iterative interval Newton method was embodied in Newton. And the interval Newton method was combined with an internal splitting operation on intervals. It reported in [5] that Newton achieved one to two order magnitudes in speed over BNR-Prolog. They demonstrated the performance of their system. It seems that their system outperforms existing ones. Comparison are hard to make from their papers, since they only give timings and number of splits. The number of splits has the advantage of being machine independent. However, they did not report the "internal splits" performed by means of the iterative interval Newton method. Since there are hierarchical redundant conditions that can be used as constraints for solving the global optimization problem, it is not clear what role of each constraint is played.

1.5 An Overview of the Dissertation

In chapter 2, we first review the basic concepts of interval arithmetic, followed by interval constraints. We then demonstrate how the interval functions can be computed by using interval constraints and prove that computing an interval function in interval constraint gives the same result as in interval arithmetic.

In chapter 3, we give a hypernarrowing algorithm which can produce a smaller interval result for the range of function f over a given box than the corresponding interval function in interval arithmetic. And then we compare the semantics of hypernarrowing with that of a constraint solver, which is followed by applications.

In chapter 4, we first review interval arithmetic methods for unconstrained global optimization, describe a generic Branch-and-Bound algorithm for unconstrained global optimization, prove the properties of the algorithm, and propose improvements on the algorithm. We investigate the role of interval constraints in global optimization and show how to obtain an interval arithmetic version and interval constraint version of the algorithm, which is followed by the implementation of a variety of versions of the Branch-and-Bound algorithm in BNR-Prolog. We then compare the computational results produced by the interval arithmetic versions with those produced by the interval constraint ones.

In chapter 5, we discuss the differences between unconstrained and constrained optimization, which is followed by the review of interval arithmetic methods for constrained global optimization. We then present the transition from the Branch-and-Bound algorithm for the unconstrained global optimization to the variety for the constrained global optimization. We also study the effect of redundant conditions as interval constraints and compare the interval arithmetic methods with the interval

constraint ones.

In chapter 6, we summarize our results and contributions, and indicate directions of future researches.

Chapter 2

Interval Arithmetic and Interval Constraints

Interval arithmetic is an arithmetic defined on intervals, rather than on real numbers. A form of interval arithmetic perhaps first appeared in 1924 in [12, 47]. Modern development of interval arithmetic began with R. E. Moore's dissertation [59, 47]. The key idea of interval arithmetic is to bound by an interval the effect of errors from all sources, including approximation errors and errors in data. Since then, applications [2, 74, 36, 8] of interval arithmetic have been developed. In most applications, interval functions [64, 2, 74, 36] are used to bound the ranges of given functions. In general, the interval yielded by an interval function of a given function f is much larger than the range of f. It is therefore important to find ways to bound the range of f as closely as possible.

Using interval constraints, we can bound the range of the function f better than using interval arithmetic. The concept of constraints was formed gradually. Suther-

land [80] is one of pioneers of constraints. A basic consistency technique for interval constraints, the Waltz consistency algorithm, was given by Waltz [89]. Interval constraints has been embedded in several programming languages [18, 44, 23, 20, 5, 57]. It provides a general approach in computer problem solving where one expects the problem to be solved by merely entering the constraints among the variables of the problem without need for any algorithm in addition to a general-purpose constraint solver that comes with the system. Of course, in interesting problems the general-purpose constraint solver is neither efficient nor convenient to use. This is also the case in global optimization problems.

In this chapter, we prove that for any interval function F of a given function f, if we leave the value Y for F unconstrained initially and compute it in interval constraints, we get the same result for Y as we compute it in interval arithmetic. This indicates that interval arithmetic is a special case of interval constraints.

2.1 Basics of Interval Arithmetic

Before the use of interval arithmetic, bounds on the range of a function were sometimes obtained with Lipschitz constants. Judicious use of interval arithmetic allows such range bounds to be computed without extensive analysis. With *outward rounding* (to be explained later), interval arithmetic provide correct results from floating-point operations on computers in the presence of rounding errors.

We use R to denote the set of real numbers and $\mathcal{F}$ a finite subset of R. Typically, $\mathcal{F}$ is the set of floating-point numbers of a computer.

Definition 2.1: [40]

For every $a, b \in R$, the real interval (or R-interval for short)

$$[a, b] \text{ represents } \{x \in R \mid a \leq x \leq b\},$$
$$[a, +\infty] \text{ represents } \{x \in R \mid a \leq x\},$$
$$[-\infty, b] \text{ represents } \{x \in R \mid x \leq b\}, \text{ and}$$
$$[-\infty, +\infty] \text{ represents } R.$$

Note $+\infty \notin R$ and $-\infty \notin R$. $[a, +\infty]$, $[-\infty, b]$ and $[-\infty, +\infty]$ are just more convenient notations for $\{x \in R \mid a \leq x\}$, $\{x \in R \mid x \leq b\}$, and R respectively.

If we replace "$a, b \in R$" by "$a, b \in \mathcal{F}$" in the definition 2.1, then we obtain the definition of floating-point interval (or $\mathcal{F}$-interval for short). We denote the set of all $\mathcal{F}$-intervals by $I(\mathcal{F})$. For any $[l, u] \in I(\mathcal{F})$, we call l the lower bound and u the upper bound of $[l, u]$. For convenience, we will use lower bound function $lb : I(\mathcal{F}) \to \mathcal{F}$ and upper bound function $ub : I(\mathcal{F}) \to \mathcal{F}$, which are defined as follows:

$$lb([l, u]) = l, \; ub([l, u]) = u \quad \text{for any } [l, u] \in I(\mathcal{F}).$$

Note that real and floating-point intervals differ only in the bounds but every interval, real or floating-point, denotes a set of real numbers.

2.1.1 Interval Arithmetic Operations

Definition 2.2: [74, 36]

If $\circ \in \{+, -, *, /\}$ and $A, B \in I(\mathcal{F})$, then $A \circ B$, the result of the interval operation $\circ$, is the smallest interval in $I(\mathcal{F})$ containing

$$\{x \circ y \mid x \in A \text{ and } y \in B\}.$$

Other operations over $I(\mathcal{F})$ like sin, cos, log, etc. can be defined in the same way.

The image of each of the four basic interval operations is the range of the corresponding real operation except for A/B which is complicated if $0 \in B$. Although the above definition characterizes these operations mathematically, the practical use of interval arithmetic is due to the following theorems.

Theorem 2.1: [83]
For any $\circ \in \{+, -, *, /\}$ and $[a, b], [c, d] \in I(\mathcal{F})$

$$[a, b] \circ [c, d] = [\min(a \circ c, a \circ d, b \circ c, b \circ d), \max(a \circ c, a \circ d, b \circ c, b \circ d)]$$

except for $[a, b]/[c, d]$, which is more complicated if $0 \in [c, d]$.

Theorem 2.2: [74, 36]
Let $[a, b], [c, d] \in I(\mathcal{F})$, then

$$\begin{aligned}
[a, b] + [c, d] &= [a + c, b + d] \\
[a, b] - [c, d] &= [a - d, b - c] \\
[a, b] * [c, d] &= [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)] \\
[a, b]/[c, d] &= [a, b] * [\tfrac{1}{d}, \tfrac{1}{c}], \quad \text{if } 0 \notin [c, d]
\end{aligned}$$

Theorem 2.3: [83]

Let $0 \in [c,d]$, then

$$[a,b]/[c,d] = \begin{cases} \emptyset & \text{if } 0 \notin [a,b] \text{ and } c = 0 \text{ and } d = 0 \\ [-\infty, +\infty] & \text{if } a < 0 \text{ and } 0 < b \\ [-\infty, +\infty] & \text{if } c < 0 \text{ and } 0 < d \\ [\frac{a}{d}, +\infty] & \text{if } c = 0 \text{ and } 0 < d \text{ and } a \geq 0 \\ [-\infty, \frac{b}{d}] & \text{if } c = 0 \text{ and } 0 < d \text{ and } b \leq 0 \\ [-\infty, \frac{a}{c}] & \text{if } c < 0 \text{ and } d = 0 \text{ and } a \geq 0 \\ [\frac{b}{c}, +\infty] & \text{if } c < 0 \text{ and } d = 0 \text{ and } b \leq 0 \end{cases}$$

It is obvious that both interval operations + and $*$ are commutative and associative. However, the distributive laws do not hold. For any intervals A,B,C,$D \in I(\mathcal{F})$, we have [1]

$$A * (B + C) \subset A * B + A * C.$$

This property of the interval operations is called *subdistributive*. Moreover, although $[0,0]$ is an identity for addition and $[1,1]$ is an identity for multiplication, that is that for any interval $X \in I(\mathcal{F})$, we have

$$[0,0] + X = X$$

and

$$[1,1] \times X = X,$$

but we do not have

$$[0,0] = X - X$$

or

$$[1,1] = X/X.$$

[1] As we regard Bourbaki [11] and Halmos [31] as authorities on set-theoretic notation, we prefer $\subset$ for the subset relation to $\subseteq$. Similarly, $\supset$ rather than $\supseteq$.

This is due to the fact that the multiple occurrences of one variable X vary independently from the definition 2.2. This phenomenon is called *interval dependency.* Because of the interval dependency, the interval result of evaluating a function f over a given interval with interval arithmetic usually is much larger than the range of the function f. For example, if

$$f(x) = x^2 - x,$$

then evaluating f over $[0,1]$ with interval arithmetic gives

$$[0,1]^2 + [0,1] = [0,1] - [0,1] = [-1,1],$$

but the range of f over $[0,1]$ is $[-\frac{1}{4}, 0]$.

In floating-point arithmetic, real numbers are approximated by floating-point numbers using rounding. This rounding introduces roundoff errors. During the process of a computation, the rounding errors are accumulated. This may lead to the result of the computation to be wrong. This is not the case in floating-point interval arithmetic. In an interval operation, the lower bound of the result interval is rounded down to the largest floating-point number less than the exact lower bound, and the upper bound of the result interval is rounded up to the smallest floating-point number larger than the exact upper bound. This rounding process is called *outward rounding.* The outward rounding does not introduce roundoff errors. It guarantees that no answer, if there exists any, escapes from the rounded interval. This contributes to the guaranteed accuracy property of interval arithmetic. Therefore, a sound implementation of the outward rounding is essential in an interval arithmetic system.

The IEEE binary floating point standard [21, 30] prescribes three rounding modes: *nearest* (round to the nearest floating-point number), *round down* (round toward $-\infty$), and *round up* (round toward $+\infty$). The *nearest* mode is the default rounding

mode. For the implementation of outward rounding, we use the *round down* mode when computing lower bounds and the *round up* mode when computing upper bounds.

2.1.2 Inclusion Functions

In order to introduce the inclusion function of the real function $f : R^n \to R$, we define floating-point box. Let

$$\mathcal{F}_n = \{I_1 \times \cdots \times I_n \mid I_i \in I(\mathcal{F}) \text{ for } i = 1, \ldots, n\} \tag{2.1}$$

the set of Cartesian products of n floating-point intervals. We call $I_1 \times \cdots \times I_n \in \mathcal{F}_n$ floating-point box or $\mathcal{F}$-box for short, and $I \in \mathcal{F}_1$ (i.e., $I(\mathcal{F})$) floating-point interval or $\mathcal{F}$-interval for short.

Definition 2.3: [74]
Let $f : R^n \to R$. Let furthermore $\Box f(X)$ be the smallest interval containing the range of f over X, where X denotes $X_1 \times \cdots \times X_n$, and $X_i \subset R$. A function $F : \mathcal{F}_n \to \mathcal{F}_1$ is called an inclusion function for f if $\Box f(X) \subset F(X)$ for any $X \in \mathcal{F}_n$.

2.1.3 How to Get Inclusion Functions

Definition 2.4: [74]
Let $f : R^n \to R$, E be an expression for f, and $X \in \mathcal{F}_n$. The *natural interval extension* for E of f to X is the function $F : \mathcal{F}_n \to \mathcal{F}_1$ defined by the expression E' that is obtained from E by replacing each occurrence of the variable x by X, each arithmetic operation by the corresponding interval arithmetic operation and each predefined function by the corresponding inclusion function. We also call F an interval

function of f.

Lemma 2.1: [74]

For any $f, g, h : R \to R$, if $f(x) = g(h(x))$, G and H are inclusion functions for g and h, and $F(X) = G(H(X))$, then F is an inclusion function for f.

Corollary 2.1: [74]

For any real number functions $f, h_i (i = 1, \ldots, n) : R^n \to R, \quad g : R^m \to R$, if $f(x) = g(h_1(x), \ldots, h_m(x))$, G and H_i $(i = 1, \ldots, n)$ are inclusion functions for g and h_i, and $F(X) = G(H_1(X), \ldots, H_m(X))$, then F is an inclusion function for f.

Theorem 2.4: [74]

The natural interval extension for any expression for f as defined above is an inclusion function for f.

The mean-value form and Taylor form [74, 64] of f are two other kinds of inclusion functions for f.

In general, the value of an inclusion function for a given function f over $X \in \mathcal{F}_n$ is much larger than $\square f(X)$, the smallest interval containing the range of the function f over X. Therefore, it is important and challenging to find ways to approximate $\square f(X)$ as well as possible. It is challenging as $\square f(X)$ itself requires solving global minimization and maximization problems where f may not be a convex function.

2.2 Solving Interval Constraints

Interval constraints, built on interval arithmetic, is a generalization of interval arithmetic. The fundamental algorithm for solving interval constraint systems was pro-

posed by Davis [22]. Following the work presented in [18, 52, 68], F. Benhamou and W. Older [6] introduced the notions of approximation and narrowing, and applied them to constraints over real numbers, integers and Booleans. M. H. van Emden [82] generalized these notions to Herbrand universes and finite domains.

2.2.1 Interval Constraint Systems

Definition 2.5: [84]
An interval constraint system is an entity consisting of

(1) A **constraint conjunction**, $A_1 \wedge \cdots \wedge A_m$, where A_i $(i = 1, \ldots, m)$ are atomic formulas of first-order predicate logic. These formulas are called *primitive constraints* or *primitive relations*, each of which has one of the forms in Table 2.1, where the variables are interpreted as reals.

(2) A **state**, $I_1 \times \cdots \times I_n$, which is an $\mathcal{F}$-box. Each component (i.e., an interval) of the box is associated with a variable occurring in the constraint conjunction. It may happen that one or more of the intervals are empty. In this case the state denotes an empty set of tuples of values for the variables. Such a state is called a *failure state* or an *inconsistent state.*

For example,

$$exp(x, 2, y) \ \wedge \ exp(y, 2, z) \ \wedge \ sum(y, z, 1)$$

is a constraint conjunction, which is interpreted as

$$x^2 = y \ \wedge \ y^2 = z \ \wedge \ y + z = 1.$$

formula	interpretation
$\text{sum}(x, y, z)$	$x + y = z$
$\text{times}(x, y, z)$	$x * y = z$
$\exp(x, n, y)$	$x^n = y$ for integer $n \geq 2$
$\text{eq}(x, y)$	$x = y$
$\text{lt}(x, y)$	$x < y$
$\text{le}(x, y)$	$x \leq y$
$\text{gt}(x, y)$	$x > y$
$\text{ge}(x, y)$	$x \geq y$
$\sin(x, y)$	$\sin(x) = y$
$\cos(x, y)$	$\cos(x) = y$
$\tan(x, y)$	$\tan(x) = y$
$\text{asin}(x, y)$	$\arcsin(x) = y$
$\text{acos}(x, y)$	$\arccos(x) = y$
$\text{atan}(x, y)$	$\arctan(x) = y$
$\text{abs}(x, y)$	$\text{abs}(x) = y$
$\ln(x, y)$	$\ln(x) = y$
$\min(x, y, z)$	if $x < y$ then $z = x$ else $z = y$
$\max(x, y, z)$	if $x > y$ then $z = x$ else $z = y$

Table 2.1: Primitive constraints

Suppose that we also have the following $\mathcal{F}$-box

$$[-5.5, 6.5] \times [-1.5, 1.5] \times [-1.6, 1.6].$$

If we associate the interval $[-5.5, 6.5]$, $[-1.5, 1.5]$ and $[-1.6, 1.6]$ with the variables x, y and z in the above constraint conjunction respectively, then we get a state. This state and the above conjunction constitute an interval constraint system.

When writing a primitive relation, we prefer to write the interpretation as shown in the right-hand column of the table. This has the advantage of improved readability at the expense of a risk of confusion. For example, when writing "$x + y = z$" as a primitive relation, we should keep in mind that it is a ternary relation written in a sort of distributed infix notation (i.e., it is the set $\{< x, y, z > | \; x + y = z\}$.) and that it is not an instance of the binary equality relation involving the result of an addition.

Usually, we denote a constraint system by $C = < S, D >$, where $S = A_1 \wedge \cdots \wedge A_m$ is the constraint conjunction, $D = I_1 \times \cdots \times I_n$ is the state, in which intervals $I_1, \ldots, I_n$ are associated with the variables $x_1, \ldots, x_n$ in the constraint conjunction S respectively.

Definition 2.6:

A *solution* of an interval constraint system is an n-tuple of values for the variables that makes the constraint conjunction $A_1 \wedge \cdots \wedge A_m$ become true if each variable is replaced by the corresponding value.

Thus the set of solutions is a set of n-tuples, hence an n-ary relation, say r, where $r \subset I_1 \times \cdots \times I_n$.

Even though the interval constraint system defined here allows only a limited number of primitive constraints, it is general enough to represent equalities or in-

equalities between polynomials of any degree in any number of variables. Of course such polynomials have to be translated to the primitive constraints by introducing auxiliary variables. Such a translation is similar to the one performed by a Fortran or C compiler, where the target code's arithmetic instructions play the role of the primitive constraints here.

For example,

$$y = x_1^2 + x_2,\ x_1 \in [-10, 10],\ x_2 \in [-100, 50],\ y \in (-\infty, +\infty)$$

is translated to an interval constraint system where the constraint conjunction is

$$x_1^2 = z\ \wedge\ z + x_2 = y$$

and the state is

$$[-10, 10] \times [-100, 50] \times [-\infty, +\infty] \times [-\infty, +\infty]$$

which is associated with the variables x_1, x_2, y and z, where z is the auxiliary variable introduced.

2.2.2 Consistency Operators

Definition 2.7:

Let $p(x_1, \ldots, x_n)$ be a primitive constraint, and $X = X_1 \times \cdots \times X_n$ be the state, in which $X_1, \ldots, X_n$ are associated with the variables $x_1, \ldots, x_n$ in the constraint. A value $v \in X_i$ for the variable x_i is said to be consistent for p and X if we can find $x_j \in X_j$ $(j \in \{1, \ldots, n\}, j \neq i)$ such that

$$p(x_1, \ldots, x_{i-1}, v, x_{i+1}, \ldots, x_n)$$

holds.

The functionality of the consistency operator associated with a primitive constraint p and a given box is to remove inconsistent values from each interval of the box and find the smallest box containing all the consistent points in the given box.

For example, if $p(x, y, z)$ is $x+y = z$ (the primitive constraint *sum* from Table 2.1) and the intervals associated with x, y and z are $[0, 2], [0, 2]$ and $[3, 5]$ respectively, then all three intervals contain inconsistent values. Now $y \leq 2$ (from $y \in [0, 2]$) and $z \geq 3$ (from $z \in [3, 5]$) imply that $x = z - y \geq 1$. Hence the values in $[0, 1)$ for x are *inconsistent.* Thus we get the interval $[1, 2]$ for x from $[0, 2]$. Similar considerations rule out values in $[0, 1)$ for y and values in $(4, 5]$ for z. Removing all inconsistent values from the given intervals leaves the intervals $[1, 2]$ for x and y and $[3, 4]$ for z.

This is an example of the consistency operator associated with the primitive constraint *sum* acting on intervals associated with variables related by *sum*. In general there is a consistency operator associated with each primitive constraint relation p that acts on intervals associated with argument places of p by first removing all inconsistent values. As the resulting sets may not be intervals, the consistency operator includes a second step, which is to replace these sets by the least intervals containing them.

Before giving the definition of consistency operator formally, we introduce *projection* and *approximation.*

Definition 2.8: [6]

For every n-ary relation $r \subset R^n$, the projection of r, denoted by $\pi_i(r)$, is defined as

follows:

$$\pi_i(r) = \{x_i \in R \mid \exists x_1 \cdots \exists x_{i-1} \exists x_{i+1} \cdots \exists x_n \text{ such that } (x_1, \ldots, x_n) \in r\}.$$

Definition 2.9: [6, 85]

The approximation of a relation r, denoted by $ap(r)$, is the least (w.r.t. inclusion relation) $\mathcal{F}$-box containing r.

Definition 2.10: [6]

Let $r \subset R^n$. The consistency operator of r is the function C_r: $\mathcal{F}_n \rightarrow \mathcal{F}_n$, such that for any $u \in \mathcal{F}_n$,

$$C_r(u) = ap(u \cap r).$$

If r is one of the primitive constraints in Table 2.1, the corresponding consistency operator can be simply computed. The formulas used for computing consistency operators can be found in [18, 71, 85, 6, 51, 83]. We list a few of important ones in the following lemmas.

Lemma 2.2: [18, 71, 85, 6, 51]

If *sum* is the ternary relation

$$sum = \{< x, y, z > \mid x + y = z, \quad x, y, z \in R\}$$

and C_{sum} is the consistency operator of *sum*, then for any $[a, b] \times [c, d] \times [e, f] \in \mathcal{F}_3$ we have

$$\begin{aligned} C_{sum}([a, b] \times [c, d] \times [e, f]) \quad = \quad & [a, b] \cap ([e, f] - [c, d]) \times \\ & [c, d] \cap ([e, f] - [a, b]) \times \\ & [e, f] \cap ([a, b] + [c, d]). \end{aligned}$$

Thus, the consistency operator C_{sum} can be easily computed in terms of the interval arithmetic operations + and −. However, the formulas for computing C_{times} in [18, 71, 51] are complicated and incomplete, because the result of interval division $[a,b]/[c,d]$ is not defined if $0 \in [c,d]$. Even in extended interval arithmetic [33], $[a,b]/[0,0]$ is not defined. Because of the discontinuity at 0, W. Older broke the computation of C_{times} into twenty seven cases. By using symmetry, he reduced these twenty seven cases to three essentially different ones. The discontinuity also led to the consistency operator C_{times} incompletely implemented in BNR-Prolog for a number of years [71].

After deriving the actually executable computation rules for interval division $[a,b]/[c,d]$ especially when the divisor is near 0, M. H. van Emden [83] has presented the simple formula for the consistency operator of *times*.

Lemma 2.3: [83]
If *times* is the ternary relation

$$times = \{< x,y,z > |\ x * y = z, \quad x,y,z \in R\}$$

and C_{times} is the consistency operator of *times*, then for any $[a,b] \times [c,d] \times [e,f] \in \mathcal{F}_3$ we have

$$\begin{aligned} C_{times}([a,b] \times [c,d] \times [e,f]) \quad = \quad & [a,b] \cap ([e,f]/[c,d]) \times \\ & [c,d] \cap ([e,f]/[a,b]) \times \\ & [e,f] \cap ([a,b] * [c,d]). \end{aligned}$$

Lemma 2.4: [71]
If *eq* is the binary relation

$$eq = \{< x,x > |\ x \in R\}$$

and C_{eq} is the consistency operator of eq, then for any $[a,b] \times [c,d] \in \mathcal{F}_2$ we have

$$\begin{aligned} C_{eq}([a,b] \times [c,d]) &= [a,b] \cap [c,d] \times \\ & \quad [a,b] \cap [c,d]. \end{aligned}$$

Theoretically speaking, the relation r defined by the constraint system C can be approximated by the consistency operator C_r. However, there is no algorithm for C_r. The following prepares our mathematical model for the practical consistency operator of the relation r.

Suppose that we have a constraint system $C = < S, D >$, where $S = A_1 \wedge \cdots \wedge A_m$, $D = I_1 \times \cdots \times I_n$. For all $i = 1, \ldots, m$, if $x_{j_1}, \ldots, x_{j_{n_i}}$ are the variables occurring in A_i, then A_i denotes the n_i-ary relation $r_i \subset I_{j_1} \times \cdots \times I_{j_{n_i}}$ as the set of tuples that, when substituted for $x_{j_1}, \ldots, x_{j_{n_i}}$, make A_i true.

Lemma 2.5: [85]
$r = r_1 \bowtie \cdots \bowtie r_m$, where $\bowtie$ denotes the natural join.

The specification of r by means of $\exists x_1 \ldots \exists x_n (A_1 \wedge \cdots \wedge A_m)$ may suggest that $r = r_1 \cap \cdots \cap r_m$, but this is only the case if each of $A_1, \ldots, A_m$ contains all the variables in the constraint conjunction. Typically, however, every constraint contains only a small subset of all the variables.

Definition 2.11: [85]
For any $u \in \mathcal{F}_n$ and $r = r_1 \bowtie \cdots \bowtie r_m$,

$$T(u) = C_{r_1}(u^1) \bowtie \cdots \bowtie C_{r_m}(u^m)$$

where for all $i = 1, \ldots, m$, u^i is the projection of u on the subset of the variables that occur in A_i.

Proposition 2.1: [85]

(i) T is monotonic; (ii) For any $u \in \mathcal{F}_n$, $u \supset T(u) \supset C_r(u)$; (iii) $T(C_r(u)) = C_r(u)$.

Theorem 2.5: [85]

For any $u \in \mathcal{F}_n$, there exists a finite M such that $T(T^M(u))) = T^M(u)$. Moreover, $u \supset T^M(u) \supset C_r(u)$.

This suggests a consistency operator for approximating $C_r(u)$ and hence r.

Definition 2.12: [85]

The consistency operator of a constraint conjunction S is the function $\Psi_S : \mathcal{F}_n \to \mathcal{F}_n$, such that for any $u \in \mathcal{F}_n$,

$$\Psi_S(u) = T^M(u).$$

Lemma 2.6: [85]

(i) Ψ_S is contracting: $\Psi_S(u) \subset u$ for all $u \in \mathcal{F}_n$.

(ii) Ψ_S is monotonic: $u_1 \subset u_2 \Rightarrow \Psi_S(u_1) \subset \Psi_S(u_2)$ for all $u_1, u_2 \in \mathcal{F}_n$.

(iii) Ψ_S is idempotent: $\Psi_S(u) = \Psi_S(\Psi_S(u))$ for all $u \in \mathcal{F}_n$.
In other words, Ψ_S maps $\mathcal{F}_n$ to the fixpoints of Ψ_S.

2.2.3 Consistency Algorithms

Definition 2.13:

A state is a consistent state for an interval constraint system if this state is unchanged under the consistency operator of any of the primitive constraints in the constraint conjunction.

A consistency algorithm takes as input an interval constraint system, which has

a state. The algorithm reduces the intervals in this initial state to one of the interval constraint system's "consistent states". It performs this reduction in such a way that any solution contained in the initial state is also contained in the consistent state obtained from it. That is, it does not eliminate any solution. It may happen that the consistent state has intervals that have a width in the same order of magnitude as the precision of the machine arithmetic. This suggests that a unique solution is contained in the consistent state.

It may also happen that the consistency algorithm reduces the state to the failure state. In that case it has been shown that the original state contains no solutions. Finally, it may happen that the intervals of the consistent state that has been reached provide too little information about solutions to be useful. In this case, the state can be split into two or more sub-states. By applying the consistency algorithm to each of these sub-states, we can obtain more information about solutions.

One can get a consistent state by starting from an initial state and repeatedly execute a cycle in which all consistency operators are applied to the current state. As intervals never increase in size and as there are only finitely many machine numbers, such an iteration reaches a consistent state.

It is usually unnecessary to apply all consistency operators in each cycle. For a primitive constraint in the constraint system, its consistency operator needs to be applied only if the intervals associated with some of the variables occurring in the constraint are changed. This optimization is achieved by Waltz/Davis algorithm [22]. The process is often referred to as "constraint propagation" or "filtering". Thus the Waltz/Davis algorithm maps an initial state of an interval constraint system to a consistent state contained in it. In the process we obtain information about

solutions possibly contained in the initial state to the extent by which the intervals have contracted.

The algorithm in Figure 2.1, which is based on the algorithms in [22, 6], is the pseudocode of an efficient procedure. In the algorithm, $I(x_i)$ represents the interval associated with the variable x_i in the constraint system. The input of the algorithm is a constraint system $C = < S, D >$. The output is a failure state or a consistent state D'.

For example, if the constraint conjunction S is

$$y = x^2 \wedge z = y^2 \wedge y + z = 1$$

and the initial state D is

$$[0,1] \times [0,1] \times [0,1],$$

then no information is gained as this initial state is a consistent state. Through splitting the interval $[0,1]$ for x into two sub-intervals $[0,0.5]$ and $[0.5,1]$, we get two states:

$$D1 = [0,0.5] \times [0,1] \times [0,1]$$

and

$$D2 = [0,0.5] \times [0,1] \times [0,1].$$

Thus we have two interval constraint systems $C1 =< S, D1 >$ and $C2 =< S, D2 >$. The consistency algorithm will map the state $D1$ of the constraint system $C1$ to empty. That is, we will obtain a failure state from $D1$. This indicates that $D1$ does not contain any solution. Applying the consistency algorithm to $D2$, we obtain as corresponding consistent state

$$0.78615137775742[3,4]^2 \times 0.61803398874989[4,5] \times 0.38196601125010[5,6].$$

[2]This is a notation invented by M. H. van Emden [84]. We use this notation instead of the

```
1:  input:  a constraint system C = < S, D >
2:  output:  inconsistency or a consistent state D'

3:  initialize Q to the queue of all the constraints in S
4:  while Q is not empty do
5:          remove a constraint r(x_1, ..., x_m) from Q
6:          X := I(x_1) × ··· × I(x_m)
7:          apply C_r to X to obtain X', i.e., X' = C_r(X)
8:          if X' = ∅ then exit with inconsistency
9:          foreach x_i in {x_1, ..., x_m} do
10:                 if X'_i ≠ I(x_i) then
11:                     I(x_i) := X'_i
12:                     foreach r' ≠ r and r' ∉ Q in which x_i appears do
13:                                 put r' into Q
14:                 end-if
15:         end-foreach
16: end-while
17: output:  D' = I(x_1) × ··· × I(x_n)
```

Figure 2.1: Consistency algorithm

This suggests that a unique solution for x, y and z is contained in this state.

The above states are obtained from BNR-Prolog [10, 6], which includes an implementation of the consistency algorithm for interval constraint systems.

2.2.4 Interval Constraint Programming Languages

So far we have discussed interval constraints as mathematical entities. It is the basis of an interval constraint programming language (or constraint language for short). Among several paradigms for combining the current programming languages with interval constraints, the Constraint Logic Programming (CLP) is the most natural one [58].

Constraint languages are declarative [53, 39, 87]. In a constraint language, it is easy to specify as a constraint system constraints (i.e., equalities and inequalities) of arbitrary complexity and a domain for the variables in the constraints. We call this type of constraint system a *high-level constraint system.* To distinguish it from the constraint system defined in Definition 2.5, we call the latter a *low-level constraint system.* A high-level constraint system can not be solved directly. To solve it, a constraint language compiler or interpretor translates it into a low-level version by

---

conventional

$$[0.786151377757423, 0.786151377757424],$$

which is not only cumbersome, but requires close attention to determine the most significant digit at which the two numerals differ. We regard

$$0.78615137775742[3, 4]$$

as a *scaled interval* notation. The numeral before the brackets modifies what is inside by shifting and scaling.

transforming the constraints to a conjunction of primitive constraints and the domain to a state.

For example, the high-level constraint system

$$y = x_1 x_2,\ x_1 + x_2^2 \leq 0,\quad x_1, x_2 \in [0, +\infty]$$

will be translated into a low-level one, of which the constraint conjunction is

$$y = x_1 * x_2\ \wedge\ y_1 = x_2^2\ \wedge\ y_2 = x_1 + y_1\ \wedge\ y_2 \leq 0,$$

and the state is

$$I(x_1) \times I(x_2) \times I(y) \times I(y_1) \times I(y_2)$$

where $I(x_1) = I(x_2) = [0, +\infty]$ and $I(y) = I(y_1) = I(y_2) = [-\infty, +\infty]$. Note that the domains for the auxiliary variables y_1 and y_2 are $[-\infty, +\infty]$. In the high-level constraint system, if the domain of a variable is not specified, then the domain for that variable in the corresponding low-level constraint system will be also $[-\infty, +\infty]$.

Constraint languages provide operations on a given high-level constraint system. The operations include:

- adding an equality or inequality to the constraint system.
- splitting the interval for a variable in the constraint system into two or more sub-intervals. Using (1) the constraints of the constraint system, (2) one of these sub-intervals for this variable, and (3) the intervals for the other variables in the constraint system, one can specify a new constraint system.

Constraint languages also allow one to explicitly or implicitly invoke the consistency algorithm to provide information about possibly existing solutions of a high-

level constraint system. The amount of information ranges from nil to near the maximum allowed by the machine numbers.

To simplify notation, we will use the term *constraint system* for either a high-level constraint system or a low-level constraint system. Which one it represents should be determined from the context.

2.3 Computing Interval Functions in Interval Constraints

The result of an interval function of f over a box X calculated in interval arithmetic contains the range of f over X. The value of the interval function can also be computed in interval constraints. In this way we can compare interval arithmetic with interval constraints within the same software system. This is a fundamental requirement for the research reported in this dissertation.

Example. Suppose that we have function

$$f(x_1, x_2) = x_1 x_2 + x_2.$$

Using interval arithmetic,

$$\begin{aligned} f([-5,5],[-10,10]) &= [-5,5] * [-10,10] + [-10,10] \\ &= [-50,50] + [-10,10] \\ &= [-60,60]. \end{aligned}$$

The same result can be obtained in interval constraints.

At first, the function f can be equivalently translated to the following conjunction of primitive constraints

$$S = \text{times}(x_1, x_2, y_1) \wedge \text{sum}(y_1, x_2, y)$$

i.e., S is

$$y_1 = x_1 * x_2 \ \wedge \ y = y_1 + x_2$$

where y_1 and y are auxiliary variables introduced; y is the variable for the function; constraints *times* and *sum* correspond to the interval operations $*$ and $+$ respectively.

And then, associate each variable in the constraint conjunction with an interval. Associate x_1 with $X_1 = [-5, 5]$, x_2 with $X_2 = [-10, 10]$, the auxiliary variables y, y_1 with $Y, Y_1 = [-\infty, +\infty]$ respectively.

Thus we get the interval constraint system $C =< S, X_1 \times X_2 \times Y \times Y_1 >$.

The execution of the consistency algorithm with input C yields the consistent state $D' = [-5, 5] \times [-10, 10] \times [-60, 60] \times [-50, 50]$, where the interval for y is $[-60, 60]$, which is the same as that of $f([-5, 5], [-10, 10])$ computed in interval arithmetic. Initially, Q in the algorithm contains two primitive constraints "times(x_1, x_2, y_1)" and "sum(y_1, x_2, y)". From the initial state

$$X_1 \times X_2 \times Y \times Y_1 = [-5, 5] \times [-10, 10] \times [-\infty, +\infty] \times [-\infty, +\infty]$$

the consistent state D' can be reached in the following two iterations.

1. remove "times(x_1, x_2, y_1)" from Q and apply C_{times} to $X_1 \times X_2 \times Y_1$

$$\begin{aligned} C_{times}(X_1 \times X_2 \times Y_1) &= C_{times}([-5, 5] \times [-10, 10] \times [-\infty, +\infty]) \\ &= [-5, 5] \times [-10, 10] \times [-50, 50] \end{aligned}$$

The state of the constraint system becomes

$$X_1 \times X_2 \times Y \times Y_1 = [-5,5] \times [-10,10] \times [-\infty,+\infty] \times [-50,50].$$

2. remove "sum(y_1, x_2, y)" from Q and apply C_{sum} to $Y_1 \times X_2 \times Y$

$$\begin{aligned} C_{sum}(Y_1 \times X_2 \times Y) &= C_{sum}([-50,50] \times [-10,10] \times [-\infty,+\infty]) \\ &= [-50,50] \times [-10,10] \times [-60,60] \end{aligned}$$

The state of the constraint system becomes

$$X_1 \times X_2 \times Y \times Y_1 = [-5,5] \times [-10,10] \times [-60,60] \times [-50,50].$$

Definition 2.14:

Let E be the expression of f used for the interval function F. The constraint conjunction S and the variable y translated from E are defined as follows:

1. If E is a variable v, then it is only translated to v.

2. If E is a constant, then it is translated to the variable y.

3. If (1) E is (E_1 op E_2), (2) E_1 is translated to the constraint conjunction S_1 and variable y_1, (3) E_2 to S_2 and y_2, and (4) y_1 is not in S_2 and y_2 is not in S_1, then E is translated to the constraint conjunction $S = A_{op}(y_1, y_2, y) \wedge S_1 \wedge S_2$ and the variable y, where A_{op} is the primitive constraint corresponding to the interval operation *op*, y is different from y_1, y_2 and any other variable in $S_1 \wedge S_2$.

4. If (1) E is op(E_1) and (2) E_1 is translated to the constraint conjunction S_1 and variable y_1, then E is translated to the constraint conjunction $S = A_{op}(y_1, y) \wedge S_1$ and the variable y, where A_{op} is the primitive constraint corresponding to the interval operation *op*, y is different from y_1 and any other variable in S_1.

Definition 2.15:

Let

1. E be the expression of f used for the interval function F,
2. S be the constraint conjunction translated from E as defined in Definition 2.14,
3. x_i $(i = 1, \ldots, n)$ be the variables in E,
4. x_{n+j} $(j = 1, \ldots, m)$ be the variables translated from constants in E, and
5. x_{n+m+k} $(k = 1, \ldots, t)$ be the auxiliary variables introduced in S.

The constraint system C translated from E over $X = X_1 \times \cdots \times X_n \in \mathcal{F}_n$ is defined as $< S, I_1 \times \ldots \times I_{n+m+t} >$, where I_i $(i \in \{1, \ldots, n\})$ is X_i, I_{n+j} $(j \in \{1, \ldots, m\})$ is the constant from which the variable x_{n+j} is translated to, and I_{n+m+k} $(k \in \{1, \ldots, t\})$ is $[-\infty, +\infty]$.

Lemma 2.7:

If $op \in \{+, -, *, /\}$, then for any $A, B \in \mathcal{F}_1$,

1. A *op* B, the result of the interval operation *op*, can be obtained by applying the corresponding consistency operator to the box consisting of A, B and $[-\infty, +\infty]$;
2. applying the operator to the box does not change A or B.

Proof: Let x, y and z be the variables occurring in the primitive constraint corresponding to *op*. Let furthermore A and B be the intervals for x and y respectively, $[-\infty, +\infty]$ the interval for z.

For $A+B$, suppose that "$sum(x,y,z)$" is the corresponding primitive constraint and C_{sum} is the consistency operator of *sum*. From Lemma 2.2, we have the following by applying C_{sum} to $A \times B \times [-\infty,+\infty]$.

$$\begin{aligned} & C_{sum}(A \times B \times [-\infty,+\infty]) \\ = \; & A \cap ([-\infty,+\infty] - B) \times B \cap ([-\infty,+\infty] - A) \times [-\infty,+\infty] \cap (A+B) \\ = \; & (A \cap [-\infty,+\infty]) \times (B \cap [-\infty,+\infty]) \times (A+B) \\ = \; & A \times B \times (A+B). \end{aligned}$$

Thus, the interval for z is the result of $A+B$ and applying C_{sum} to the box $A \times B \times [-\infty,+\infty]$ does not change A or B.

Similarly, we can prove that (1) $A-B$, $A*B$ and A/B can be obtained by applying the corresponding consistency operator to the box consisting of A, B and $[-\infty,+\infty]$ and (2) applying the consistency operator to the box does not change A or B. ∎

Theorem 2.6:

For any $X \in \mathcal{F}_n$, if E is the expression of f used for the interval function F, y is the variable translated from E, and C is the constraint system translated from E over X as defined in Definition 2.15, then the output of the consistency algorithm with input C will be a consistent state D' and the interval in D' for y equals the value of $F(X)$ computed in interval arithmetic.

Proof: The proof is by induction on the number m of interval operations in E.

If $m=0$, then E is a variable or a constant. There is no primitive constraint in C. If E is the variable x and the interval for x is X, the consistency algorithm will output the consistent state $D'=X$. The interval for y (note that y is the same as x)

in D' is X, which equals the value of $F(X)$ computed in interval arithmetic. If E is the constant $[a, b]$, the algorithm will output $D' = [a, b]$. The interval for y in D' is $[a, b]$, which is the same as $F(X) = [a, b]$.

Suppose that for any $m \leq k$, the theorem is true.

If $m = k + 1$, i.e., the number of interval operations in E is $k + 1$, then E has the form $(E_1$ op $E_2)$ or op(E_1). For the former, let y_i and $C_i =< S_i, I_{1_i} \times \cdots \times I_{s_i} >$ be translated from E_i over X^i $(i = 1, 2)$, where X^i is the projection of X on the subset of the variables that occur in E_i, and $S_i = A_{1_i} \wedge \cdots \wedge A_{m_i}$. Suppose that A_{op} is the primitive constraint corresponding to the interval operation op and y is the variable translated from E, then $C =< S, I_1 \times \cdots \times I_s >$, where $S = S_1 \wedge S_2 \wedge A_{op}(y_1, y_2, y)$, is the constraint system translated from E over X.

It has been shown in [66] that the result of executing the consistency algorithm with a constraint system C as its input does not depend on the order in which primitive constraints are chosen from the constraint queue Q in Algorithm 2.1. Thus we can select constraints in the following order:

1. as long as there exists a primitive constraint of C_1 in Q, choose that constraint;
2. if there is a primitive constraint of C_2 in Q and no constraint of C_1 in Q, then choose the constraint of C_2;
3. choose A_{op} if there is no constraint of C_1 or C_2 in Q.

From Lemma 2.7, applying $C_{A_{op}}$ does not change the intervals for y_1 and y_2, so it does not lead to any constraints of C_1 or C_2 to be put back in Q. For each constraint of C_1 (C_2), applying the corresponding consistency operator does not affect any constraint of C_2 (C_1) since there is not an auxiliary variable shared by C_1 and C_2

according to Definition 2.14. Thus, the execution of the algorithm with input C until there is not any constraint of C_1 in Q is equivalent to the execution of the algorithm with the input C_1. And then the execution of the algorithm with input C until there is not any constraint of C_2 in Q is equivalent to the execution of the algorithm with the input C_2.

Suppose that E_i $(i = 1, 2)$ is the expression used for the interval function F_i. Since the number of interval operations in E_i is less than or equal to k, according to induction hypothesis, the execution of the consistency algorithm with the input C_i $(i = 1, 2)$ will output the consistent state D'^i and the interval for y_i in D'^i equals the value of $F_i(X^i)$ computed in interval arithmetic.

Thus from the interval for y is $[-\infty, +\infty]$ and Lemma 2.7, after applying the consistency operator of A_{op}, the consistency algorithm will output the consistent state D', in which the interval Y for y is:

$$Y = Y_1 \text{ op } Y_2,$$

where Y_i is the interval for y_i in D'^i. Therefore

$$Y = F_1(X^1) \text{ op } F_2(X^2) = F(X),$$

i.e., the interval for y in D' equals $F(X)$ computed in interval arithmetic.

Similarly, we can prove that the theorem is true if E has the form $\text{op}(E_1)$. ∎

Chapter 3

Hypernarrowing

Hypernarrowing is built on a consistency algorithm. It can be used to narrow the given intervals in a constraint system. The interval Y for F, hypernarrowed by the hypernarrowing, bounds the range of f closer than the corresponding interval function in interval arithmetic. We will present its origin and a hypernarrowing algorithm. And then we compare the semantics of hypernarrowing with that of a constraint solver, which is followed by applications.

3.1 Where the Idea of Hypernarrowing Comes From

Given a function f, an interval function F for f and an element $X \in \mathcal{F}_n$, the value Y of $F(X)$ computed in interval arithmetic is generally much larger than $\Box f(X)$, the smallest interval containing the range of f over X. Since $F(X)$ can be translated to a constraint system C, and after a consistency algorithm is applied to C, the interval

for Y in C is the same as the value of $F(X)$, why not probe the large interval for Y obtained from applying the consistency algorithm to C by means of constraints on Y? We show by example how to obtain a smaller interval for Y by adding constraints to the constraint system C.

Suppose we have the following function

$$f(x) = 12x_1^2 - 6.3x_1^4 + x_1^6 + 6x_2(x_2 - x_1)$$

Here we choose the natural interval extension of f (see Definition 2.4) as the inclusion function for f (see Definition 2.3). The BNR-Prolog system [10] for interval constraints allows one to enter the definition of the relation `f` defined as

```
f(X1,X2,Y) :- Y is 12*X1**2 - 6.3*X1**4 + X1**6 + 6*X2*(X2 -X1).
```

Read this as: "The relation `f` holds between `X1`, `X2` and `Y` if `Y` equals the polynomial shown." The definition of the relation `f` is translated to an equivalent conjunction of primitive constraints, typically introducing auxiliary variables.

For $[-2,4] \times [-2,4] \in \mathcal{F}_2$, we can enter the query

```
?- X1:real(-2, 4), X2:real(-2, 4), f(X1,X2,Y).
```

Because we leave `Y` initially unconstrained, the consistency algorithm yields an interval for `Y` equal to the result of the interval function computed in interval arithmetic. In this example the interval for `Y` is $[-1757, 4432]$.

When we add the constraint "$Y < -137$" to the constraint system by entering the query

```
?- X1:real(-2, 4), X2:real(-2, 4), f(X1,X2,Y), Y < -137.
```

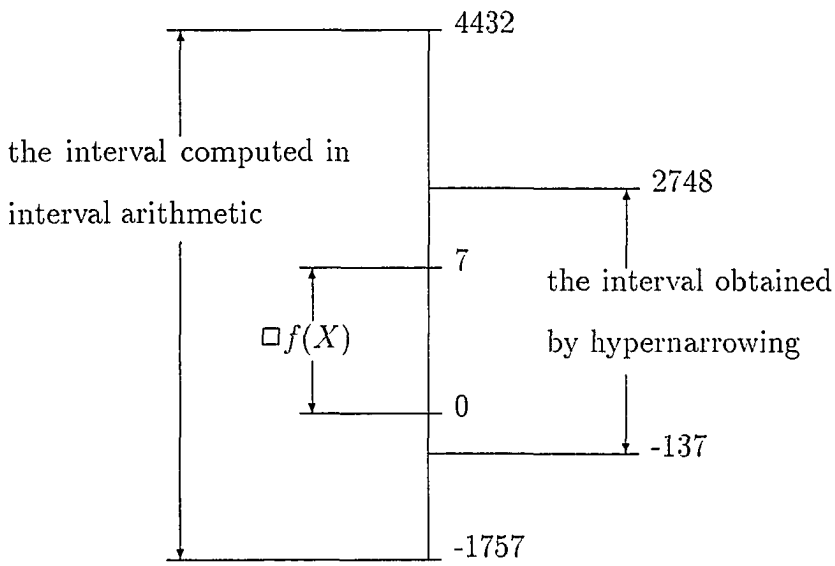

Figure 3.1: Intervals for $f(x) = 12x_1^2 - 6.3x_1^4 + x_1^6 + 6x_2(x_2 - x_1)$ over $[-2, 4] \times [-2, 4]$

The BNR-Prolog reports that the constraint system has no solution. This implies that there are no solutions for "$f(x_1, x_2) < -137$" within the given intervals for x_1 and x_2. Therefore, -137 is a lower bound of f over the given domain $[-2, 4] \times [-2, 4]$. When we add the constraint "$Y < -136$", the BNR-Prolog leaves open the possibility of solutions. Similarly, 2748 is a upper bound of f over the given domain. Thus with the interval constraint system, we can improve the interval $[-1757, 4432]$, obtained from the interval function used, to $[-137, 2748]$. W. J. Older discovered this idea and implemented it in "absolve" [67]. The hypernarrowing is a generalization of this idea, which will be described later.

By applying hypernarrowing to the interval for Y in the constraint system translated from an interval function, we can obtain an interval for Y that contains the range of f that is smaller than the one obtained by computing the interval function in interval arithmetic.

Figure 3.1 illustrates three kinds of intervals for the given function f over domain $X = [-2,4] \times [-2,4] \in \mathcal{F}_2$. The smallest interval is $\square f(X)$. The largest interval is the one obtained from the interval function F of f in interval arithmetic. The one in between is the interval obtained by hypernarrowing the interval Y for F in the constraint system translated from the interval function. This middle one gives a better approximation of $\square f(X)$.

3.2 A Hypernarrowing Algorithm

The hypernarrowing algorithm is based on a consistency algorithm. Suppose that we have a consistency algorithm, **Narrowing**, which takes the constraint system $C = < S, I_1 \times \cdots \times I_n >$ as its input, and $I_1' \times \ldots \times I_n'$ as its output, where $I_1, \ldots, I_n \in \mathcal{F}_1$, $I_1' \times \cdots \times I_n'$ is the narrowed version of $I_1 \times \cdots \times I_n$. The hypernarrowing algorithm takes as its input (1) the constraint system C, (2) a nonempty subset S_I of $\{I_1, \ldots, I_n\}$, in which we want the hypernarrowing algorithm to hypernarrow all the intervals, and (3) a predetermined tolerance ϵ. Its output is the hypernarrowed version of $I_1 \times \cdots \times I_n$.

The hypernarrowing algorithm is described in Figure 3.2. In the algorithm, the variable S_a is used for storing the intervals which the hypernarrowing algorithm will hypernarrow, S_d for storing intervals temporarily. Initially, $S_a = S_I$, $S_d = \emptyset$.

At first, the algorithm chooses an interval I_c from S_a, moves I_c from S_a to S_d, and hypernarrows the interval I_c. And then, for each narrowed interval I_i in the constraint system, it moves all the intervals in S_d which are related to I_i from S_d to S_a. This procedure is repeated until S_a is empty, i.e., all the intervals, which we want the hypernarrowing algorithm to hypernarrow, can not be narrowed any more.

Hypernarrowing($< S, I_1 \times \cdots \times I_n >, S_I, \epsilon$, output)

1: **begin**
2: $S_a := S_I$; $S_d := \emptyset$;
3: **while** $S_a \neq \emptyset$ **do**
4: choose an interval I_c from S_a;
5: move I_c from S_a to S_d;
6: **force**($I_c, < S, I_1 \times \cdots \times I_n >, \epsilon$, output);
7: **foreach** narrowed $I_i \in$ output **do**
8: move all I_j related to I_i from S_d to S_a.
9: **end-while**
10: **end**

force($I_c, < S, I_1 \times \cdots \times I_n >, \epsilon$, output)

1: **begin**
2: **forceL**($I_c, < S, I_1 \times \cdots \times I_n >, \epsilon$, output1);
3: **forceU**($I_c, < S,$ output1 $>, \epsilon$, output);
4: **end**

Figure 3.2: A hypernarrowing algorithm

For a given constraint system C and an interval I_c in C, to hypernarrow the interval I_c is to hypernarrow the lower part and upper part of I_c. Two procedures named *forceL* and *forceU* hypernarrow the lower part and upper part of I_c respectively, (see Figure 3.3). Since these two procedures are symmetric, we only explain the procedure forceL.

For a given interval I_c and the constraint system

$$< S, I_1 \times \cdots \times I_{c-1} \times I_c \times I_{c+1} \times \cdots \times I_n >$$

at first, *forceL* tries to find a lower part I_{c_1} of the interval I_c such that the constraint system

$$< S, I_1 \times \cdots \times I_{c-1} \times I_{c_1} \times I_{c+1} \times \cdots \times I_n >$$

is not consistent. And then, it applies the consistency algorithm to the following constraint system :

$$< S, I_1 \times \cdots \times I_{c-1} \times I_{c_2} \times I_{c+1} \times \cdots \times I_n >$$

where the interval $I_{c_2} = \{x \mid x \in I_c, x \notin I_{c_1}\}$. Thus the interval I_c can be at least hypernarrowed to I_{c_2} and the other intervals in the constraint system may also be narrowed if a nonempty interval I_{c_1} is found.

3.3 Waltz, Solve and Hypernarrowing

Waltz algorithm is a basic consistency algorithm. Both solve and hypernarrowing are built on the Waltz algorithm.

forceL($I_c, < S, I_1 \times \cdots \times I_n >, \epsilon$, output)

1: **begin**

2: $I_{c_1} := I_c$; $I_{c_2} := \emptyset$; output1 := $I_1 \times \cdots \times I_n$;

3: **while** output1 $\neq$ empty and width(I_{c_1})$> \epsilon$ **do**

4: bisect I_{c_1} into two intervals $I_{c_{11}}$ and $I_{c_{12}}$;

5: $I_{c_1} := I_{c_{11}}$; $I_{c_2} := I_{c_{12}} \cup I_{c_2}$;

6: **Narrowing**($< S, I_1 \times \cdots \times I_{c-1} \times I_{c_1} \times I_{c+1} \times \cdots \times I_n >$, output1)

7: **end-while**

8: **if** width(I_{c_1}) $> \epsilon$ **then**

9: **Narrowing**($< S, I_1 \times \cdots \times I_{c-1} \times I_{c_2} \times I_{c+1} \times \cdots \times I_n >$, output)

10: **else** output := $\langle I_1, \ldots, I_n \rangle$

11: **end**

forceU($I_c, < S, I_1 \times \cdots \times I_n >, \epsilon$, output)

1: **begin**

/***** symmetric to forceL *****/

11: **end**

Figure 3.3: Algorithms for hypernarrowing the lower and upper part of an interval

3.3.1 Comparing Waltz and Hypernarrowing

Although hypernarrowing is built on the Waltz algorithm, the iteration procedures in these two algorithms are the same. The differences between them exist in each step of the iterations.

1. Waltz maintains a queue Q of constraints, hypernarrowing a queue Q_I of intervals.

2. Waltz removes a constraint A_i from Q and applies the function similar to C_{A_i} to the domains of variables in A_i, hypernarrowing removes an interval I_k (the domain of a variable in some constraint) from Q_I and applies the function *force* to I_k.

3. For each narrowed interval I_j of the variable X_j in A_i, Waltz adds every constraint $A_s \neq A_i$ that contains X_j and is not in Q into Q; for each narrowed interval I_j, hypernarrowing adds every interval $I_s \notin Q_I$ that is related to I_j through a constraint into Q_I.

The hypernarrowing algorithm makes an improvement on Waltz. For example, suppose that we want to find the roots of the following function over [-1.0e+15,1.0e+15]$\times$ [-1.0e+15,1.0e+15].

$$f(x_1, x_2) = x_1^2 + x_1 x_2 + x_2^2$$

The solution of $f(x_1, x_2) = 0$ given by BNR-Prolog with the consistency algorithm similar to Waltz is the interval [-1.0e+15,1.0e+15] for x_1 and x_2. Here is the BNR-Prolog query for finding the solution.

```
?- [X1,X2]:real(-1.0e+15, 1.0e+15), X1**2 + X1*X2 + X2**2 == 0.
```

However, the solution given by our hypernarrowing algorithm with $\epsilon = 0.1$ is the interval $[-0.04, 0.04]$ for x_1 and $[-0.04, 0.04]$ for x_2. Here follows the query with hypernarrowing.

```
?- [X1,X2]:real(-1.0e+15, 1.0e+15), X1**2 + X1*X2 + X2**2 == 0,
    hypernarrowing([X1,X2], 0.1).
```

3.3.2 Comparing "Solve" with Hypernarrowing

Lemma 3.1:

The set $\mathcal{F}_n$ together with the set inclusion as the binary relation is a poset.

Figure 3.4 is a rough representation of the poset $\mathcal{F}_n$. In the poset $\mathcal{F}_n$, R^n is the largest element and $\emptyset$ is the least one.

Certain n-ary relations on R are singletons. They contain only one tuple. Let s be one such. In general, the least element $\check{s} \in \mathcal{F}_n$ such that $\check{s} \supset s$ is not a singleton. We call it the *approximation singleton*, being the image of a singleton under approximation. In the poset $\mathcal{F}_n$ there is nothing between $\emptyset$ and the approximation singletons.

Suppose that an n-ary relation $r \subset R^n$ is defined by a conjunction S_r of primitive constraints. For any $u \in \mathcal{F}_n$, $\Psi_{S_r}(u)$ is a fixpoint under u. But this fixpoint often gives too little information about the tuples in $r \cap u$.

In order to get more information about the tuples in $r \cap u$ for any $u \in \mathcal{F}_n$, **"solve"** splits u into $u_1, \ldots, u_k$ ($u_1 \cup \cdots \cup u_k = u$ and $u_1 \cap \cdots \cap u_k = \emptyset$). In general, $u_1, \ldots, u_k$ are not fixpoints. Hence $\Psi_{S_r}(u_1), \ldots, \Psi_{S_r}(u_k)$ can be usefully lower than $u_1, \ldots, u_k$ respectively. Because $u_1, \ldots, u_k$ are disjoint, so are $\Psi_{S_r}(u_1), \ldots, \Psi_{S_r}(u_k)$. If $\Psi_{S_r}(u_i)$

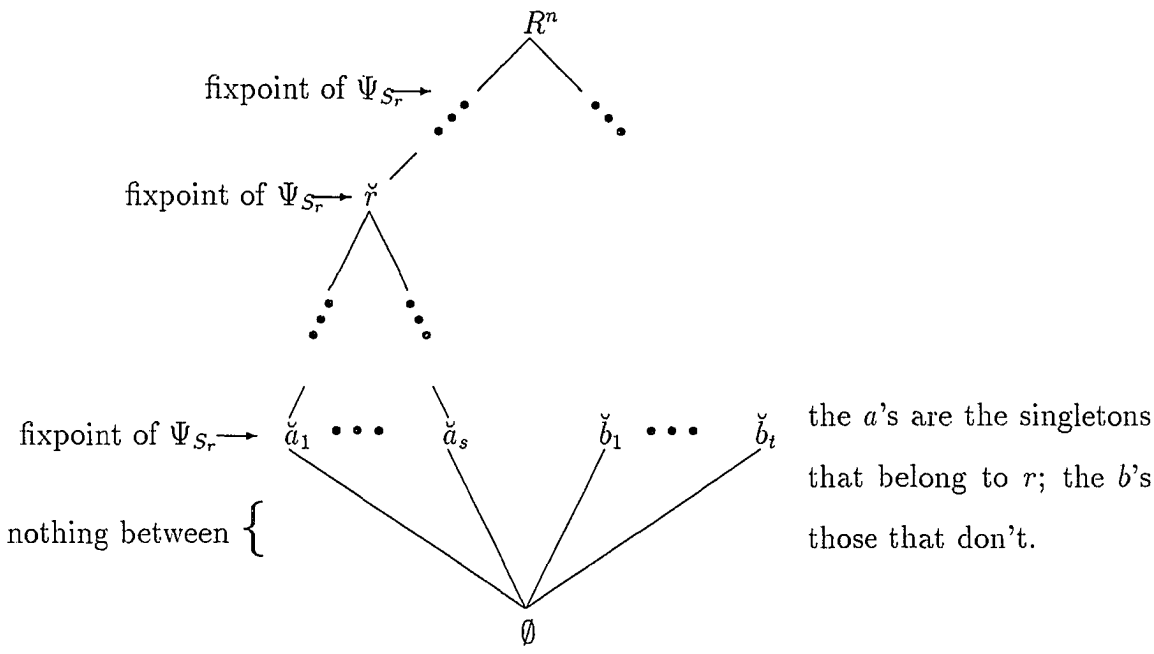

Figure 3.4: A rough map of the poset $\mathcal{F}_n$

($i \in \{1, \ldots, k\}$) is not low enough, **"solve"** will split u_i into sub-boxes. This process is performed on each sub-box u_s of u until $\Psi_{S_r}(u_s)$ is empty or low enough. If we make the partition fine enough, then $\Psi_{S_r}(u_s)$ will be either $\emptyset$ or an approximation singleton that may contain one or more tuples in $r \cap u$. That is what **"solve"** attempts to do (see Figure 3.5).

Hypernarrowing is different (see Figure 3.6). It searches for a partition that splits u into two subsets u_1 and u_2 such that $\Psi_{S_r}(u_2) = \emptyset$. One way to obtain u_2 consists of (1) selecting an interval I_i of u, (2) using binary search to find the largest possible low (or upper) part I_{i_1} of the interval I_i such that $\Psi_{S_r}(u_2) = \emptyset$, where

$$u_2 = I_1 \times \cdots I_{i-1} \times I_{i_1} \times I_{i+1} \cdots \times I_n.$$

u_1 is in general not a fixpoint, so $\Psi_{S_r}(u_1) \neq u_1$. Hypernarrowing is to repeat this

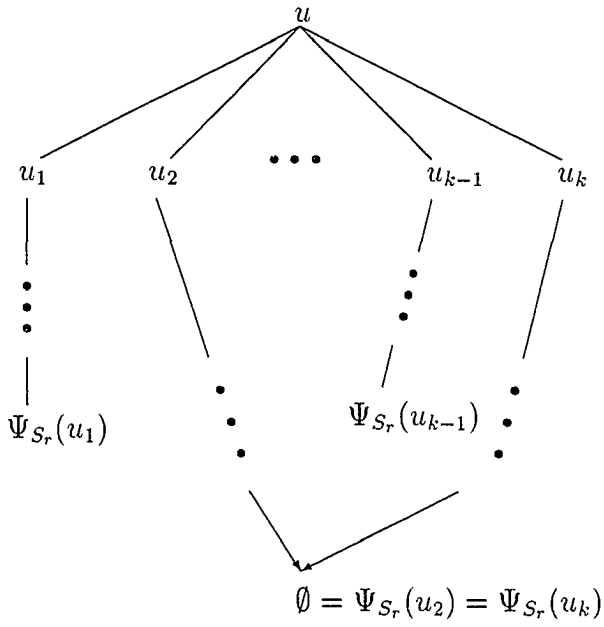

Figure 3.5: The behavior of "solve"

starting at $\Psi_{S_r}(u_1)$ until we can not find I_{i_1} whose width is greater than a given tolerance for any interval of u or sub-box of u. The best result of hypernarrowing is the least $\mathcal{F}$-box containing $r \cap u$.

3.4 Applications of the Hypernarrowing

Interval arithmetic has been used to solve many problems. However, most applications of interval arithmetic use the largest interval mentioned in Figure 3.1. Why not try the middle one which is obtained by hypernarrowing?

Global optimization using interval arithmetic is a typical application. In order to compare the differences between using the largest interval and the hypernarrowed

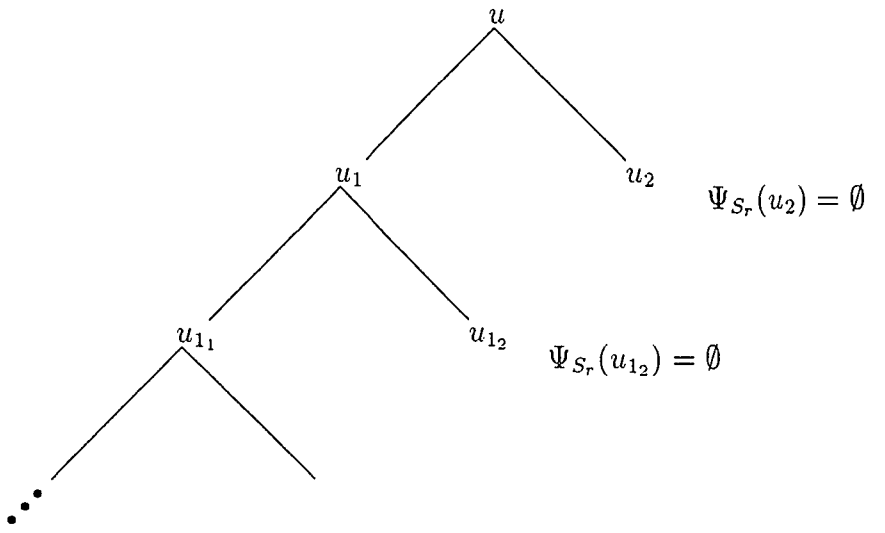

Figure 3.6: The behavior of hypernarrowing

one, at first, we implemented in BNR-Prolog a simple version of Hansen's algorithm [36] for unconstrained global optimization, which uses the largest interval. The simple version does not use the monotonicity, convexity tests, or the one step interval Newton method. And then we wrote a BNR-Prolog program to implement an improved version, which uses the middle interval in Figure 3.1. The only difference between these two programs is that the former uses the largest interval obtained by interval arithmetic and the latter uses the one obtained by hypernarrowing. The two programs are tested on the test problems used for testing the Hansen's algorithm. In order to avoid ambiguity, we denote the former by *Program A*, the latter by *Program B*. The computational results (see table 3.1) show that the Program B achieves a factor of two to eighteen in inclusion function evaluations. It is 1.8 to 9.7 times as fast as the Program A. However, compared to the program using interval arithmetic, the

		Problem 1	Problem 2	Problem 28	Problem 29
Input	W	6	8	3.78	2.4
	ϵ	10^{-1}	10^{-2}	10^{-2}	10^{-2}
Program A	N	2067	319	161	61
	t	505.9	49.5	27.7	5.4
Program B	N'	161	17	45	27
	t'	52.2	5.8	8.8	3
Comparison	N/N'	12.8	18.7	3.5	2.2
	t/t'	9.7	8.5	3.1	1.8

Table 3.1: Running results of Program A and Program B

program using interval constraints but not hypernarrowing only achieves a factor of two to five in inclusion function evaluations and is 1.8 to 4.9 times faster [84, 16]. The more interesting thing is that the execution of the Program B needs much less memory space than the Program A. All this suggests that we can improve most of the applications of interval arithmetic by using the middle interval which is obtained by hypernarrowing.

Here is an explanation of the symbols used in the table 3.1.

W Width of initial box.

ϵ Intended absolute accuracy.

N Number of all inclusion function evaluations till termination.

t Running time.

Chapter 4

Unconstrained Global Optimization

Given a function $f : R^n \rightarrow R$ and a domain $X \in \mathcal{F}_n$ ($\mathcal{F}_n$ is defined in equation 2.1), the unconstrained global optimization problem is to find

$$f^* = \min \{f(x) \mid x \in X\} \tag{4.1}$$

where f^* is the global minimum. We study interval constraint methods for seeking the minimum value f^* of f and the locations x^* at which this minimum value occurs. The interval constraint methods are based on interval arithmetic methods. Like the interval arithmetic methods, they can give a solution with guaranteed accuracy.

After reviewing the interval arithmetic methods and describing a Branch-and-Bound algorithm for solving 4.1, we prove the properties of the algorithm. We analyze the role of interval constraints in global optimization. Next we describe the implementations of both the interval arithmetic version and the interval constraint version of the algorithm. The implementations are done in the constraint logic programming

language BNR-Prolog [10, 67, 7], and are followed by computational results. Finally, we introduce improvements on the memory use of the Branch-and-Bound algorithm.

4.1 Overview of Interval Arithmetic Methods

Given a function $f : R^n \rightarrow R$ and a domain $X \in \mathcal{F}_n$, R. E. Moore [64] discovered that the combination of an inclusion function of f with a certain method of partitioning X could be used to determine the range and thus the global minimum and maximum values of the function f. S. Skelboe [78] was able to reduce the number of inclusion function evaluations used in Moore's method by combining Moore's method with the *Branch-and-Bound* principle. H. Ratschek and J. Rokne [74] call this method Moore-Skelboe algorithm.

The Moore-Skelboe algorithm first partitions the initial box X into smaller sub-boxes. The search for the global minimum f^* is performed iteratively by (1) selecting those sub-boxes, for which the lower bound on the function value is the least, and (2) partitioning these sub-boxes further. It is more likely that these boxes contain a global minimizer x^*.

For some sub-boxes, a simple test can show that they do not contain any global minimizer. But the Moore-Skelboe algorithm does not eliminate any. An improvement to the Moore-Skelboe algorithm was made by K. Ichida and Y. Fujii [43]. Their method combined Branch-and-Bound with a test that allows one to reject a sub-box. In the following, we discuss this test and some other tests proposed by E. R. Hansen [34, 35, 36].

Midpoint Test. Suppose that f_{ub}^* is the lowest upper bound of the global minimum obtained by evaluating the function f at the *midpoint* of each sub-box and choosing the smallest value. For a given box B, if we can determine that

$$f(x) \leq f_{ub}^* \quad \text{for some } x \in B \tag{4.2}$$

does not hold, then no minimizer can be in B. Thus B can be removed from further consideration.

We call the inequality 4.2 *midpoint condition.* One way to determine that the midpoint condition does not hold is to see whether the following inequality holds:

$$lb(f(B)) \leq f_{ub}^* \tag{4.3}$$

where $f(B)$ is the interval value of the natural interval extension of the function f over B. If the inequality 4.3 does not hold, then the midpoint condition does not hold and we can reject B without sacrifice of correctness. This way of rejecting boxes is called *midpoint test.*

Stationarity Test. The midpoint test only uses the information about the function f. It can and should always be used in Branch-and-Bound to reject sub-boxes. The stationarity test uses the gradient of f to determine whether a given box B can be rejected. The use of this test in Branch-and-Bound is subject to the existence of the gradient of f.

Definition 4.1: [29]

Suppose that

$$g_i(x) = \frac{\partial f(x)}{\partial x_i} \quad (i = 1, \ldots, n).$$

The vector

$$\nabla f(x) = (g_1(x), \ldots, g_n(x))^T$$

is the *gradient* of the function f at x, where ∇ is the *gradient operator.*

For a given box B, if we can determine that

$$g_i(x) = 0 \quad \text{for } i = 1, \ldots, n \quad \text{for some } x \in B \tag{4.4}$$

does not hold, then the gradient of f is not zero in B. Thus the global minimum can not occur in B and B can be rejected.

The equality 4.4 is called *stationarity condition.* One way of determining that the stationarity condition does not hold is to check whether the following inequality holds:

$$lb(g_i(B)) \leq 0 \leq ub(g_i(B)) \quad \text{for } i = 1, \ldots, n \tag{4.5}$$

where $g_i(B)$ is the interval value of the natural interval extension of the function g_i over B. If the inequality 4.5 does not hold, then the stationarity condition does not hold and we can reject B for further consideration. This way of discarding boxes is called *stationarity test* or *monotonicity test.*

Convexity Test. The convexity test uses the second-order derivatives of the function f to determine whether a given box B can be rejected. It can be used in Branch-and-Bound if f is twice differentiable.

Definition 4.2: [29]

Suppose that

$$h_{ij}(x) = \frac{\partial^2 f(x)}{\partial x_i \partial x_j} \quad (i, j = 1, \ldots, n).$$

The matrix

$$[h_{ij}(x)], \quad i,j = 1,\ldots,n$$

is the *Hessian matrix* of the function f at x.

Since the Hessian matrix of f at x must be positive semi-definite at a minimizer x^*, we can reject a box B if we can guarantee that the Hessian matrix is *not* positive semi-definite anywhere in B. One necessary condition for the Hessian matrix to be positive semi-definite is that its diagonal elements $h_{ii}(x)$ $(i = 1,\ldots,n)$ are non-negative. If we can determine that the following inequality does not hold,

$$h_{ii}(x) \geq 0 \quad \text{for } i = 1,\ldots,n \quad \text{for some } x \in B \tag{4.6}$$

then B can be rejected.

We call the inequality 4.6 *convexity condition.* One way to determine whether the convexity condition holds is to see if the following inequality holds:

$$lb(h_{ii}(B)) \geq 0 \quad \text{for } i = 1,\ldots,n \tag{4.7}$$

where $h_{ii}(B)$ is the interval value of the natural interval extension of the function h_{ii} over B. If the inequality 4.7 does not hold, then the convexity condition 4.6 does not hold and B can be rejected. This way of rejecting boxes is called *convexity test.*

In addition to these two tests, Hansen used *the linear method, the quadratic method* and *the one-pass interval Newton method* to reject a sub-box B' of B. These three methods are based on Taylor's theorem and interval analysis, and will be discussed in the following.

Linear Method. Consider the one-dimensional case for the linear method. From Taylor's theorem, expanding $f(x)$ about a point x_0, we have

$$f(x) = f(x_0) + (x - x_0)f'(\xi) \quad \text{for some } \xi \text{ between } x_0 \text{ and } x. \tag{4.8}$$

For a given box B, a point $x_0 \in B$ and an upper bound f^*_{ub} of f^*, if we can determine that

$$f(x_0) + (x - x_0)f'(\xi) \leq f^*_{ub} \quad \text{for some } x \in B \text{ and some } \xi \text{ between } x_0 \text{ and } x \tag{4.9}$$

does not hold, then B can be rejected. The condition 4.9 is called *linear condition.*

The linear method tries to find a sub-box B' of B such that the linear inequality

$$U + (x - x_0)V \leq 0 \quad \text{for some } x \in B' \tag{4.10}$$

does not hold (i.e., there is not any point $x \in B'$ at which the value of f is less than or equal to f^*_{ub}), where x_0 is a point of B, $U = f(x_0) - f^*_{ub}$ and $V = f'(B)$ are constant intervals. After B' is found, we can get the box B'_c complementary to B'. Thus the sub-box B' of B can be rejected through replacing B by $B \cap B'_c$. If $B \cap B'_c = \emptyset$, then B is totally rejected.

Quadratic Method. Still consider the one-dimensional case. Suppose that f has the derivative of the second order. From Taylor's theorem, expanding $f(x)$ about a point x_0, we have

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(\xi) \quad \text{for some } \xi \text{ between } x_0 \text{ and } x.$$

Given a box B, a point $x_0 \in B$ and an upper bound f^*_{ub} of f^*, if we can determine

that

$$f(x_0)+(x-x_0)f'(x_0)+\frac{1}{2}(x-x_0)^2f''(\xi) \leq f^*_{ub} \text{ for some } x \in B \text{ and } \xi \text{ between } x_0 \text{ and } x \tag{4.11}$$

does not hold, then B can be rejected. We call the inequality 4.11 *quadratic condition.*

The quadratic method tries to find a sub-box B' of B such that the quadratic inequality

$$f(x_0) - f^*_{ub} + (x - x_0)f'(B) + \frac{1}{2}(x - x_0)^2 f''(B) \leq 0 \quad \text{for some } x \in B' \tag{4.12}$$

does not hold. After B' is found, we can get the box B'_c complementary to B'. Thus the sub-box B' of B can be rejected through replacing B by $B \cap B'_c$. If $B \cap B'_c = \emptyset$, then B is totally rejected.

Newton Method. For the Newton method, consider the one-dimensional case. Suppose that g is the gradient of the function f and B is a box. From Taylor's theorem, we have

$$g(x) = g(x^*) + (x - x^*)g'(\xi) \quad \text{for some } \xi \text{ between } x^* \text{ and } x. \tag{4.13}$$

If x^* is the minimizer, then $g(x^*) = 0$. Thus we have

$$x^* = x - \frac{g(x)}{g'(\xi)} \quad \text{for some } \xi \text{ between } x^* \text{ and } x. \tag{4.14}$$

We call this equation *Newton condition.* One way of determining whether there exists $x^*, x \in B$ such that the condition 4.14 holds is to use interval arithmetic to try to find an $x^* \in B$.

Let B contain both x^* and x. Since ξ is between x and x^*, it follows that $\xi \in B$. Therefore, $g'(\xi) \in g'(B)$. Thus $x^* \in x - g(x)/g'(B)$. Based on this fact, the algorithm

described in Figure 4.1 can be used to find x^*. The algorithm produces two possible outputs. One is $X = \emptyset$, the other is $X \neq \emptyset$ and X is small enough. In the former case, it is determined that B does not contain any minimizer. Thus B can be rejected. For the latter case, B is replaced by X.

1: **input:** g, g', B;
2: $X := B$;
3: **while** X is not small enough **do**
4: $x := midpoint(X)$;
5: $X' := x - g(x)/g'(X)$;
6: $X := X \cap X'$;
7: **end-while**
8: **output:** $X = \emptyset$ or $X \neq \emptyset$;

Figure 4.1: Newton algorithm in Interval Arithmetic

It is wasteful to use the interval Newton method to iterate to convergence. The reason is that it may be converging to a local minimizer which is not a (global) minimizer. Because of this, Hansen uses one execution of the body of the loop of the Newton method, which is called one-pass interval Newton method [36].

4.2 Branch-and-Bound for Unconstrained Optimization

Branch-and-Bound [56, 77, 81] is an algorithm for finding the minimum of a function over a given set of points. This set may be countable, as in the discrete optimization problems, or the set may be a box in Euclidean n-space, as here. We will consider from the start the continuous instance of the Branch-and-Bound algorithm.

4.2.1 The Major Components of the Algorithm

The essential constituents of the algorithm to determine the global minimum are: (1) partitioning a box into sub-boxes, (2) finding an upper bound f_{ub}^* for f^*, and (3) using conditions to reject sub-boxes.

Partitioning a box into sub-boxes. Let

$$|[a,b]| = b - a$$

denote the width of the interval $[a,b]$. The width of a box $B = X_1 \times \cdots \times X_n$ is defined as the following

$$|B| = \max\{|X_i| \mid i = 1, \ldots, n\}.$$

Given a box $B = X_1 \times \cdots \times X_n$, if $X_k = [a_k, b_k]$ has the largest width among those for $X_1, \ldots, X_n$, and X_k is split into $X_{k_1} = [a_k, c_k]$ and $X_{k_2} = [c_k, b_k]$, where $c_k = a_k + (b_k - a_k)/2$ is the midpoint of the interval X_k, then the box B is partitioned into two sub-boxes

$$B_1 = X_1 \times \cdots \times X_{k-1} \times X_{k_1} \times X_{k+1} \times \cdots \times X_n$$

and

$$B_2 = X_1 \times \cdots \times X_{k-1} \times X_{k_2} \times X_{k+1} \times \cdots \times X_n.$$

Finding an upper bound f^*_{ub} of f^*. Suppose that f^*_{ub} is the lowest upper bound of f^* currently known. Initially $f^*_{ub} = ub(f(\text{mid}(X)))$, where X is the given box over which we want to find the global minimum. For a given sub-box B of X, we may be able to improve f^*_{ub} by arbitrarily selecting a point c in B and obtaining a new f^*_{ub} by

$$f^*_{ub} := \min(f^*_{ub}, ub(f(c))).$$

Using conditions to reject boxes. Given a box B and the lowest upper bound f^*_{ub} of f^* found so far, we can use the midpoint condition 4.2 in any circumstances. If the midpoint condition does not hold, then the box B can be rejected. One way of determining whether the midpoint condition holds is the midpoint test, which is the implementation of the midpoint condition in interval arithmetic. In the following section, we will give another way.

If the function f is differentiable, we can use the stationarity condition 4.4 and the linear condition 4.9 to reject the box under consideration.

When f is twice differentiable, we can use the convexity condition 4.6, the quadratic condition 4.11 and the Newton condition 4.14 to reject the box under consideration.

4.2.2 The Data Structures Used in the Algorithm

The data structures and variables manipulated by the algorithm are:

(1) Floating-point number f^*_{ub}, which is initialized as $ub(f(\text{mid}(X)))$.

(2) A list A and a priority queue L of tuples $< B, l, u >$, where B is a box in which a minimizer may occur, l is a lower bound of f over B, u is an upper bound for the minimum of f over B. The priority queue L is ordered according to increasing l.

A non-empty priority queue allows one to perform a "remove" operation. The result of this operation is the removal of a tuple $< B, l, u >$ with the lowest value of l. The priority queue also allows one to perform an "add" operation, which adds an arbitrary tuple to the priority queue. A priority queue is an "abstract data type" in the sense of [55]. Its significance is based on the fact that certain data structures allow *add* and *remove* to be performed in time $O(\log n)$ where n is the number of elements in the priority queue.

The list A is initially empty and contains on termination the tuples whose boxes are small enough to qualify as part of the final result. Priority queue L contains at any time tuples with boxes that may intersect with the minimizer. Initially, it only contains $< X, lb(f(X)), ub(f(\text{mid}(X))) >$, where X is the box of the given optimization problem. The function f always yields an interval, even when its argument is a point.

4.2.3 The Description of the Algorithm

After the data structures and initializations of the Branch-and-Bound algorithm thus given, we specify the iteration of the algorithm as in Figure 4.2.

The input data for the algorithm are :

- a function f and a given box X,
- a tolerance ϵ_X on box size and a tolerance ϵ_F on function width.

The output data of the algorithm are :

- a lower bound f^*_{lb} and an upper bound f^*_{ub} for f^*,
- a list

$$A = [< B_1, l_1, u_1 >, \ldots, < B_s, l_s, u_s >]$$

where for $i = 1, \ldots, s$, $l_i = lb(f(B_i))$, $u_i = ub(f(mid(B_i)))$, $|B_i| \leq \epsilon_X$ and $(u_i - l_i) \leq \epsilon_F$.

In the algorithm, the key part is "conditions($< B, l, u >, f^*_{ub}$)", which is underlined. The failure of any single one of the conditions implies that B does not contain any minimizer. Hence neither it, nor any subset of it should appear on the answer list A. Thus it can be omitted from L without sacrificing the correctness of the algorithm.

In the beginning, the algorithm initializes f^*_{ub}, L and A. In each iteration, it removes a tuple $< B, l, u >$ from L. If the tuple satisfies the following criterion for the answer list A:

$$|B| \leq \epsilon_X \wedge (u - l) \leq \epsilon_F \tag{4.15}$$

then it will be added into A. Otherwise, an attempt is made to reject the tuple by subjecting it to the conditions. If it is not rejected, then (1) the box B in the tuple will be partitioned into two sub-boxes B_1 and B_2, (2) lower bound l_i $(i = 1, 2)$ and upper bound u_i $(i = 1, 2)$ of f^* over B_i will be computed, (3) f^*_{ub} will be replaced by $\min(f^*_{ub}, u_1, u_2)$, and (4) two new tuples $< B_1, l_1, u_1 >$ and $< B_2, l_2, u_2 >$ will be added to L.

1: **input:** $f, X, \epsilon_X, \epsilon_F$;
2: $f^*_{ub} := ub(f(\text{mid}(X)))$;
3: $L := [< X, lb(f(X)), f^*_{ub} >]$;
4: $A := \emptyset$;
5: **while** $L \neq \emptyset$ **do**
6: remove $< B, l, u >$ from L;
7: **if** $|B| \leq \epsilon_X \wedge (u - l) \leq \epsilon_F$ **then**
8: add $< B, l, u >$ to A
9: **else if** conditions$(< B, l, u >, f^*_{ub})$ **then**
10: partition B into B_1 and B_2 with midpoints m_1 and m_2;
11: $l_1 := lb(f(B_1))$; $l_2 := lb(f(B_2))$;
12: $u_1 := ub(f(m_1))$; $u_2 := ub(f(m_2))$;
13: $f^*_{ub} := min(f^*_{ub}, u_1, u_2)$;
14: add $< B_1, l_1, u_1 >$ and $< B_2, l_2, u_2 >$ to L;
15: **end-if**
16: **end-while**
17: $f^*_{lb} = \min\{l_i \mid < B_i, l_i, u_i > \in A\}$;
18: **output:** f^*_{lb}, f^*_{ub}, A;

Figure 4.2: Branch-and-Bound algorithm for unconstrained optimization

In the algorithm, "partition" is assumed to have the property of that both B_1 and B_2 are non-empty and smaller than B, otherwise the algorithm may not terminate. The algorithm does not handle the case that B can not be partitioned. This case arises only for functions that are pathological in the sense that when the size of box B is less than or equal to the distance between two adjacent floating point numbers, the value of $(u - l)$ is still greater than ϵ_F.

Theorem 4.1:

When the algorithm terminates, it is the case that

$$f_{lb}^* \leq f^* \leq f_{ub}^*, \tag{4.16}$$

$$f_{ub}^* - f_{lb}^* \leq \epsilon_F, \tag{4.17}$$

$$x^* \in B_1 \cup \cdots \cup B_s \tag{4.18}$$

and

$$u_i - f_{lb}^* \leq 2\epsilon_F \quad \text{for any } i \in \{1, \ldots, s\}. \tag{4.19}$$

Proof: Since f_{lb}^* is a lower bound of the function f over the original given domain X and f_{ub}^* is the value of f at the midpoint of a sub-box of X, it is obvious that the inequality 4.16 holds.

Suppose that f_{lb}^* equals to l_j of the tuple $< B_j, l_j, u_j >$ in the answer list A, where $j \in \{1, \ldots, s\}$. From the criterion 4.15 for the answer list A, we have

$$u_j - l_j \leq \epsilon_F$$

i.e.,

$$u_j - f_{lb}^* \leq \epsilon_F.$$

Since f_{ub}^* is the lowest one among all u_i $(i = 1, \ldots, s)$, we have

$$f_{ub}^* - f_{lb}^* \leq \epsilon_F.$$

That is that the inequality 4.17 holds.

The Branch-and-Bound algorithm only rejects the sub-boxes of the original given box X that do not satisfy the conditions for the global minimizers. Thus all the minimizers that occur in X are still in the union of the sub-boxes of X that are not rejected. Therefore we have

$$x^* \in B_1 \cup \cdots \cup B_s.$$

That is that the formula 4.18 holds.

Suppose that the inequality 4.19 does not hold. Thus we have

$$u_k - f_{lb}^* > 2\epsilon_F \quad \text{for some } k \in \{1, \ldots, s\}, \tag{4.20}$$

i.e.,

$$u_k - \epsilon_F > f_{lb}^* + \epsilon_F \quad \text{for some } k \in \{1, \ldots, s\}. \tag{4.21}$$

From the criterion 4.15 for the answer list A, we have

$$u_k - l_k \leq \epsilon_F, \tag{4.22}$$

i.e.,

$$l_k \geq u_k - \epsilon_F. \tag{4.23}$$

From the inequalities 4.23 and 4.21, we have

$$l_k > f_{lb}^* + \epsilon_F. \tag{4.24}$$

From the inequality 4.17, we have

$$f_{lb}^* + \epsilon_F \geq f_{ub}^*. \tag{4.25}$$

From the inequalities 4.24 and 4.25, we have

$$l_k > f^*_{ub}. \tag{4.26}$$

This contradicts with "conditions($< B_k, l_k, u_k >, f^*_{ub}$)". If the inequality 4.26 holds, then the midpoint condition implemented either in interval arithmetic or in interval constraints can not be satisfied and the tuple $< B_k, l_k, u_k >$ will be rejected. Thus $< B_k, l_k, u_k >$ can not be in the answer list A. Therefore, the inequality 4.19 holds. ∎

This is the Branch-and-Bound algorithm for unconstrained global optimization. Its performance varies with the effectiveness of the implementation of "conditions($< B, l, u >, f^*_{ub}$)". If we use the midpoint test, the stationarity test, the convexity test, the linear method, the quadratic method and the one-pass interval Newton method, we obtain Hansen's Branch-and-Bound algorithm for unconstrained global optimization. Since all the tests and methods use interval arithmetic, we call Hansen's algorithm an interval arithmetic version of Branch-and-Bound. In the next section, we translate all the conditions into a pure constraint processing task in Interval Constraints.

4.3 Interval Arithmetic vs Interval Constraints

"conditions($< B, l, u >, f^*_{ub}$)" in the Branch-and-Bound algorithm can be implemented in two ways. One way is in interval arithmetic. The other is in interval constraints. Since BNR-Prolog can simulate interval arithmetic, we can implement these two ways in this same language.

Suppose that Y is the variable for the value of the function f over a box B and F_{ub}

is the lowest upper bound of f^* found so far. The following one line of BNR-Prolog code implements the midpoint condition in interval arithmetic:

```
less(Y, Fub) :- range(Y, [Ylb, Yub]), Ylb =< Fub.
```

This line of code means that the formula "less(Y, Fub)" holds if $lb(Y) \leq F_{ub}$, where $Y = f(B)$. In BNR-Prolog, the implementation of the midpoint condition in interval constraints is as follows:

```
less(Y, Fub) :- Y =< Fub.
```

This line of code means that the formula "less(Y, Fub)" holds if adding the interval constraint "$y \leq F_{ub}$" to the existing constraint system "$y = f(x), x \in B$" does not produce a failure state.

Although these two lines of code look similar, there is a huge difference between the behaviors of the program with one line and that with the other. This difference brings great effect on the performance of the algorithm.

4.3.1 Translating Conditions into an Interval Constraint System

For a given box B and an upper bound f^*_{ub} of f^*, we can create the following interval constraint system from the midpoint condition 4.2:

$$x \in B, \quad f(x) \leq f^*_{ub}. \tag{4.27}$$

For any other applicable conditions, we can add them as interval constraints into the constraint system 4.27 and obtain a new constraint system. For example, if the

function f has the first-order derivative, we can add the stationarity condition 4.4 and the linear condition 4.9 as interval constraints into the constraint system 4.27 and obtain the following interval constraint system:

$$x \in B, \quad f(x) \leq f^*_{ub}, \quad f'(x) = 0, \quad f(x_0) + (x - x_0)f'(\xi) \leq f^*_{ub} \tag{4.28}$$

where ξ is between x_0 and x.

Whenever we have an interval constraint system, we can invoke the consistency algorithm described in Figure 2.1 with this constraint system as an input. In BNR-Prolog, when we add an interval constraint into the existing constraint system and obtain a new constraint system, the consistency algorithm is automatically called with this new constraint system as its input. The output of the consistency algorithm will be inconsistency (i.e., failure) or a reduced non-empty box B'. The inconsistency means that some of the conditions translated into the interval constraints in the constraint system can not be satisfied for the given B and f^*_{ub}. That is that we can not find some $x \in B$ such that all the conditions translated into the interval constraints in the constraint system are satisfied. Thus the box B can be rejected. The inconsistency also implies that the box B is reduced to an empty box $\emptyset$. When the box B is reduced to an non-empty box B', we have that some of the conditions can not hold for $B - B'$.

4.3.2 Comparing Interval Arithmetic with Interval Constraints on Small Examples

Let us call *feasible* any box that contains at least one point that satisfies all the applicable conditions. The interval constraints translated from the midpoint condition 4.2,

the stationarity condition 4.4 or the convexity condition 4.6 can be used to *reduce* the size of the box B and also, in certain cases, to tell whether the box B in its entirety is infeasible.

The midpoint test, the stationarity test and the convexity test are the interval arithmetic implementations of the midpoint condition, the stationarity condition and the convexity condition respectively. These three tests do not change the size of the box B. They can only be used to determine whether the box B in its entirety is infeasible. If it is infeasible, we can eliminate it. However, if these tests can not tell us the box B in its entirety is infeasible, then the box B is not changed. In this case, these tests do nothing.

By using the linear method, the quadratic method or the Newton method, we may delete the entire box B or some of it. It seems that these methods have the same functionality as the interval constraints corresponding to them. In fact, the interval constraint versions work differently and have more power. To illustrate this, we present some examples; and then explain why interval constraints are more powerful than their corresponding interval arithmetic tests.

Example 1: Interval constraints more powerful for midpoint condition. Suppose that we wish to find the global minimum of

$$f(x_1, x_2) = x_1^2(4 + x_1^2(-2.1 + (1/3)x_1^2)) + 4x_2^2(x_2^2 - 1) + x_1x_2.$$

For the given box $B = [-1.9, -1.8] \times [0.6, 0.8]$ and $f_{ub}^* = -0.8$, the lower bound of $f(B)$ is -2.13, which is below f_{ub}^*. So we can not eliminate B by using the midpoint test, which is the implementation of the midpoint condition in interval arithmetic. Even though the lower bound of $f(B)$ is below f_{ub}^*, the box B is deleted just by the

interval constraint $f(x) \leq f_{ub}^*$, which is the implementation of the midpoint condition in interval constraints. The reason is that this constraint reduces the box B to an empty result.

For the given box $B = [0, 1.3] \times [0, 1.5]$ and $f_{ub}^* = -0.2$, the lower bound of $f(B)$ is -9.0, which is below f_{ub}^*. The midpoint test does nothing for this box. By the interval constraint $f(x) \leq f_{ub}^*$, the box $B = [0, 1.3] \times [0, 1.5]$ will be shrunk to $B = [0, 1.3] \times [0.2, 1.0]$, and the interval for $f(x)$ is shrunk to $[-3.6, -0.2]$ from $[-9.0, 20.0]$.

Example 2: Interval constraints more powerful for linear condition. Suppose that we wish to find the global minimum of

$$f(x) = x^6 - 15x^4 + 27x^2 + 250.$$

For the given box $B = [-3.0, -2.0]$ and $f_{ub}^* = 7.00000000000245$, using the linear method, we can not eliminate any part of the box B. But by using the interval constraints corresponding to the linear method, we get box B shrunk to $[-3.0, -2.2]$.

For the given box $B = [-3.147, -3.06067]$ and $f_{ub}^* = 7.00000000000245$, the linear method narrows the box to $[-3.0704, -3.06067]$. The interval constraints corresponding to the linear method produces an empty box. This means that the entire box is deleted.

4.3.3 Comparing Interval Arithmetic with Interval Constraints in General

The examples show that the linear method *can* be less effective than the same idea expressed as interval constraints. Below we show that this is *always* the case.

For a given box B and an upper bound f^*_{ub} of f^*, the linear method tries to delete infeasible points from B, i.e., tries to find a sub-box B' of B such that the linear inequality

$$U + (x - x_0)V \leq 0 \quad \text{for some } x \in B'$$

does not hold and removes B' from B, where x_0 is a point of B, $U = f(x_0) - f^*_{ub}$ and $V = f'(B)$ are constant intervals. Since both x and x_0 are contained in B, ξ is in B and $f'(\xi) \in f'(B)$.

If we use interval constraints for unconstrained global optimization, we just use the formula

$$f(x_0) - f^*_{ub} + (x - x_0)f'(\xi) \leq 0 \tag{4.29}$$

as an interval constraint, where ξ is between x_0 and x. It is not necessary to write a function or procedure to solve the linear inequality.

From the point of view of interval constraints, the linear method only considers the following linear constraint indirectly and independently

$$U + (x - x_0)V \leq 0, \quad x \in B$$

where x_0 is a point of B, U and V are constant intervals. $U = f(x_0) - f^*_{ub}$, $V = f'(B)$. Thus only the interval for x may be shrunk. U and V are not changed since they are constants. If we use interval constraints, the nonlinear constraint 4.29 is considered.

Not only may the interval for x be shrunk, but also the interval for V. In the place of V, is $f'(\xi)$, which may get shrunk. This shrunk interval may make the interval for x become smaller further.

In addition, the interval constraints in a constraint system interact with each other. For example, suppose that we have a box V and a constraint $A_1(x)$ in the constraint system, where $x \in V$. If adding constraint $A_2(x)$ (where $x \in V$) makes V shrink, this shrunk V will propagate to A_1, which may make the box V shrink further. This procedure repeats until the box V can not be shrunk or one of two constraints A_1 and A_2 can not be satisfied (i.e., an empty result is produced).

Thus we see that interval constraint versions of the linear condition are more effective than the linear method. Similar considerations will show that the interval constraint versions of the quadratic condition and the Newton condition are also more effective.

4.3.4 Overview of the Implementation Variations

There are six conditions (i.e., the midpoint, the stationarity, the convexity, the linear, the quadratic and the Newton condition). The Branch-and-Bound algorithm described in Figure 4.2 may use any non-empty sub-set of these six conditions. Each of these six conditions can be implemented in two different ways, in interval arithmetic and in interval constraints. There are many combinations.

We will focus on two sets of conditions. One set only contains the midpoint condition. The Branch-and-Bound algorithm using this set of conditions is called *order zero algorithm*, since it only uses the information about the objective function f. The other set includes the midpoint condition and stationarity condition. The Branch-

IAU_0	Interval Arithmetic version of order 0 algorithm
ICU_0	Interval Constraints version of order 0 algorithm
IAU_1	Interval Arithmetic version of order 1 algorithm
ICU_1	Interval Constraints version of order 1 algorithm

Table 4.1: Names of different algorithms

and-Bound algorithm using this set of conditions is called *order one algorithm*, since it uses the information about the objective function f and the first-order derivative of f.

Thus there are four combinations. We have four names for these combinations (see Table 4.1) if we use strings IAU and ICU to represent the Interval Arithmetic and Interval Constraint version of algorithm for Unconstrained global optimization, and subscript 0 and 1 to represent order 0 and 1 algorithm respectively.

4.3.5 Implementations of the Algorithms

We first present a generic implementation of the algorithm described in Figure 4.2. The implementations of IAU_0, ICU_0, IAU_1 and ICU_1 can be easily derived from this generic implementation. Figure 4.3 shows the main part of the BNR-Prolog program for this generic implementation. We omit the definitions of *partition*, *small*, *fb* and *min*. The user should provide the definitions of f, g_i and h_{ii} $(i = 1, \ldots, n)$.

For example, suppose that we want to find the minimum value of the following function

$$f(x_1, x_2) = 4(x_1 - 5)^2 + (x_2 - 6)^2.$$

We can define relation f in BNR-Prolog as follows:

```
f([X1,X2],Y) :- Y is 4*(X1 - 5)**2 + (X2 - 6)**2.
```

Similarly we can define g_i and h_{ii} $(i = 1, \ldots, n)$.

Here we briefly explain the relations defined in the program.

The formula "$partition(B, B_1, B_2)$" holds if the sub-boxes B_1 and B_2 are the result of partitioning the box B along the dimension in which the interval is the largest.

The formula "$small(B, Y, U)$" holds if $|B| \leq \epsilon_X$ and $U - lb(Y) \leq \epsilon_F$.

The formula "$fb(B, U)$" holds if U equals $ub(f(m))$, where m is the middle point of the box B.

The formula "$min([X_1, ..., X_n], M)$" holds if M is the minimal one of $X_1, ..., X_n$.

The formula "$insert([B, Y, U], L, L_a)$" holds if $L_a = [[B_1, Y_1, U_1], \ldots, [B_n, Y_n, U_n]]$ is the result of adding $[B, Y, U]$ into the list L such that $lb(Y_1) \leq \cdots \leq lb(Y_n)$.

The formula "$g\_min(L, F_{ub}, A0, A)$" holds if

(1) $L = [[B_1, Y_1, U_1], \ldots, [B_s, Y_s, U_s]]$, where B_i $(i = 1, \ldots, s)$ is a sub-box of the initial domain X, Y_i is the interval value of the function f over B_i, and U_i is the value of f at the middle point of B_i.

(2) F_{ub} is the lowest upper bound of f^* found so far.

(3) $A0$ has the same form as A, which is described just below.

(4) $A = [[B'_1, Y'_1, U'_1], \ldots, [B'_n, Y'_n, U'_n]]$ contains all elements which satisfy that $|B'_i| \leq \epsilon_X$ and $U'_i - lb(Y'_i) \leq \epsilon_F$ $(i = 1, \ldots, n)$.

```
g_min([],Fub,A,A) :- !.          % L = [], terminate, Answers are in A
g_min([[B,Y,U]|L0],Fub,A0,A) :- % remove [B,Y,U] from L=[[B,Y,U]|L0]
   small(B,Y,U), !,              %  width(B)=<Ex and U-lb(Y)=<Ey
   insert([B,Y,U],A0,A1),        %  insert [B,Y,U] into A0 and get A1
   g_min(L0,Fub,A1,A).
g_min([[B,Y,U]|L0],Fub,A0,A) :-
   conditions([B,Y,U],Fub), !,  % conditions may be satisfied
   partition(B,B1,B2),           % partition box B into B1 and B2
   f(B1,Y1), fb(B1,U1),          % Y1=f(B1),U1=ub(f(m1)),m1=midpoint(B1)
   f(B2,Y2), fb(B2,U2),          % Y2=f(B2),U2=ub(f(m2)),m2=midpoint(B2)
   min([Fub,U1,U2],Fub1),        % Fub1=min(Fub,U1,U2)
   insert([B1,Y1,U1],L0,L1),     % insert [B1,Y1,U1] and [B2,Y2,U2]
   insert([B2,Y2,U2],L1,L2),     % into L0 and get L2
   g_min(L2,Fub1,A0,A).
g_min([[B,Y,U]|L0],Fub,A0,A) :- % conditions can not be satisfied
   g_min(L0,Fub,A0,A).           % remove [B,Y,U] from [[B,Y,U]|L0]

conditions([B,Y,U],Fub) :-       %    conditions:
   less(Y, Fub),                                  % f(x) =< Fub
   g1(B,G1),eq(G1,0),...,gn(B,Gn),eq(Gn,0),       % gi(x) = 0, i=1,...,n
   h11(B,H1),ge(H1,0),...,hnn(B,Hn),ge(Hn,0), % hii(x) >= 0, i=1,...,n
   t1(B,Y,Fub),  % f(x)=f(x0)+(x-x0)f'(E)=<Fub, E is between x0 and x
   t2(B,Y,Fub),  % f(x)=f(x0)+(x-x0)f'(x0)+0.5(x-x0)f"(E)(x-x0)=<Fub
   newton(B,Y).  % x = x0 + f'(x0)/f"(E)

insert([B,Y,U],[],[[B,Y,U]]) :- !.
insert([B,Y,U],[[B1,Y1,U1]|T],[[B,Y,U],[B1,Y1,U1]|T]) :-
   range(Y, [L,_]), range(Y1, [L1,_]), L =< L1, !.
insert([B,Y,U],[[B1,Y1,U1]|T],[[B1,Y1,U1]|R]) :- insert([B,Y,U],T,R).
```

Figure 4.3: Code of Branch-and-Bound algorithm for unconstrained optimization

The Implementations of IAU_0, ICU_0, IAU_1 and ICU_1. The implementation given in Figure 4.3 becomes the implementation of an order zero algorithm if we just keep "*less(Y, Fub)*" in the definition of the relation *conditions*. If we have both "*less(Y, Fub)*" and "*g1(B,G1),eq(G1,0),...,gn(B,Gn),eq(Gn,0)*", we obtain the implementation of an order one algorithm.

We obtain the Interval Constraint version of the Branch-and-Bound algorithm by providing the definitions for the relations *less* and/or *eq* in Interval Constraint as follows:

```
less(Y, Fub) :- Y =< Fub.
```

and/or

```
eq  (Y, 0)   :- Y == 0.
```

If we define the relations *less* and/or *eq* in Interval Arithmetic in the following, we have the Interval Arithmetic version of the algorithm.

```
less(Y, Fub) :- range(Y, [Ylb, Yub]), Yub =< Fub.
```

and/or

```
eq  (Y, 0)   :- range(Y, [Ylb, Yub]), Ylb =< 0, 0 =< Yub.
```

4.3.6 Computational Results

Two groups of computational results are given in this section. The first group is the results from running IAU_0 and ICU_0. The second group is from running IAU_1 and ICU_1.

All the results are from running the algorithms on the test problems selected from [36, 42, 75]. We choose the test problems of polynomials with a few of variables. These test problems are listed in Table 4.2.

Problem	Objective Function
1	$f(x) = 4(x_1 - 5)^2 + (x_2 - 6)^2$
2	$f(x) = 0.26(x_1^2 + x_2^2) - 0.48x_1x_2$
3	$f(x) = (1.5 - x_1 + x_1x_2)^2 + (2.25 - x_1 + x_1x_2^2)^2 + (2.625 - x_1 + x_1x_2^3)^2$
4	$f(x) = 100(x_2 - x_1^2)^2 + (x_1 - 1)^2$
5	$f(x) = 12x_1^2 - 6.3x_1^4 + x_1^6 + 6x_2(x_2 - x_1)$
6	$f(x) = x^6 - 15x^4 + 27x^7 + 250$
7	$f(x) = \sum_{i=1}^{3} [(x_1 - x_i^2)^2 + (x_i - 1)^2]$
8	$f(x) = \sum_{i=1}^{3} [(Ax_1 - Bx_i^2)^2 + (Cx_i - D)^2], A = B = C = D = [0.999, 1.001]$
9	$f(x) = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 - 2x_3)^4 + 10(x_1 - x_4)^4$
10	$f(x) = \sum_{i=2}^{3} [(x_1 - x_i^2)^2 + (1 - x_i)^2]$

Table 4.2: Test problems for unconstrained global optimization

Before giving the computational results, we explain the symbols used for showing and comparing the results as follows:

IA Interval Arithmetic version of an algorithm.

IC Interval Constraint version of an algorithm.

W_i Width of initial box.

ϵ Intended tolerance for box size and function width.

W_{x^*} Width of final box for x^*.

W_{f^*} Width of bound on f^*.

N_{max} Maximum Number of elements in L during the execution.

t Running time.

NA The result is not available because of memory overflow.

In Table 4.5, 4.6, 4.9, 4.10, 4.13 and 4.14, each of these symbols with superscript A represents that the corresponding results are from running the interval arithmetic version of an algorithm, that with superscript C represents that the corresponding results are from running the interval constraint version of the algorithm, and that with M represents that the corresponding results are from running a version of the algorithm with the improvements on Memory use.

Running Results of IAU_0 and ICU_0. Table 4.3 shows the computational results of IAU_0 and ICU_0. Table 4.5 gives some comparisons between the computational results of these two implementations.

From Table 4.5, we can see that ICU_0 achieves a factor of 2.7 to 27.5 in speedup on the unconstrained optimization test problems over IAU_0. Moreover, most of the output results produced by ICU_0 are better than those produced by IAU_0, the final boxes for f^* and x^* produced by ICU_0 are usually smaller. Finally, running IAU_0 needs more memory space, the maximum numbers of elements (tuples) in L during the execution of IAU_0 on the test problems are about 2.7 to 27 times as large as those of elements in L during the execution of ICU_0.

Running Results of IAU_1 and ICU_1. Table 4.4 shows the computational results of IAU_1 and ICU_1. Table 4.6 gives some comparisons between the computational results of IAU_1 and ICU_1.

From Table 4.6, we can see that ICU_1 achieves a factor of 5.8 to 123.5 in speedup on the unconstrained optimization test problems over IAU_1. The maximum numbers of elements in L during the execution of IAU_1 on the test problems are about 5.9 to 126 times as large as those of elements in L during the execution of ICU_1.

4.4 Time vs Memory Trade-off

We present improvements on memory use of the Branch-and-Bound algorithm, which are applicable both to the interval arithmetic version and to the interval constraint version of the algorithm. This is followed by test results.

In general, the improvements on memory use imply additional computation time. The interesting thing is that these improvements speed up the execution of the Branch-and-Bound algorithm for solving some unconstrained global optimization problems.

4.4.1 Reducing Memory Use

During the iteration of the Branch-and-Bound algorithm, the upper bound f_{ub}^* may be updated to a new value, which is less than the previous f_{ub}^*. Whenever this happens, some tuples in L may be discarded by checking the conditions with the new f_{ub}^*. This is because for a given tuple, the conditions that succeed with the old f_{ub}^* may fail with the new one. Thus we may reduce the size of L by checking each tuple in L with the new f_{ub}^*.

This can be achieved by changing the line that updates f_{ub}^* in the algorithm and defining a procedure called "reduce" that discards tuples from a given priority queue

that do not satisfy the conditions.

The line in the algorithm

$$f_{ub}^* := min(f_{ub}^*, u_1, u_2)$$

is replaced by the following segment

$f_{ub}^{*\prime} := min(f_{ub}^*, u_1, u_2)$
if $f_{ub}^{*\prime} < f_{ub}^*$ **then**
reduce(L, $f_{ub}^{*\prime}$);
end-if
$f_{ub}^* := f_{ub}^{*\prime}$;

This code segment checks whether the new upper bound $f_{ub}^{*\prime}$ is less than the old upper bound f_{ub}^*. If $f_{ub}^{*\prime}$ is less than f_{ub}^*, it will call the procedure "reduce" to discard all the tuples in L that do not satisfy the conditions with $f_{ub}^{*\prime}$. Finally, it updates f_{ub}^* by $f_{ub}^{*\prime}$.

It is possible that the change of f_{ub}^* is quite small. In this case, calling "reduce" is not very useful. To avoid this kind of situation, we can replace

$$f_{ub}^{*\prime} < f_{ub}^*$$

in the **if-then** line of the segment by

$$f_{ub}^{*\prime} * c < f_{ub}^*$$

where $c > 1$ is a constant. The choice of c should suggest that $f_{ub}^{*\prime} * c < f_{ub}^*$ imply that $f_{ub}^{*\prime}$ is significantly less than f_{ub}^* . We choose $c = 1.5$ in our implementation.

The procedure "reduce" is described in Figure 4.4. For a given priority queue L of tuples and an upper bound $f^*_{ub}{}'$ of f^*, it checks every tuple in L with $f^*_{ub}{}'$. It removes all the tuples in L that do not satisfy the conditions.

In the procedure "reduce", "conditions($< B, l, u >$, $f^*_{ub}{}'$)" may shrink the box B. If B is shrunk to, say, B', $ub(f(m))$ may be less than u, where m is the midpoint of B'. Thus it is better to update u to $\min(u, ub(f(m)))$ and $f^*_{ub}{}'$ with $\min(f^*_{ub}{}', u)$ whenever B shrinks.

This can be achieved through replacing the **if-then** statement in the procedure "reduce" by one assignment statement and one complicated **if-then** statement. The improved procedure "reduce" is shown in Figure 4.5. It considers that the box B may shrink. In the improved "reduce", m is the midpoint of the shrunk box B, "shrunk(B, B_{old})" checks whether the box B is shrunk after the conditions are executed.

The Branch-and-Bound algorithm with the improvements is given in Figure 4.6. From this improved version, both an interval arithmetic and an interval constraint version can be obtained in the same way as from the algorithm in Figure 4.2.

4.4.2 Implementations of the Algorithms with Improvements on Memory Use

The first improvement on the Branch-and-Bound algorithm can be implemented through changing the last line of the third clause that defines *g_min* and defining in BNR-Prolog the relation *reduce* for the procedure "reduce" given in Figure 4.4. The last line of the third clause

```
g_min(L2,Fub1,A0,A).
```

```
    reduce(L, f*_ub')
1:  begin
2:          L' := ∅;
3:          while  L ≠ ∅  do
4:                  remove < B, l, u > from L;
5:                  if  conditions(< B, l, u >, f*_ub')  then
6:                      add < B, l, u > to L';
7:                  end-if
8:          end-while
9:          L := L';
10: end
```

Figure 4.4: Procedure "reduce"

```
    reduce(L, f*_ub')
1:  begin
2:          L' := ∅;
3:          while  L ≠ ∅  do
4:                  remove < B, l, u > from L;
5:                  B_old := B;
6:                  if  conditions(< B, l, u >, f*_ub')  then
7:                      if  shrunk(B, B_old)  then
8:                          l := lb(f(B));
9:                          u := min(u, ub(f(m)));
10:                         f*_ub' := min(f*_ub', u);
11:                     end-if  ;
12:                     add < B, l, u > to L';
13:                 end-if
14:         end-while
15:         L := L';
16: end
```

Figure 4.5: Procedure "reduce" considering box shrunk

1: **input:** $f, X, \epsilon_X, \epsilon_F$;
2: $f^*_{ub} := ub(f(\text{mid}(X)))$;
3: $L := [< X, lb(f(X)), f^*_{ub} >]$;
4: $A := \emptyset$;
5: **while** $L \neq \emptyset$ **do**
6: remove $< B, l, u >$ from L;
7: **if** $|B| \leq \epsilon_X \wedge (u - l) \leq \epsilon_F$ **then**
8: add $< B, l, u >$ to A
9: **else if** conditions$(< B, l, u >, f^*_{ub})$ **then**
10: partition B into B_1 and B_2 with midpoints m_1 and m_2;
11: $l_1 := lb(f(B_1))$; $l_2 := lb(f(B_2))$;
12: $u_1 := ub(f(m_1))$; $u_2 := ub(f(m_2))$;
13: $f^{*\prime}_{ub} := min(f^*_{ub}, u_1, u_2)$
14: **if** $f^{*\prime}_{ub} * c < f^*_{ub}$ **then**
15: reduce(L, $f^{*\prime}_{ub}$);
16: **end-if**
17: $f^*_{ub} := f^{*\prime}_{ub}$;
18: add $< B_1, l_1, u_1 >$ and $< B_2, l_2, u_2 >$ to L;
19: **end-if**
20: **end-while**
21: reduce(A, f^*_{ub});
22: $f^*_{lb} = \min\{l_i \mid < B_i, l_i, u_i > \in A\}$;
23: **output:** f^*_{lb}, f^*_{ub}, A;

Figure 4.6: Branch-and-Bound algorithm with improvements on memory use

can be changed by the following two lines

```
may_reduce(L2, Fub1, Fub, Lr),
g_min(Lr, Fub1, A0, A).
```

where the definition of *may_reduce* is described in Figure 4.7. The definition of the relation *reduce* is also given in BNR-Prolog in Figure 4.7.

The other improvement on the Branch-and-Bound algorithm can be implemented by adding a new parameter for a possibly lower f_{ub}^* in the relations *may_reduce* and making some changes for the definitions of *reduce* accordingly. The definitions of *may_reduce* and *reduce* for both of the improvements is given in Figure 4.8.

Using the same way as implementing IAU_0, ICU_0, IAU_1 and ICU_1, we obtain the implementations of IAU_0', ICU_0', IAU_1' and ICU_1' from the code given in Figure 4.3 for the implementation of the algorithm with the improvements. Here the $'$ represents an algorithm with the improvements.

4.4.3 IAU_0 and IAU_1 vs IAU_0' and IAU_1'

The results of running IAU_0 and IAU_1 are contained in Table 4.3 and 4.4 respectively. Here we just give the results of running IAU_0' and IAU_1' in Table 4.7 and 4.8 respectively.

Table 4.9 (4.10) gives some comparisons between the results of running IAU_0 and IAU_0' (IAU_1 and IAU_1').

From Table 4.9, we can see that IAU_0' is faster than IAU_0 on most of the test problems. Moreover, IAU_0' uses less memory space than IAU_0, the maximum numbers

```
may_reduce(L, Fub1, Fub, Lr) :-
    Fub1*1.5 < Fub, !,   % if Fub1 is significantly less than Fub
    reduce(L, Fub1, Lr). % then  reduce  L to Lr
may_reduce(L, Fub1, Fub, L). % otherwise, do nothing

reduce([], Fub1, []) :- !.
reduce([[B,Y,U]|T], Fub1, [[B,Y,U]|R]) :- % keep [B,Y,U] if
    conditions([B,Y,U],Fub1),!, % conditions may be held for Fub1
    reduce(T, Fub1, R).         % reduce T, the Tail of the List
reduce([_|T], Fub1, R) :- % conditions can not be held for Fub1
    reduce(T, Fub1, R).   % discard [B,Y,U] and reduce T
```

Figure 4.7: Code for the procedure "reduce"

```
may_reduce(L,Fub1,Fub,Lr,Fub2) :-
    Fub1*1.5 < Fub, !,      % if Fub1 is significantly less than Fub
    reduce(L,Fub1,Lr,Fub2). % then reduce L to Lr and get a new Fub2
may_reduce(L,Fub1,Fub,L, Fub2). % otherwise, do nothing

reduce([], Fub1, [], Fub1) :- !.
reduce([[B,Y,U]|T], Fub1, [[B,Y,U]|R], Fub2) :- % keep [B,Y,U] if
   copy(B, Bold),                  % Bold := B
   conditions([B,Y,U],Fub1), !,    % conditions may be held for Fub1
   may_shrink(B,Bold,Fub1,Fub11), % get new bound Fub11 if B is shrunk
   reduce(T, Fub11, R, Fub2).      % reduce T with the new bound Fub11
reduce([_|T], Fub1, R, Fub2) :- % conditions can not be held for Fub1
    reduce(T, Fub1, R, Fub2).   % discard [B,Y,U] and reduce T
```

Figure 4.8: Code for the procedure "reduce" considering box shrunk

of elements in L during the execution of IAU_0 are upto 1.9 times as large as those of elements in L during the execution of IAU_0'.

The results in Table 4.10 are similar to those in Table 4.9.

4.4.4 ICU_0 and ICU_1 vs ICU_0' and ICU_1'

The results of running ICU_0 and ICU_1 are contained in Table 4.3 and 4.4 respectively. Here we give the results of running ICU_0' and ICU_1' in Table 4.11 and 4.12 respectively.

Table 4.13 (4.14) gives some comparisons between the results of running ICU_0 and ICU_0' (ICU_1 and ICU_1').

From Table 4.13, we can see that ICU_0' is faster than ICU_0 on almost half of the test problems. It uses less memory space than ICU_0, the maximum numbers of elements in L during the execution of ICU_0 are upto 4.5 times as large as those of elements in L during the execution of ICU_0'.

The results in Table 4.14 are similar to those in Table 4.13.

Problem	Method	W_i	ϵ	W_{x^*}	W_{f^*}	N_{max}	t
1	IA	$2 \cdot 10^6$	10^{-1}	$6.1 \cdot 10^{-2}$	$2.9 \cdot 10^{-2}$	378	8.232
	IC	$2 \cdot 10^6$	10^{-1}	10^{-1}	$2.0 \cdot 10^{-2}$	14	0.299
2	IA	6	10^{-1}	$9.5 \cdot 10^{-2}$	10^{-1}	2267	305.2
	IC	6	10^{-1}	$9.7 \cdot 10^{-2}$	$9.6 \cdot 10^{-2}$	358	23.645
3	IA	9	10^{-5}	$8.6 \cdot 10^{-6}$	$2.5 \cdot 10^{-9}$	769	44.619
	IC	9	10^{-5}	$7.4 \cdot 10^{-6}$	$9.7 \cdot 10^{-11}$	95	4.767
4	IA	$2 \cdot 10^3$	10^{-5}	$7.5 \cdot 10^{-6}$	$2.5 \cdot 10^{-8}$	568	12.908
	IC	$2 \cdot 10^3$	10^{-5}	$8.8 \cdot 10^{-6}$	$2.7 \cdot 10^{-9}$	67	1.512
5	IA	6	10^{-1}	NA	NA	NA	NA
	IC	6	10^{-1}	$9.5 \cdot 10^{-2}$	$9.0 \cdot 10^{-2}$	114	8.262
6	IA	9	10^{-1}	NA	NA	NA	NA
	IC	9	10^{-1}	$6.2 \cdot 10^{-5}$	10^{-1}	434	23.635
7	IA	$2 \cdot 10^6$	10^{-5}	$7.3 \cdot 10^{-6}$	$3.1 \cdot 10^{-10}$	232	12.623
	IC	$2 \cdot 10^6$	10^{-5}	$7.4 \cdot 10^{-6}$	$5.4 \cdot 10^{-11}$	86	4.731
8	IA	$2 \cdot 10^6$	10^{-1}	$6.0 \cdot 10^{-2}$	$2.7 \cdot 10^{-2}$	154	13.593
	IC	$2 \cdot 10^6$	10^{-1}	$7.1 \cdot 10^{-2}$	$5.0 \cdot 10^{-3}$	23	2.178
9	IA	6	10^{-1}	$9.5 \cdot 10^{-2}$	$9.4 \cdot 10^{-2}$	900	56.392
	IC	6	10^{-1}	$9.1 \cdot 10^{-2}$	$1.2 \cdot 10^{-2}$	41	2.571
10	IA	3.78	10^{-5}	$7.2 \cdot 10^{-6}$	$4.7 \cdot 10^{-10}$	354	13.853
	IC	3.78	10^{-5}	$6.3 \cdot 10^{-6}$	$1.1 \cdot 10^{-10}$	28	1.444

Table 4.3: Running results of IAU_0 and ICU_0

Problem	Method	W_i	ϵ	W_{x^*}	W_{f^*}	N_{max}	t
1	IA	$2 \cdot 10^6$	10^{-1}	$6.1 \cdot 10^{-2}$	$2.0 \cdot 10^{-2}$	378	10.742
	IC	$2 \cdot 10^6$	10^{-1}	$1.6 \cdot 10^{-15}$	10^{-31}	3	0.087
2	IA	6	10^{-1}	$9.5 \cdot 10^{-2}$	10^{-1}	163	9.752
	IC	6	10^{-1}	$3.9 \cdot 10^{-3}$	$2.3 \cdot 10^{-6}$	3	0.093
3	IA	9	10^{-5}	$8.6 \cdot 10^{-6}$	$2.5 \cdot 10^{-9}$	769	78.645
	IC	9	10^{-5}	$7.4 \cdot 10^{-6}$	$9.7 \cdot 10^{-11}$	95	9.211
4	IA	$2 \cdot 10^3$	10^{-5}	$7.5 \cdot 10^{-6}$	$2.5 \cdot 10^{-8}$	568	16.964
	IC	$2 \cdot 10^3$	10^{-5}	$8.3 \cdot 10^{-6}$	$1.8 \cdot 10^{-9}$	34	1.484
5	IA	6	10^{-1}	$9.5 \cdot 10^{-2}$	$5.3 \cdot 10^{-2}$	31	2.421
	IC	6	10^{-1}	$7.7 \cdot 10^{-29}$	$4.6 \cdot 10^{-56}$	3	0.142
6	IA	9	10^{-1}	$3.4 \cdot 10^{-5}$	$5.6 \cdot 10^{-2}$	135	7.822
	IC	9	10^{-1}	$1.9 \cdot 10^{-13}$	$3.1 \cdot 10^{-10}$	7	0.236
7	IA	$2 \cdot 10^6$	10^{-5}	$7.3 \cdot 10^{-6}$	$3.1 \cdot 10^{-10}$	232	17.434
	IC	$2 \cdot 10^6$	10^{-5}	$7.1 \cdot 10^{-6}$	$8.3 \cdot 10^{-11}$	39	3.021
8	IA	$2 \cdot 10^6$	10^{-1}	$6.0 \cdot 10^{-2}$	$2.7 \cdot 10^{-2}$	154	22.455
	IC	$2 \cdot 10^6$	10^{-1}	$6.9 \cdot 10^{-2}$	$1.4 \cdot 10^{-2}$	13	1.959
9	IA	6	10^{-1}	$9.5 \cdot 10^{-2}$	$9.4 \cdot 10^{-2}$	770	62.673
	IC	6	10^{-1}	$7.5 \cdot 10^{-2}$	$2.6 \cdot 10^{-3}$	33	3.271
10	IA	3.78	10^{-5}	$7.2 \cdot 10^{-6}$	$4.7 \cdot 10^{-10}$	354	19.044
	IC	3.78	10^{-5}	$6.2 \cdot 10^{-6}$	$1.4 \cdot 10^{-11}$	39	2.895

Table 4.4: Running results of IAU_1 and ICU_1

Problem	$W^A_{x^*}/W^C_{x^*}$	$W^A_{f^*}/W^C_{f^*}$	N^A_{max}/N^C_{max}	t^A/t^C
1	610	1.45	27	27.5
2	0.98	1.04	6.3	12.9
3	1.16	25.8	8.1	9.4
4	0.85	9.3	8.5	8.5
5	NA	NA	NA	NA
6	NA	NA	NA	NA
7	0.99	5.7	2.7	2.7
8	0.85	5.4	6.7	6.2
9	1.04	7.8	22.0	21.9
10	1.43	4.3	12.6	9.6

Table 4.5: Comparisons between IAU_0 and ICU_0

Problem	$W^A_{x^*}/W^C_{x^*}$	$W^A_{f^*}/W^C_{f^*}$	N^A_{max}/N^C_{max}	t^A/t^C
1	$3.8 \cdot 10^{13}$	$2.0 \cdot 10^{29}$	126	123.5
2	24.4	$4.3 \cdot 10^{4}$	54.3	104.9
3	1.16	25.8	8.1	8.5
4	0.90	13.9	16.7	11.4
5	$1.2 \cdot 10^{27}$	$1.2 \cdot 10^{54}$	10.3	17.1
6	$1.8 \cdot 10^{8}$	$1.8 \cdot 10^{8}$	19.3	33.1
7	1.03	3.7	5.9	5.8
8	0.87	1.9	11.9	11.5
9	1.27	36.2	23.3	19.2
10	1.16	33.6	9.1	6.6

Table 4.6: Comparisons between IAU_1 and ICU_1

Problem	Method	W_i	ϵ	W_{x^*}	W_{f^*}	N_{max}	t
1	IA	$2 \cdot 10^6$	10^{-1}	$6.1 \cdot 10^{-2}$	$2.9 \cdot 10^{-2}$	269	7.902
2	IA	6	10^{-1}	$9.5 \cdot 10^{-2}$	10^{-1}	1950	262.75
3	IA	9	10^{-5}	$8.6 \cdot 10^{-6}$	$2.5 \cdot 10^{-9}$	599	40.418
4	IA	$2 \cdot 10^3$	10^{-5}	$7.5 \cdot 10^{-6}$	$2.5 \cdot 10^{-8}$	378	14.273
5	IA	6	10^{-1}	NA	NA	NA	NA
6	IA	9	10^{-1}	NA	NA	NA	NA
7	IA	$2 \cdot 10^6$	10^{-5}	$7.3 \cdot 10^{-6}$	$3.1 \cdot 10^{-10}$	121	12.393
8	IA	$2 \cdot 10^6$	10^{-1}	$6.0 \cdot 10^{-2}$	$2.7 \cdot 10^{-2}$	121	13.543
9	IA	6	10^{-1}	$9.5 \cdot 10^{-2}$	$9.4 \cdot 10^{-2}$	698	51.071
10	IA	3.78	10^{-5}	$7.2 \cdot 10^{-6}$	$1.5 \cdot 10^{-10}$	251	12.603

Table 4.7: Running results of IAU_0'

Problem	Method	W_i	ϵ	W_{x^*}	W_{f^*}	N_{max}	t
1	IA	$2 \cdot 10^6$	10^{-1}	$6.1 \cdot 10^{-2}$	$2.0 \cdot 10^{-2}$	269	10.472
2	IA	6	10^{-1}	$9.5 \cdot 10^{-2}$	10^{-1}	163	9.792
3	IA	9	10^{-5}	$8.6 \cdot 10^{-6}$	$2.5 \cdot 10^{-9}$	599	74.515
4	IA	$2 \cdot 10^3$	10^{-5}	$7.5 \cdot 10^{-6}$	$2.5 \cdot 10^{-8}$	568	16.964
5	IA	6	10^{-1}	$9.5 \cdot 10^{-2}$	$5.3 \cdot 10^{-2}$	31	2.50
6	IA	9	10^{-1}	$3.4 \cdot 10^{-5}$	$5.6 \cdot 10^{-2}$	134	7.812
7	IA	$2 \cdot 10^6$	10^{-5}	$7.3 \cdot 10^{-6}$	$3.1 \cdot 10^{-10}$	121	17.444
8	IA	$2 \cdot 10^6$	10^{-1}	$6.0 \cdot 10^{-2}$	$2.7 \cdot 10^{-2}$	121	22.485
9	IA	6	10^{-1}	$9.5 \cdot 10^{-2}$	$9.4 \cdot 10^{-2}$	568	58.032
10	IA	3.78	10^{-5}	$7.2 \cdot 10^{-6}$	$4.7 \cdot 10^{-10}$	251	17.914

Table 4.8: Running results of IAU_1'

Problem	$W^A_{x^*}/W^{AM}_{x^*}$	$W^A_{f^*}/W^{AM}_{f^*}$	N^A_{max}/N^{AM}_{max}	t^A/t^{AM}
1	1	1	1.41	1.04
2	1	1	1.16	1.16
3	1	1	1.28	1.1
4	1	1	1.5	0.9
5	NA	NA	NA	NA
6	NA	NA	NA	NA
7	1	1	1.92	1.02
8	1	1	1.27	1.0
9	1	1	1.29	1.1
10	1	3.13	1.41	1.1

Table 4.9: Comparisons between IAU_0 and IAU'_0

Problem	$W^A_{x^*}/W^{AM}_{x^*}$	$W^A_{f^*}/W^{AM}_{f^*}$	N^A_{max}/N^{AM}_{max}	t^A/t^{AM}
1	1	1	1.41	1.03
2	1	1	1	1.0
3	1	1	1.28	1.06
4	1	1	1	1
5	1	1	1	0.97
6	1	1	1.01	1.0
7	1	1	1.92	1.0
8	1	1	1.27	1.0
9	1	1	1.36	1.08
10	1	1	1.41	1.06

Table 4.10: Comparisons between IAU_1 and IAU'_1

Problem	Method	W_i	ϵ	W_{x^*}	W_{f^*}	N_{max}	t
1	IC	$2 \cdot 10^6$	10^{-1}	$1.8 \cdot 10^{-2}$	$1.9 \cdot 10^{-2}$	7	0.230
2	IC	6	10^{-1}	$9.7 \cdot 10^{-2}$	$9.6 \cdot 10^{-2}$	358	24.10
3	IC	9	10^{-5}	$7.7 \cdot 10^{-6}$	$3.5 \cdot 10^{-8}$	21	2.412
4	IC	$2 \cdot 10^3$	10^{-5}	$6.8 \cdot 10^{-6}$	$1.4 \cdot 10^{-9}$	36	1.298
5	IC	6	10^{-1}	$9.5 \cdot 10^{-2}$	$7.7 \cdot 10^{-2}$	114	8.032
6	IC	9	10^{-1}	$6.1 \cdot 10^{-5}$	10^{-1}	434	24.255
7	IC	$2 \cdot 10^6$	10^{-5}	$2.3 \cdot 10^{-6}$	$1.3 \cdot 10^{-12}$	33	4.761
8	IC	$2 \cdot 10^6$	10^{-1}	$9.6 \cdot 10^{-2}$	$9.8 \cdot 10^{-2}$	20	3.026
9	IC	6	10^{-1}	$7.1 \cdot 10^{-2}$	$1.6 \cdot 10^{-2}$	24	2.625
10	IC	3.78	10^{-5}	$2.7 \cdot 10^{-9}$	$3.7 \cdot 10^{-18}$	18	1.540

Table 4.11: Running results of ICU_0'

Problem	Method	W_i	ϵ	W_{x^*}	W_{f^*}	N_{max}	t
1	IC	$2 \cdot 10^6$	10^{-1}	$1.6 \cdot 10^{-15}$	10^{-31}	3	0.103
2	IC	6	10^{-1}	$3.9 \cdot 10^{-3}$	$2.3 \cdot 10^{-6}$	3	0.097
3	IC	9	10^{-5}	$7.7 \cdot 10^{-6}$	$3.5 \cdot 10^{-8}$	21	3.88
4	IC	$2 \cdot 10^3$	10^{-5}	$9.8 \cdot 10^{-6}$	$2.4 \cdot 10^{-9}$	19	1.095
5	IC	6	10^{-1}	$7.7 \cdot 10^{-29}$	$4.1 \cdot 10^{-56}$	3	0.189
6	IC	9	10^{-1}	$5.1 \cdot 10^{-14}$	$8.7 \cdot 10^{-11}$	7	0.286
7	IC	$2 \cdot 10^6$	10^{-5}	$9.7 \cdot 10^{-6}$	$3.8 \cdot 10^{-11}$	20	3.491
8	IC	$2 \cdot 10^6$	10^{-1}	$5.6 \cdot 10^{-2}$	$7.5 \cdot 10^{-3}$	8	2.001
9	IC	6	10^{-1}	$8.8 \cdot 10^{-2}$	$1.1 \cdot 10^{-1}$	20	3.051
10	IC	3.78	10^{-5}	$6.7 \cdot 10^{-6}$	$2.3 \cdot 10^{-11}$	17	2.121

Table 4.12: Running results of ICU_1'

Problem	$W^C_{x^*}/W^{CM}_{x^*}$	$W^C_{f^*}/W^{CM}_{f^*}$	N^C_{max}/N^{CM}_{max}	t^C/t^{CM}
1	5.6	1.05	2	1.3
2	1	1	1	0.98
3	0.96	2.77	4.5	1.98
4	1.29	1.93	1.86	1.16
5	1	1.17	1	1.03
6	1.02	1	1	0.97
7	3.22	41.5	2.6	0.99
8	0.74	5.1	1.15	0.72
9	1.28	0.75	1.71	0.98
10	$2.3 \cdot 10^3$	$3.0 \cdot 10^7$	1.56	0.94

Table 4.13: Comparisons between $\bar{I}CU_0$ and ICU'_0

Problem	$W^C_{x^*}/W^{CM}_{x^*}$	$W^C_{f^*}/W^{CM}_{f^*}$	N^C_{max}/N^{CM}_{max}	t^C/t^{CM}
1	1	1	1	0.84
2	1	1	1	0.96
3	0.96	$2.8 \cdot 10^{-3}$	4.5	2.37
4	0.85	0.75	1.8	1.36
5	1	1.12	1	0.75
6	3.73	3.56	1	0.83
7	0.73	2.18	1.95	0.87
8	1.23	1.87	1.63	0.98
9	0.85	$2.4 \cdot 10^{-2}$	1.65	1.07
10	0.93	0.61	2.29	1.36

Table 4.14: Comparisons between ICU_1 and ICU'_1

Chapter 5

Constrained Global Optimization

The constrained global optimization problem is much harder to solve than the unconstrained variety. This holds in the interval arithmetic approach as well as in other approaches. Using interval constraints, we found that the constrained optimization problem is not as much harder to solve than the unconstrained version as it is in the other approaches. W. J. Older explored in BNR-Prolog the application of interval constraints in several areas [73, 69, 65, 72, 70]. In constrained optimization, Older used the Kuhn-Tucker condition [49, 28] and demonstrated it on one problem from [29]. Recently, P. Van Hentenryck and L. Michel reported their Numerica system for global optimization in [37, 38]. They use interval constraints, therefore obtaining the same advantages over interval arithmetic as reported in [16, 84, 17]. They presented a Branch-and-Bound algorithm and discussed the performance of their system. Here, we investigate the transition from solving the unconstrained global optimization problem to the constrained one, present the characteristics of the interval constraint method, study the effect of the Fritz-John condition [36, 46] as redundant constraints,

and compare the interval arithmetic approach with the interval constraint method.

To avoid ambiguity between the "constraints" in the sense of constrained global optimization and interval constraints, we call the former "conditions", the latter "constraints".

5.1 Unconstrained vs Constrained Global Optimization

From the definition 1.1 of a global optimization problem, we obtain a *constrained global optimization problem* if the conditions 1.1 or 1.2 are present; otherwise, we get an unconstrained one. For the unconstrained optimization problem, x is still constrained to lie in the initial box. But usually the initial box is chosen to be so large as to contain all the minimizers and such that no occurrence of the global minimum is on the boundary of the box.

When we solve the constrained global optimization problem using interval constraints, the problem's defining conditions 1.1 and 1.2 are translated to constraints. From the point of view of interval constraints, constrained global optimization is the same type of problem as unconstrained one. But this does not mean that one can use the algorithm in Figure 4.2 for solving constrained global optimization problems. In this chapter we investigate the necessary modifications.

5.1.1 Existence of the Global Minimum

For an unconstrained global optimization problem, if the objective function is bounded in a given domain, then the global minimizer(s) exists in the domain. This is implied by Bolzano-Weierstrass theorem [41, 24]. In constrained global optimization, there is the additional complication of a feasible region: a minimizer must be a feasible point. If there exists at least one feasible point and the objective function is bounded, then it follows that a global minimizer exists; otherwise, the problem has no solution [29].

A constrained global optimization problem is a mathematical model of a physical problem. Given a physical problem, we usually know that it has solutions and optimal solution(s). For example, there exist designs and best design(s) for a chemical reactor design problem.

However, it is difficult to know whether there are feasible points (i.e., solutions) for a constrained global optimization problem of mathematical form. For a given point x, even though we can check whether it satisfies the inequality condition 1.1 using interval arithmetic or interval constraints, it is generally impossible to assure that x satisfies the equality condition 1.2. If we make a single rounding error in computing $q_j(x)$ for some $j = 1, \ldots, r$, we do not know whether $q_j(x)$ is zero.

5.1.2 Necessary Conditions

The unconstrained global optimization problem has the properties that at any occurrence of the global minimum:

- the gradient of the objective function (see definition 4.1) has to be zero, and

- the Hessian matrix of the objective function (see definition 4.2) has to be positive semi-definite.

These two properties are referred to as *first order necessary condition* and *second order necessary condition* respectively [28]. In the previous chapter we explored the idea of using these conditions as constraints. As shown there, the idea resulted in a dramatic improvement in solving unconstrained global optimization problems in interval constraints.

However, we can not use these two conditions for solving constrained global optimization problems. In a constrained global optimization problem, it is quite common that the only occurrences of the global minimum are on the boundary of the feasible region. At such occurrences the gradient of the objective function is typically not zero, and the Hessian matrix is not positive semi-definite. Fortunately, some first order necessary conditions for constrained global optimization were introduced [49, 28, 36, 46].

The most general necessary condition for the constrained optimization problem described in the definition 1.1 is the Fritz-John condition [36, 46]. The Kuhn-Tucker condition is also well known. It requires a type of condition called *regularity condition.* The regularity condition is rather troublesome to verify in practice.

Fritz-John Condition. Assume that the objective function f, constraint functions p_i $(i = 1, \ldots, m)$ and q_j $(j = 1, \ldots, r)$ in the problem given by the definition 1.1 are differentiable. If a point x is a local minimizer of the problem, then there exist $u_i \geq 0, u_i \in R$ $(i = 0, 1, \ldots, m)$ and $v_j \in R$ $(j = 1, \ldots, r)$ such that

$$F(t) = (F_1(t), F_2(t), \ldots, F_{m+r+2}(t))^T = 0 \tag{5.1}$$

where

$$F_1(t) = u_0 \nabla f(x) + \sum_{i=1}^{m} u_i \nabla p_i(x) + \sum_{j=1}^{r} v_j \nabla q_j(x), \tag{5.2}$$

$$F_2(t) = \sum_{i=0}^{m} u_i + \sum_{j=1}^{r} v_j - 1, \tag{5.3}$$

$$F_{i+2}(t) = u_i p_i(x) \ (i = 1, \ldots, m), \tag{5.4}$$

$$F_{m+j+2}(t) = v_j q_j(x) \ (j = 1, \ldots, r), \tag{5.5}$$

$t = (x_1, \ldots, x_n, u_0, u_1, \ldots, u_m, v_1, \ldots, v_r)^T$ and ∇ is the gradient operator.

The equation $F_2(t) = 0$, i.e.,

$$\sum_{i=0}^{m} u_i + \sum_{j=1}^{r} v_j - 1 = 0 \tag{5.6}$$

is called the *normalization condition.* The components u_i $(i = 0, \ldots, m)$, v_j $(j = 1, \ldots, r)$ are called the *Lagrangian multipliers.* The function whose gradient with respect to x occurs in the equation 5.2

$$\Psi(x, u, v) = u_0 f(x) + \sum_{i=1}^{m} u_i p_i(x) + \sum_{j=1}^{r} v_j q_j(x) \tag{5.7}$$

is called the *generalized Lagrangian function* of the problem given by the definition 1.1.

For an unconstrained optimization problem, the Fritz-John condition reduces to

$$\nabla f(x) = 0,$$

which is the first-order necessary condition for the unconstrained optimization problem.

Kuhn-Tucker Condition. Before describing the Kuhn-Tucker condition, we present the definition of the regularity condition.

Definition 5.1: [74]

Let x be a feasible point of the problem given by the definition 1.1. Let $A(x) = \{i \mid p_i(x) = 0, i \in \{1,\ldots,m\}\}$ be the so-called *active index set.* The point x is said to satisfy the *regularity condition* if the set of gradients $p_i'(x)$, $q_j'(x)$ with $i \in A(x)$, $j = 1,\ldots,r$ is linearly independent.

Assume that the objective function f, constraint functions p_i $(i = 1,\ldots,m)$ and q_j $(j = 1,\ldots,r)$ in the problem given by the definition 1.1 are differentiable. The Kuhn-Tucker condition says that if a point x is a local minimizer of the problem and satisfies the regularity condition, then there exist $u_i \geq 0, u_i \in R$ $(i = 1,\ldots,m)$ and $v_j \in R$ $(j = 1,\ldots,r)$ such that

$$K(t) = (K_1(t), K_2(t), \ldots, K_{m+r+1}(t))^T = 0 \tag{5.8}$$

where

$$K_1(t) = \nabla f(x) + \sum_{i=1}^{m} u_i \nabla p_i(x) + \sum_{j=1}^{r} v_j \nabla q_j(x), \tag{5.9}$$

$$K_{i+1}(t) = u_i p_i(x) \ (i = 1,\ldots,m), \tag{5.10}$$

$$K_{m+j+1}(t) = v_j q_j(x) \ (j = 1,\ldots,r), \tag{5.11}$$

$t = (x_1,\ldots,x_n,u_1,\ldots,u_m,v_1,\ldots,v_r)^T$ and ∇ is the gradient operator.

The function whose gradient with respect to x occurs in the equation 5.9

$$L(x,u,v) = f(x) + \sum_{i=1}^{m} u_i p_i(x) + \sum_{j=1}^{r} v_j q_j(x) \tag{5.12}$$

is called the *Lagrangian function* of the problem given by the definition 1.1.

To solve the equality system corresponding to the Fritz-John condition or the Kuhn-Tucker condition is very hard, since the equalities in the system are nonlinear and quite complex.

E. R. Hansen explored the Fritz-John condition in his interval arithmetic method for constrained global optimization through using the interval Newton method to solve the equality system [36]. But computational results are lacking [36, 46, 47].

In an interval constraint method for the constrained optimization, we do not solve the Fritz-John or Kuhn-Tucker conditions, we just add them in the form of their definitions into an interval constraint system. This will be discussed in detail in the following.

5.2 Overview of Interval Arithmetic Methods

Hansen and Sengupta [36] were the first to use interval arithmetic to solve the constrained global optimization problems with inequality conditions. Their algorithm is the combination of the Branch-and-Bound with several tests such as the midpoint test. The value of the objective function f at a feasible point in the original domain is an upper bound of f^*. In the midpoint test, Hansen used the lowest upper bound of f^* found so far.

For the constrained global optimization problem with equality conditions, there are difficulties in determining whether a given point is feasible, thus it is hard to find an upper bound of f^*. Because of rounding errors, it is generally impossible to verify that a point c satisfies an equality condition. To overcome the difficulties, Hansen and Walster [74, 32] suggested that Moore's existence theorem [60] should be applied to the constrained global optimization with the equality condition to check whether a given box contains a feasible point. Bao and Rokne [74, 4] generalized Moore's existence theorem to include the equality condition.

The algorithm for constrained global optimization is a modification of the one in Figure 4.2 for unconstrained global optimization. One important point is that $ub(f(mid(X)))$ in line 2, $ub(f(m_1))$ and $ub(f(m_2))$ in line 12 can no longer be counted on to provide an upper bound for f^*, as $mid(X)$, $m_1 = mid(B_1)$ or $m_2 = mid(B_2)$ may not be a feasible point in constrained global optimization. Instead we define $f_b(B)$ and use it in the place of $ub(f(mid(B)))$ to find an upper bound for f^*.

Another important observation is that the necessary condition for unconstrained global optimization is different from that for constrained one. This suggests that some of tests used for unconstrained global optimization can not be used in the constrained case.

In the following, we define $f_b(B)$ to find an upper bound of f^*, which is followed by the explanation of several tests that can be used for constrained global optimization.

Finding an Upper Bound. For the constrained global optimization problem only with inequality conditions, the value of the objective function f at a point c in the original domain is an upper bound u of the global minimum f^* if the point c satisfies

$$ub(p_i(c)) \leq 0 \quad \text{for all } i = 1, \ldots, m. \tag{5.13}$$

Formally, for a given box B, we can obtain u by using the following formula:

$$u = f_b(B) = \begin{cases} ub(f(c)) & \text{if } c \in B \text{ and } ub(p_i(c)) \leq 0 \quad \text{for } i = 1, \ldots, m \\ +\infty & \text{otherwise.} \end{cases} \tag{5.14}$$

Hansen calls the point that holds for the condition 5.13 a *certainly feasible* point. Hansen set an upper bound f^*_{ub} of f^* equal to the smallest upper bound found in this way and used it in the midpoint test.

However, it generally impossible to assure that the point c satisfies

$$q_j(c) = 0 \quad \text{for all } j = 1, \ldots, r \tag{5.15}$$

i.e.,

$$lb(q_j(c)) = ub(q_j(c)) = 0 \quad \text{for all } j = 1, \ldots, r \tag{5.16}$$

because of rounding errors. If we make a single rounding error in computing $q_j(c)$ for some $j = 1, \ldots, r$, we do not know whether $q_j(c)$ is zero or not. Thus the value of f at the point c can not be used as an upper bound of f^* for the constrained global optimization problem with the equality condition.

One way to obtain an upper bound of f^* is due to Hansen-Walster[32] and Bao-Rokne[4]. Suppose that we have proved that a feasible point exists in a box B. We compute $f(B)$, getting the result $[lb(f(B)), ub(f(B))]$. Then $ub(f(B))$ is an upper bound u of f^*. Formally, for a given box B, u is obtained according to the following formula:

$$u = f_b(B) = \begin{cases} ub(f(B)) & \text{if B contains a feasible point} \\ +\infty & \text{otherwise.} \end{cases} \tag{5.17}$$

In order to prove that a given box B contains a feasible point, we proceed as follows:

(1) Without restriction of generality, let

$$\begin{aligned} &lb(p_i(B)) \leq 0 < ub(p_i(B)) && \text{for } i = 1, \ldots, m'(\leq m) \\ &0 \in q_j(B) \wedge q_j(B) \neq 0 && \text{for } j = 1, \ldots, r'(\leq r) \\ &s := m' + r'. \end{aligned} \tag{5.18}$$

where m' is the number of inequalities we can not determine whether they hold over the given box B, r' is the number of equalities we can not determine if

they hold over B. The remaining conditions should not be violated:

$$\begin{aligned} ub(p_i(B)) &\leq 0 \qquad\qquad \text{for } i = m'+1,\ldots,m \\ lb(q_j(B)) = ub(q_j(B)) &= 0 \quad \text{for } j = r'+1,\ldots,r. \end{aligned} \tag{5.19}$$

(2) Select s components of B (for simplicity the first s) and build a box

$$Z = X_1 \times \cdots \times X_s.$$

Let $c = (c_1,\ldots,c_n)$ be the midpoint of B. Let

$$\begin{aligned} h_i(z) &= p_i(z, c_{s+1},\ldots,c_n) \quad i = 1,\ldots,m' \\ h_{m'+j}(z) &= q_j(z, c_{s+1},\ldots,c_n) \quad j = 1,\ldots,r', \end{aligned} \tag{5.20}$$

where $z = (x_1,\ldots,x_s)^T$. Then $h = (h_1,\ldots,h_s)^T$ is an s-dimensional vector-value function.

(3) Apply one step of the interval Newton method to solve $h(z) = 0$. Suppose that Z' is the result of solving $h(z) = 0$ by using one step of the interval Newton method. The property of the interval Newton method guarantees that there is a unique solution to $h(z) = 0$ within Z if Z' is contained in the interior of Z and not empty. So if $Z' \subset Z$ and $Z' \neq \emptyset$, there exists a feasible point $x' = (x'_1,\ldots,x'_s, c_{s+1},\ldots,c_n)^T$ in B.

Here we integrate formulas 5.14 and 5.17 into one formula for finding an upper bound u of f^* as follows:

$$u = f_b(B) = \begin{cases} ub(f(c)) & \text{if no equalities and } ub(p_i(c)) \leq 0 \text{ for all } i = 1,\ldots,m \\ ub(f(B)) & \text{if equalities exist and applying one step of the interval} \\ & \quad \text{Newton method to solve } h(z) = 0 \text{ for } z \in Z \text{ gives the} \\ & \quad \text{result } Z' \subset Z \text{ and } Z' \neq \emptyset \\ +\infty & \text{otherwise} \end{cases} \tag{5.21}$$

where $c = (c_1, \ldots, c_n)^T$ is the midpoint of $B = X_1 \times \cdots \times X_n$, $Z = X_1 \times \cdots \times X_s$, h is an s-dimensional vector-value function whose components are defined in formula 5.20.

It is possible that for every sub-box B split in a Branch-and-Bound algorithm we can not conclude that there is a feasible point in B. In this case, we do not obtain an upper bound of f^*. This makes the constrained global optimization problem hard in interval arithmetic.

Inequality and Equality Test. Given a box B, if the problem's defining conditions (i.e., the inequality condition and the equality condition) can not be satisfied anywhere in B, then B is rejected. In interval arithmetic, $p_i(B)$ $(i = 1, \ldots, m)$ and $q_j(B)$ $(j = 1, \ldots, r)$ are computed and the results are denoted by

$$p_i(B) = [lb(p_i(B)), ub(p_i(B))]$$

and

$$q_j(B) = [lb(q_j(B)), ub(q_j(B))].$$

If we can determine that

$$lb(p_i(B)) \leq 0 \quad \text{for all } i = 1, \ldots, m \tag{5.22}$$

does not hold, then $p_i(x) > 0$ for all $x \in B$ for some $i \in \{1, \ldots, m\}$ (i.e., there is no point $x \in B$ such that $p_i(x) \leq 0$) and B is rejected. We call the inequality 5.22 the *inequality test.*

If we can determine that

$$0 \in [lb(q_j(B)), ub(q_j(B))] \quad \text{for all } j = 1, \ldots, r, \tag{5.23}$$

i.e.,

$$lb(q_j(B)) \leq 0 \text{ and } 0 \leq ub(q_j(B)) \quad \text{for all } j = 1, \ldots, r, \tag{5.24}$$

does not hold, then there is no point $x \in B$ such that $q_j(x) = 0$ for some $j \in \{1, \ldots, r\}$ and B is rejected. We call the inequality 5.24 the *equality test.*

Note: Hansen treated each equality condition $q_j(x) = 0$ $(j = 1, \ldots, r)$ as two inequality conditions

$$q_j(x) \leq 0 \text{ and } q_j(x) \geq 0.$$

If $lb(q_j(B)) \leq 0$ and $ub(q_j(B)) \geq 0$ does not hold (i.e., $lb(q_j(B)) > 0$ or $ub(q_j(B)) < 0$), then the inequality conditions do not hold and B is rejected. This is equivalent to treating $q_j(x) = 0$ as an equality condition.

Fritz-John Test. Hansen solves the Fritz-John condition $F(t) = 0$ using the interval Newton method. Suppose that we seek the solution of $F(t) = 0$ in a given box B and the interval Newton method produces the result B'. If

$$B' \cap B \neq \emptyset \tag{5.25}$$

does not hold, then there is no solution in B and B is rejected. The inequality 5.25 is called the *Fritz-John test.*

5.3 Branch-and-Bound for Constrained Optimization

The Branch-and-Bound algorithm for Constrained Global Optimization, which is denoted by *BBCGO*, is described in Figure 5.1. The algorithm is applicable to both interval arithmetic and interval constraints. We obtain this algorithm from the Branch-and-Bound algorithm shown in Figure 4.2 for Unconstrained Global Optimization

(denoted by *BBUGO* in the following) by making a few changes. The underlined parts in Figure 5.1 are the main changes.

5.3.1 Methods for Finding An Upper Bound

In *BBUGO*, we use

$$f^*_{ub} := ub(f(mid(X))), \quad u_1 := ub(f(mid(B_1))), \quad u_2 := ub(f(mid(B_2)))$$

to obtain an upper bound of f^* over boxes X, B_1 and B_2 respectively. For an unconstrained global optimization problem, the value of the objective function f at an arbitrary point of a box is an upper bound of f^* over the box. We choose the midpoint there.

In *BBCGO*, we use

$$f^*_{ub} := f_b(X), \quad u_1 := f_b(B_1), \quad u_2 := f_b(B_2)$$

to obtain an upper bound of f^* over boxes X, B_1 and B_2 respectively, where one definition of f_b is given in Formula 5.21, another definition will be discussed in the following section. For a constrained global optimization problem, the value of the objective function f at an arbitrary point of a box is only usable if that point is feasible.

5.3.2 Conditions for Rejecting Boxes

In *BBUGO*, we can always use condition 4.2 (i.e., the midpoint condition) for rejecting boxes. If the function f is differentiable, conditions 4.4 and 4.9 (i.e., the stationarity condition and the linear condition) can be used. If f is twice differentiable, we can use

1: **input:** $f, p_i(i = 1, \ldots, m), q_j(j = 1, \ldots, r), X, \epsilon_X, \epsilon_F, \epsilon_P, \epsilon_Q$;
2: $f^*_{ub} := f_b(X)$;
3: $L := [< X, lb(f(X)), f^*_{ub} >]$;
4: $A := \emptyset$;
5: **while** $L \neq \emptyset$ **do**
6: remove $< B, l, u >$ from L;
7: **if** $|B| \leq \epsilon_X \wedge |f(B)| \leq \epsilon_F \wedge ub(p_i(B)) \leq \epsilon_P \wedge |q_j(B)| \leq \epsilon_Q \wedge 0 \in q_j(B)$ **then**
8: add $< B, l, u >$ to A
9: **else if** conditions$(< B, l, u >, f^*_{ub})$ **then**
10: partition B into B_1 and B_2 with midpoints m_1 and m_2;
11: $l_1 := lb(f(B_1))$; $l_2 := lb(f(B_2))$;
12: $u_1 := f_b(B_1)$; $u_2 := f_b(B_2)$;
13: $f^*_{ub} := min(f^*_{ub}, u_1, u_2)$;
14: add $< B_1, l_1, u_1 >$ and $< B_2, l_2, u_2 >$ to L;
15: **end-if**
16: **end-while**
17: $f^*_{lb} := \min\{l_i \mid < B_i, l_i, u_i > \in A\}$;
18: **if** $f^*_{ub} = +\infty$ **then**
19: $f^*_{ub} := \max\{ub(f(B_i)) \mid < B_i, l_i, u_i > \in A\}$;
20: **output:** f^* is in $[f^*_{lb}, f^*_{ub}]$ if there exists a feasible point in X, A;
21: **else**
22: **output:** There is a feasible point in X, f^* is in $[f^*_{lb}, f^*_{ub}]$, A;
23: **end-if**

Figure 5.1: Branch-and-Bound algorithm for constrained optimization

conditions 4.6, 4.11 and 4.14 (i.e., the convexity condition, the quadratic condition and the Newton condition).

In *BBCGO*, the problem's defining conditions 1.1 and 1.2 can be used for rejecting boxes. Condition 4.2 (i.e., the midpoint condition) is always applicable. We can also use conditions 4.9, 4.11 (i.e., the linear condition, the quadratic condition) and the Newton condition if the function f is twice differentiable. Although we can not use conditions 4.4 and/or 4.6 (i.e., the stationarity condition and/or the convexity condition) in *BBCGO*, we can use condition 5.1 (i.e., the Fritz-John condition).

5.3.3 Conditions for the Answer List

In *BBUGO* shown in Figure 4.2, a triple $< B, l, u >$ is put into the answer list A if it satisfies the following condition:

$$|B| \leq \epsilon_X \wedge (u - l) \leq \epsilon_F. \tag{5.26}$$

This condition guarantees that the box B of every triple $< B, l, u >$ in the answer list A is small enough and the value of the function f at the midpoint of B is close enough to the real minimum f^*. However, this condition is not suitable for constrained global optimization. It does not consider the problem's defining conditions. Some conditions should be added for considering the problem's defining conditions. Furthermore, it is hard for the triple $< B, l, u >$ to satisfy the condition

$$(u - l) \leq \epsilon_F, \tag{5.27}$$

since u is $+\infty$ in general. It is better to change this condition.

Adding More Conditions for the Answer List. Suppose that $\epsilon_P > 0$ and $\epsilon_Q > 0$ are the tolerances for the inequalities and equalities of the problem's defining conditions respectively. We usually want to assure that any point x in the box B of every triple $< B, l, u >$ in the answer list satisfies the conditions

$$p_i(x) \leq \epsilon_P \quad \text{for } i = 1, \ldots, m \tag{5.28}$$

and

$$-\epsilon_Q \leq q_j(x) \leq \epsilon_Q \quad \text{for } j = 1, \ldots, r. \tag{5.29}$$

We can achieve this by adding the conditions

$$ub(p_i(B)) \leq \epsilon_P \quad \text{for } i = 1, \ldots, m \tag{5.30}$$

and

$$|q_j(B)| \leq \epsilon_Q \wedge 0 \in q_j(B) \quad \text{for } j = 1, \ldots, r \tag{5.31}$$

into condition 5.26, i.e., by using conditions 5.26, 5.30 and 5.31 instead of just using condition 5.26.

Condition 5.30 guarantees that any point x in B satisfies condition 5.28. This indicates that $p_i(x) \leq 0$ holds within the given tolerance ϵ_P for $i = 1, \ldots, m$. Condition 5.31 makes sure that any point x in B satisfies condition 5.29. This means that $q_j(x) = 0$ holds within the given tolerance ϵ_Q for $j = 1, \ldots, r$.

Replacing the Condition $(u - l) \leq \epsilon_F$. For a triple $< B, l, u >$ in *BBUGO*, an upper bound u of the minimum f^* over the box B is always defined. The value of the function f at an arbitrary point of B is defined and is an upper bound of f^*. We choose the midpoint of B. However, this is not the case for constrained global optimization. An arbitrary point of B is not necessarily a feasible point, neither is

the midpoint of B. Thus u is not defined in general. Therefore, condition 5.27 may not be satisfied. One suitable replacement of condition 5.27 is

$$|f(B)| \leq \epsilon_F. \tag{5.32}$$

After the replacement and the addition of conditions 5.30 and 5.31, we have the following condition in *BBCGO* for adding a triple $< B, l, u >$ into the answer list A:

$$|B| \leq \epsilon_X \wedge |f(B)| \leq \epsilon_F \wedge ub(p_i(B)) \leq \epsilon_P \wedge |q_j(B)| \leq \epsilon_Q \wedge 0 \in q_j(B) \tag{5.33}$$

where $i = 1, \ldots, m$ and $j = 1, \ldots, r$.

5.3.4 Input and Output of the Algorithm

The input and output of the algorithm for constrained global optimization contain more information than those of the algorithm for unconstrained one.

Input. In addition to f, p_i $(i = 1, \ldots, m)$ and q_j $(j = 1, \ldots, r)$, the input of *BBCGO* includes the tolerances ϵ_P and ϵ_Q. ϵ_P and ϵ_Q are for the inequalities and equalities of the problem's defining conditions respectively.

Output. The **output** statement in *BBUGO* is replaced by the following **if-then-else** statement:

if $f^*_{ub} = +\infty$ **then**

$f^*_{ub} :=$ $\max\{ub(f(B_i)) \mid < B_i, l_i, u_i > \in A\}$;

output: f^* is in $[f^*_{lb}, f^*_{ub}]$ if there exists a feasible point in X, A;

else

output: There is a feasible point in X, f^* is in $[f^*_{lb}, f^*_{ub}]$, A;

end-if .

It is possible that we can not assure that there exists a feasible point during any iteration of the algorithm. In this case, f^*_{ub} is initialized as $+\infty$ and keeps being $+\infty$ until the end of the algorithm. We can only make sure that f^* is between

$$f^*_{lb} = \min\{l_i \mid < B_i, l_i, u_i > \in A\}$$

and

$$f^*_{ub} = \max\{ub(f(B_i)) \mid < B_i, l_i, u_i > \in A\}$$

if there exists a feasible point in the original domain X. The problem we solve using the algorithm is the mathematical model of a physical problem. In general, we know that the physical problem has solutions. So we can make sure that f^* is between f^*_{lb} and f^*_{ub}.

In the case that at least one feasible point exists, we can guarantee that f^* is between f^*_{lb} and f^*_{ub}.

5.4 An Interval Constraint Method

In the previous section, we described the Branch-and-Bound algorithm for constrained global optimization. The performance of the algorithm depends on the effectiveness of the implementation of "conditions($< B, l, u >, f^*_{ub}$)" and "$f_b(B)$". If we use the mid-point test, inequality test, equality test and the Fritz-John test, we obtain Hansen's interval arithmetic method for constrained global optimization. Since all the tests use

interval arithmetic, we call Hansen's algorithm an interval arithmetic version of the Branch-and-Bound algorithm for constrained global optimization, denoted by IAC.

In this section, we demonstrate how to translate all the interval arithmetic computations in "conditions($< B, l, u >, f^*_{ub}$)" and "$f_b(B)$" into an interval constraint processing task. Thus we can obtain an interval constraint version of the Branch-and-Bound algorithm for constrained global optimization, which is denoted by ICC in the following.

5.4.1 Finding An Upper Bound in Interval Constraints

In Branch-and-Bound, we need an upper bound of f^* over a given box. In unconstrained global optimization, the upper bound of the interval obtained by evaluating the objective function over the box is an upper bound for the minimum over that box. Another upper bound is the value of the objective function at an arbitrary point in the box. The latter is usually much better.

In constrained global optimization with Branch-and-Bound, an upper bound is harder to obtain, and this makes constrained optimization harder also in interval constraints. An upper bound of the interval for the objective function evaluated over the box in the presence of the constraints is easily computed in interval constraints. However, it is only valid as an upper bound in Branch-and-Bound if there exists at least one feasible point in the box. This is not known. Similarly, an upper bound obtained from evaluating the objective function at an arbitrary point is only usable if that point is feasible.

In section 5.2, we described the interval arithmetic methods for finding an upper bound u of f^* for constrained global optimization. Through translating interval

arithmetic computations in formula 5.21 into an interval constraint processing task, we obtain an interval constraint method for finding u. This method is formulated as:

$$u = f_b(B) = \begin{cases} ub(f(c)) & \text{if no equalities and } ub(p_i(c)) \leq 0 \text{ for all } i = 1, \ldots, m \\ ub(f(B)) & \text{if equalities exist and applying the consistency} \\ & \quad \text{algorithm to the constraint system} \\ & \quad \text{“}z \in Z, h(c') + J(z, Z)(z - c') = 0\text{” shrinks } Z \text{ but} \\ & \quad \text{does not produce any failure state.} \\ +\infty & \text{otherwise} \end{cases} \tag{5.34}$$

where c, B, Z and h are the same as in formula 5.21, $c' = (c_1, \ldots, c_s)^T$ is the midpoint of Z, and $J(z, Z)$ is the Jacobian matrix of h over Z.

We have three different cases for computing u:

(1) There is a feasible point c in B. In this case, we compute $ub(f(c))$ as the value of u.

(2) The box B contains a feasible point. We compute $ub(f(B))$ as the value of u.

(3) No feasible point in B. We set u to $+\infty$.

Since interval constraints is not less effective than interval arithmetic, there is not less chance to find a real upper bound u of f^* using the interval constraint method formulated in 5.34 than using interval arithmetic method formulated in 5.21. The computational results in Table 5.2 suggest that interval constraints is more effective.

5.4.2 Using Conditions as Constraints

In constrained global optimization, interval constraints are especially interesting. All the problem's defining conditions, both inequality 1.1 and equality 1.2, can be directly used as constraints. In addition, the midpoint condition 4.2 can be used as constraints. Furthermore, we can also use the Fritz-John condition or the Kuhn-Tucker condition as constraints if the function f, p_i $(i = 1, \ldots, m)$ and q_j $(j = 1, \ldots, r)$ are differentiable.

For a given box B, an upper bound f^*_{ub} of f^* and the conditions used as constraints, we have the following interval constraint system (note that we use the Fritz-John condition here):

$$
\begin{array}{ll}
x \in B & \\
y = f(x), y \leq f^*_{ub} & \\
p_i(x) \leq 0 & (i = 1, \ldots, m) \\
q_j(x) = 0 & (j = 1, \ldots, r) \\
u_i \in [0, +\infty] & (i = 0, 1, \ldots, m) \\
v_j \in [-\infty, +\infty] & (j = 1, \ldots, r) \\
u_0 \nabla f(x) + \sum_{i=1}^{m} u_i \nabla p_i(x) + \sum_{j=1}^{r} v_j \nabla q_j(x) = 0 & \\
u_i p_i(x) = 0 & (i = 1, \ldots, m) \\
v_j q_j(x) = 0 & (j = 1, \ldots, r) \\
\sum_{i=0}^{m} u_i + \sum_{j=1}^{r} v_j - 1 = 0 &
\end{array}
\tag{5.35}
$$

where the second line is the midpoint condition, the third and fourth line are the problem's defining conditions (i.e., the inequality condition and equality condition), and the rest is the Fritz-John condition. If applying the consistency algorithm to the constraint system 5.35 results in a failure state (i.e., produces an empty box),

then the box B can be rejected. It is also possible that the output of the consistency algorithm is a shrunk non-empty box B'. In this case, some of the conditions used can not hold for $B - B'$, so that we conclude that no point in $B - B'$ satisfies all the conditions used as constraints. Note that in this case we can not conclude that every point $x \in B'$ satisfies all the conditions.

In interval arithmetic the problem's defining conditions 1.1 and 1.2 can not be used directly. The Fritz-John condition or the Kuhn-Tucker condition is essential, because using either of them is the only way in which the problem's defining conditions can be indirectly taken into account.

A novel feature of interval constraint approach, and perhaps first demonstrated in [16, 84, 17], is that it allows one to select from a hierarchy of redundant conditions. The conditions on the minimization are completely determined by the problem's defining conditions 1.1 and 1.2. In principle it is possible in interval constraints to only include conditions 1.1 and 1.2.

5.5 An Implementation of the Interval Constraint Method

We have implemented in BNR-Prolog the interval constraint method for constrained global optimization. Figure 5.2 shows the main part of the source code. We omit the definitions of *small*, *fb*, *partition* and *min*. The user should provide in BNR-Prolog the definitions of f, p_i $(i = 1, \ldots, m)$ and q_j $(j = 1, \ldots, r)$. The user should also give the definitions of f', p_i' $(i = 1, \ldots, m)$ and q_j' $(j = 1, \ldots, r)$ if f, p_i $(i = 1, \ldots, m)$ and

q_j $(j = 1, \ldots, r)$ are differentiable and she/he wants to use the Fritz-John condition as constraints.

For example, suppose that we want to find the minimum value of the following function

$$\begin{aligned} & f(x_1, x_2) = (x_1 - 2)^2 + (x_2 - 1)^2 \qquad \text{subject to} \\ & p_1(x_1, x_2) = x_1^2 - x_2 \\ & p_2(x_1, x_2) = x_1 + x_2 - 2. \end{aligned} \tag{5.36}$$

We should define the relations f, p_1 and p_2 in BNR-Prolog as follows:

```
f([X1,X2],Y) :- Y is (X1 - 2)**2 + (X2 - 1)**2.
p1([X1,X2],P1) :- P1 is X1**2 - X2.
p2([X1,X2],P2) :- P2 is X1 + X2 - 2.
```

Similarly we can define f', p_i' $(i = 1, \ldots, m)$ and q_j' $(j = 1, \ldots, r)$ in BNR-Prolog.

Here we explain the relations defined in the program for the *ICC* that are different from those defined for *ICU*.

The formula "$small(B, Y)$" holds if $|B| \leq \epsilon_X$, $|Y| \leq \epsilon_F$, $ub(p_i(B)) \leq \epsilon_P$ (for all $i = 1, \ldots, m$), $|q_j(B)| \leq \epsilon_Q$ and $0 \in q_j(B)$ (for all $j = 1, \ldots, r$).

The formula "$fb(B, U)$" holds if the formula 5.34 holds.

The formula "$conditions([B, Y], Fub)$" holds if applying the consistency algorithm to the constraint system 5.35 does not produce any failure state.

```
g_min([],Fub,A,A) :- !.          % L = [], terminate, Answers are in A
g_min([[B,Y,U]|L0],Fub,A0,A) :-% remove [B,Y,U] from L=[[B,Y,U]|L0]
   small(B,Y), !,                % |B|=<Ex, |Y|=<Ey, ub(pi(B))=<Ep,
                                 % |qj(B)|=<Eq, and  0 in qj(B)
   insert([B,Y,U],A0,A1),        % insert [B,Y,U] into A0 and get A1
   g_min(L0,Fub,A1,A).
g_min([[B,Y,U]|L0],Fub,A0,A) :-
   conditions([B,Y],Fub), !,     % conditions may be satisfied
   partition(B,B1,B2),           % partition box B into B1 and B2
   f(B1,Y1), fb(B1,U1),          % Y1=f(B1),U1=ub(fb(m1)),m1=midpoint(B1)
   f(B2,Y2), fb(B2,U2),          % Y2=f(B2),U2=ub(fb(m2)),m2=midpoint(B2)
   min([Fub,U1,U2],Fub1),        % Fub1=min(Fub,U1,U2)
   insert([B1,Y1,U1],L0,L1),     % insert [B1,Y1,U1] and [B2,Y2,U2]
   insert([B2,Y2,U2],L1,L2),     % into L0 and get L2
   g_min(L2,Fub1,A0,A).
g_min([[B,Y,U]|L0],Fub,A0,A) :-% conditions can not be satisfied
   g_min(L0,Fub,A0,A).           % reject [B,Y,U] from [[B,Y,U]|L0]

conditions([B,Y],Fub) :-         %   constraints for:
   Y =< Fub,                                % condition f(x) <= Fub
   p1(B,P1),P1=<0, ..., pm(B,Pm),Pm=<0, % inequalities
   q1(B,Q1),Q1==0, ..., qr(B,Qr),Qr==0, % equalities
   fritz_john_condition(B).              % Fritz-John condition

insert([B,Y,U],[],[[B,Y,U]]) :- !.
insert([B,Y,U],[[B1,Y1,U1]|T],[[B,Y,U],[B1,Y1,U1]|T]) :-
   range(Y, [L,_]), range(Y1, [L1,_]), L =< L1, !.
insert([B,Y,U],[[B1,Y1,U1]|T],[[B1,Y1,U1]|R]) :- insert([B,Y,U],T,R).
```

Figure 5.2: Code of interval constraint method for constrained optimization

5.6 Test Results

In addition to conditions 1.1 and 1.2 (i.e., the problem's defining conditions), there are several conditions that are redundant, but are extremely effective in reducing the required amount of computation. Condition 4.2 (i.e., the midpoint condition) and condition 5.1 (i.e., the Fritz-John condition) are independent of each other. In our first series of tests (ICC_1 in Table 5.2) we only use conditions 1.1, 1.2 and 4.2. The novelty here is that we take the problem's defining conditions into account without using the Fritz-John condition.

In ICC_2 of Table 5.2, we improve on ICC_1 by *adding* the Fritz-John condition. As one can see, this gives a considerable improvement in performance in two of the three test problems.

Since our results give guaranteed inclusions, it only makes sense to compare performance with the methods that give the same guarantees. As far as we know, these can only be found in [74, 36, 37, 38]. Although E. R. Hansen, in [36], gives a copious variety of test results on unconstrained optimization, he gives the test result for only one problem on constrained optimization, which we include in Table 5.1. Even for these few problems, the performance measures given in [74, 36] are spotty; see the open spaces in Table 5.2.

Although Van Hentenryck and Michel reported the performance of their Numerica system in [37, 38], they only give timings and numbers of splits. Comparisons are hard to make from [37, 38]. The number of splits have the advantage of being machine-independent. However, they do not report the splits performed in narrowing by means of Newton's method and an inequality ("internal splits").

Problem	Objective Function and Conditions	Domain
1	$f(x) = 0.1(x_1^2 + x_2^2)$ $p_1(x) = 2\sin(2\pi x_2) - \sin(4\pi x_1)$	$[-1,1] \times [-1,1]$
2	$f(x) = (x_1 - 2)^2 + (x_2 - 1)^2$ $p_1(x) = x_1^2 - x_2$ $p_2(x) = x_1 + x_2 - 2$	$R \times R$
3	$f(x) = 12x_1^2 - 6.3x_1^4 + x_1^6 + 6x_1x_2 + 6x_2^2$ $p_1(x) = 1 - 16x_1^2 - 25x_2^2$ $p_2(x) = 13x_1^3 - 145x_1 + 85x_2 - 400$ $p_3(x) = x_1x_2 - 4$	$[-2,4] \times [-2,4]$

Table 5.1: Test problems for constrained global optimization

We also successfully ran our algorithm on many of the problems with three or fewer variables in [29, 42, 75].

Table 5.2 shows the performance figures from [74, 36], as far as given, (in the part labeled IAC, which is short for Interval Arithmetic method for Constrained global optimization) and compares with two versions of our algorithm (in the parts labeled ICC_1 and ICC_2; see above for their differences).

Here is an explanation of the symbols used in the tables.

W_i Width of initial box.

ϵ Intended tolerance for box size, function width, inequalities and equalities.

W_{x^*} Width of final box for an occurrence x^*.

W_{f^*} Width of bound on the global minimum f^*.

N_i Number of iterations of an algorithm till termination.

N_f Total number of function evaluations.

Each of these symbols with superscript A represents that the corresponding results are from running the interval Arithmetic version of an algorithm, that with superscript C represents that the corresponding results are from running the interval Constraint version of the algorithm.

From Table 5.2, one can see that the interval constraint algorithm using conditions 1.1, 1.2, 4.2 and 5.1 as constraints achieves a factor of 4 to 58 in the number of iterations on the constrained global optimization test problems and a factor of 23 to 76 in the number of function evaluations over the interval arithmetic methods. Moreover, the final boxes for x^* and f^* produced by our interval constraint algorithm are smaller.

Comparing the running results of ICC_1 and ICC_2, we find that using the Fritz-John conditions as redundant constraints gives a considerable improvement in performance in two of the three test problems. For these two problems, ICC_2 achieves a factor of 10 to 18 in the number of iterations and a factor of 9 to 34 in the number of function evaluations over ICC_1.

5.7 Discussion

We have compared our algorithm with the one of Hansen. He uses Interval Newton method to solve the Fritz-John condition. Conventionally, some form of Newton method has to be used to solve this condition. We only use the condition in its definitional form to prune the Branch-and-Bound search. In our approach, the use of Inter-

		Problem 1	Problem 2	Problem 3
Input	W_i	2	$+\infty$	6
	ϵ	10^{-4}	10^{-6}	10^{-2}
IAC	N_i^A	175		72
	N_f^A	686	252	
	$W_{x^*}^A$	10^{-4}	10^{-6}	10^{-2}
	$W_{f^*}^A$	10^{-10}	10^{-6}	10^{-2}
ICC_1	$N_i^{C_1}$	3	51	335
	$N_f^{C_1}$	9	101	995
	$W_{x^*}^C$	0	10^{-7}	10^{-3}
	$W_{f^*}^C$	0	10^{-7}	10^{-3}
ICC_2	$N_i^{C_2}$	3	5	19
	$N_f^{C_2}$	9	11	29
	$W_{x^*}^C$	0	10^{-16}	10^{-5}
	$W_{f^*}^C$	0	10^{-16}	10^{-5}
Comparison	$\frac{N_i^A}{N_i^{C_2}}$	58		4
	$\frac{N_f^A}{N_f^{C_2}}$	76	23	
	$\frac{N_i^{C_1}}{N_i^{C_2}}$	1	10	18
	$\frac{N_f^{C_1}}{N_f^{C_2}}$	1	9	34

Table 5.2: Running results of IAC and ICC

val Newton amounts to the addition of constraints that are redundant with respect to the definition of the Fritz-John condition, which is itself redundant to conditions 1.1 and 1.2. So far experience shows that adding additional redundant conditions as constraints generally improves performance. Without this additional possibility we already achieve better performance than the results published by Hansen.

Chapter 6

Concluding Remarks

6.1 Summary and Contributions

Interval Arithmetic and Interval Constraints. We have specified the essential components of interval arithmetic and interval constraints, which include interval functions, interval constraint systems and consistency algorithms. Interval constraints is based on interval arithmetic, but is a generalization of interval arithmetic. An interval function F of a real function f over a given domain X can be computed in interval constraints. $F(X)$ can be translated into an interval constraint system C, in which there is a variable y for the value of F. We have proved that applying a consistency algorithm to C gives the same interval result for y as computing $F(X)$ in interval arithmetic.

The interval value of $F(X)$ is an approximation of $\Box f(X)$, the smallest interval containing the range of function f over X. In general it is much larger than $\Box f(X)$.

Hypernarrowing. We have presented an algorithm called hypernarrowing algorithm, which is based on a consistency algorithm. Through applying the hypernarrowing algorithm to the constraint system C translated from $F(X)$ and the interval for y, we usually obtain a much smaller interval for y than applying the consistency algorithm to C. Combining the hypernarrowing with the simple version of Hansen's algorithm for unconstrained global optimization that uses interval arithmetic and the midpoint test, we have achieved a factor of 1.8 to 9.7 in speedup over the simple version. We have also compared the semantics of the hypernarrowing algorithm with that of a constraint solver.

Unconstrained Global Optimization. After reviewing interval arithmetic methods and describing a generic Branch-and-Bound algorithm for solving unconstrained global optimization problems, we have proved the properties of the algorithm. We have investigated the role of interval constraints in global optimization, explained why interval constraints is more powerful than interval arithmetic and demonstrated how to obtain an interval arithmetic version and an interval constraint version of the Branch-and-Bound algorithm. The implementations of a variety of the interval arithmetic and interval constraint versions of the Branch-and-Bound algorithm have been given in BNR-Prolog. Computational results have showed that the interval constraint version of the order zero algorithm achieves a factor of 2.7 to 27.5 in speedup on the unconstrained global optimization test problems over the corresponding interval arithmetic version. The interval constraint version of the order one algorithm achieves a factor of 5.8 to 123.5 in speedup over the corresponding interval arithmetic version. In addition to the higher speed, the interval constraint versions are more declarative and use less memory space. We have also proposed two improvements on

the memory use of the Branch-and-Bound algorithm. The interesting thing is that these improvements also speed up the execution of the algorithm on about half of the unconstrained global optimization test problems.

Constrained Global Optimization. We have discussed the differences between solving the unconstrained global optimization problem and the constrained one. After reviewing the interval arithmetic methods for constrained global optimization, we have presented the transition from the Branch-and-Bound algorithm for unconstrained global optimization to the one for constrained global optimization. We have demonstrated how to obtain an interval constraint version of the Branch-and-Bound through using the applicable conditions as constraints. The interval constraint version has been implemented in BNR-Prolog and run on a few of test problems. The test results indicate that the interval constraint method using the problem's defining conditions, the midpoint condition and the Fritz-John condition as interval constraints in their definitional form achieves a factor of 4 to 58 in the number of iterations on the constrained optimization test problems over the interval arithmetic method. We have also investigated the effect of using the redundant Fritz-John condition as constraints. Computational results show that it gives considerable improvement in performance in most of cases.

6.2 Suggestions For Future Work

Combining a Point Method with an Interval Constraint One. The merit of point methods is their high efficiency. A point method can provide a real-valued approximate minimum f_m and the minimizer. The minimum f_m can be used as an

upper bound of f^* in an interval constraint method. In the interval constraint method, an upper bound u of f^* is obtained by computing the function value at a sampled point or the upper bound of function f over box B. Since f_m is generally much lower than u, using f_m as the upper bound of f^* may greatly speed up the interval constraint method. In addition, we can use the minimizer to guide the splitting of a box into sub-boxes. This leads to a considerable improvement of interval arithmetic methods [14]. Therefore, it is worth studying the combination of a point method and an interval constraint method.

Among point methods, a local optimization method is much faster than a global one [92]. So the higher priority should give to the study of the combination of a local optimization method and an interval constraint method.

Interval Constraint Compiler A high-level interval constraint system is translated into a low-level one, which consists of a conjunction of primitive constraints and an initial state. The consistency algorithm transforms the initial state into a consistent state or a failure state through "interpreting" the primitive constraints (i.e., applying the corresponding consistency operators). The consistency algorithm is an interpretor and the primitive constraints are the intermediate instructions to be interpreted. It is possible to design and implement an interval constraint compiler that translates a low-level (or high-level) interval constraint system into a sequence of C/C++ or assembly instructions [15]. This is analogous to designing a Prolog compiler [1, 91, 90, 54, 19] based on the techniques used in a Prolog interpretor [86, 13, 79]. In the place of a Prolog interpretor is the consistency algorithm. In general, the object code generated by a compiler from a source program achieves a factor of one to two order magnitudes in speed over the interpreted program [27].

Parallel Interval Constraint Methods. Interval constraint methods for solving the global optimization problem are based on an exhaustive search in a given box. The box is split into sub-boxes and the search continues in a selected sub-box. Since there are a number of sub-boxes during the search, we can use N processors and execute the interval constraint algorithm on each processor for searching a selected sub-box in parallel.

Much research work has been done about parallel interval arithmetic algorithms for global optimizations [48, 9, 14, 25, 26, 50, 63]. It is a good starting point for studying parallel interval constraint algorithms.

Improving the Consistency Algorithm. There exists potential parallelism in computing each consistency operator and the consistency algorithm. Exploiting this parallelism, we can improve the efficiency of solving interval constraints systems. There are two levels of parallelism that can be exploited to speed up the consistency algorithm. The computation of the consistency operator for a primitive constraint can be parallelized. For example, from the formula for computing C_{sum}, the consistency operator for the primitive constraint *sum*,

$$\begin{aligned} C_{sum}([a,b]\times[c,d]\times[e,f]) &= [a,b]\cap([e,f]-[c,d])\times \\ &\quad [c,d]\cap([e,f]-[a,b])\times \\ &\quad [e,f]\cap([a,b]+[c,d]) \end{aligned}$$

we can see that three interval operations and intersections can be parallelized. In the consistency algorithm, we only choose one primitive constraint at each iteration and apply the consistency operator of the constraint. It is possible to select several primitive constraints at an iteration and apply the consistency operators corresponding to

these constraints in parallel.

Bibliography

[1] Hassan A¨-Kaci. *Warren's Abstract Machine.* MIT Press, 1991.

[2] Götz Alefeld and Jürgen Herzberger. *Introduction to Interval Computations.* Academic Press, 1983.

[3] A. B. Babichev, O. B. Kadyrova, T. P.Kashevarova, A. S. Leshchenko, and A. L. Semenov. UniCalc, a novel approach to solving systems of algebraic equations. *Interval Computations*, 1993(2):29–47, 1993.

[4] P.G. Bao and J.G. Rokne. Existence of a unique zero of nonlinear systems. 1987.

[5] F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *Logic Programming: Proc. 1994 International Symposium*, pages 124–138, 1994.

[6] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and Boolean constraints. *Journal of Logic Programming.* To appear.

[7] Frédéric Benhamou and William J. Older. Programming with CLP(BNR): Examples on finite domains. To be published Journal of Logic Programming, 1993.

[8] A. Bernat and V. Kreinovich. Data processing beyond traditional statistics: applications of interval computations. a brief introduction. In V. Kreinovich, editor, *Extended Abstracts of International Workshop on Applications of Interval Computations*, pages 13–21, El Paso, TX, 1995.

[9] S. Berner. A parallel method for verified global optimization. In G. Alefeld, A. Frommer, and B. Lang, editors, *Scientific Computing and Validated Numerics*, Mathematical Research, volume 90, pages 200–206, Berlin, 1996. Akademie Verlag.

[10] BNR. BNR Prolog user guide and reference manual. 1988.

[11] Nicolas Bourbaki. *Elements of Mathematics: Theory of Sets.* Addison-Wesley Publishing Company, 1968.

[12] J. C. Burkill. Functions of Intervals. In *Proceedings of the Lodon Mathematical Society*, pages 22:375–446, 1924.

[13] J.A. Campbell. *Implementations of Prolog.* Ellis Horwood, 1984.

[14] O. Caprani, B. Godthaab, and K. Madsen. Use of a real-valued local minimum in parallel interval global optimization. *Interval Computations*, 1993(2):71–82, 1993.

[15] H. M. Chen and M. H. van Emden. Compiling interval constraint systems. In preparation.

[16] H. M. Chen and M. H. van Emden. Adding interval constraints to the Moore-Skelboe global optimization algorithm. In V. Kreinovich, editor, *International Journal of Reliable Computing (Formerly Interval Computations) Supplement*, pages 54–57, 1995.

[17] H. M. Chen and M. H. van Emden. An interval constraint method for constrained global optimization. In *Workshop on Constraint Programming Applications: An Inventory and Taxonomy*, Cambridge, Massachusetts, USA, August, 1996.

[18] J. G. Cleary. Logical arithmetic. *Future Computing Systems*, 2:125–149, 1987.

[19] Philippe Codognet and Daniel Diaz. wamcc: compiling Prolog to C. In Leon Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming*, pages 317–331. MIT Press, 1995.

[20] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[21] D. Stevenson, chairman, Floating-Point Working Group, Microprocessor Standards Subcommittee. IEEE standard for binary floating point arithmetic (IEEE/ANSI 754-1985). Technical report, IEEE, 1985.

[22] E. Davis. Constraint propagation with labels. *Artificial Intelligence*, 32:281–331, 1987.

[23] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint programming language CHIP. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, 1988.

[24] L. C. W. Dixon and G. P. Szego. *Towards Global Optimization 2.* North-Holland, Amsterdam, 1978.

[25] J. Eriksson. *Parallel Global Optimization using Interval Analysis.* PhD thesis, University of Umeå, Institute of Information Processing, 1991.

[26] J Eriksson and P. Lindstrœm. A Parallel Interval Method Implementation for Global Optimization Using Dynamic Load Balancing. *Reliable Computing*, 1(1):77–92, 1995.

[27] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a Compiler with C.* The Bejamin/Cummings Publishing Company, Inc., 1991.

[28] R. Fletcher. *Practical Methods of Optimization.* John Wiley and Sons Ltd., 1990.

[29] C. A. Floudas and P. M. Pardalos. *A Collection of Test Problems for Constrained Global Optimization Algorithms.* Springer-Verlag, 1990. Lecture Notes in Computer Science 455.

[30] David Goldberg. What every computer scientist should know about floating-point computation. *Computing Surveys*, 1991.

[31] Paul R. Halmos. *Naive Set Theory.* D. Van Nostrand, 1960.

[32] E. R. Hansen and G. W. Walster. Nonlinear equation and optimization. 1987.

[33] Eldon Hansen. On solving equations using interval arithmetic. *Math. Comput.*, 22:374–384, 1968.

[34] Eldon Hansen. Global optimization using interval analysis – the one-dimensional case. *Jour. Optimiz. Theory Applic.*, 29:331–344, 1979.

[35] Eldon Hansen. Global optimization using interval analysis – the multi-dimensional case. *Numer. Math.*, 34:247–270, 1980.

[36] Eldon Hansen. *Global Optimization Using Interval Analysis.* Marcel Dekker, 1992.

[37] P. Van Hentenryck and L. Michel. Helios: a modeling language for global optimization. In *Proceedings of the Second International Conference on the Practical Application of Constraint Technology*, pages 317–335. The Practical Applications Company, 1996.

[38] Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica: A Modeling Language for Global Optimization.* MIT Press, 1997.

[39] T. Hickey. CLP(F) and constrained ODEs,. In Yap Editors Jourdan, Lim, editor, *Proceedings of the Workshop on Constraint Languages and their use in Problem Modelling*, pages 69–79, Nov. 1994.

[40] T. Hickey, M.H. van Emden, and H. Wu. A unified framework for interval constraints and interval arithmetic. Technical report, University of Victoria, Canada, 1998.

[41] Einar Hille. *Analysis.* Blaisdell Publishing Company, 1964.

[42] Willi Hock and Klaus Schittkowski. *Test Examples for Nonlinear Programming Codes.* Springer-Verlag, 1981. Lecture Notes in Economics and Mathematicl Systems 187.

[43] K. Ichida and Y. Fujii. An interval arithmetic method for global optimization. *Computing*, 23(1):85–97, 1979.

[44] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *ACM Conference on Principles of Programming Languages*, Munich, 1987.

[45] A. H. G. Rinnooy Kan and G. T. Timmer. New methods in optimization and their industrial uses. In *Argument for the Unsolvability of Global Optimization Problems*, pages 133–155. Birkhauser Verlag, 1989. Basel.

[46] R. B. Kearfott. An interval branch and bound algorithm for bound constrained optimization problems. *Journal of Global Optimization*, 2:259–280, 1992.

[47] R. Baker Kearfott. *Rigorous Global Search: Continuous Problems.* Kluwer Academic Publishers, 1996. Nonconvex Optimization and Its Applications.

[48] Vladik Kreinovich and Andrew Bernat. Parallel algorithms for interval computations: An introduction. *Interval Computations*, (3):6–62, 1994.

[49] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In *The Second Berkeley Symposium on Mathematical Statistics and Probability.* University of California Press, 1951. Ed. J. Neyman.

[50] A. Leclerc. Parallel interval global optimization in C++. *Interval Computations*, 1993(3):148–163, 1993.

[51] J. H. M. Lee. *Numerical Computation As Deduction In Constraint Logic Programming.* PhD thesis, University of Victoria, Victoria, BC, Canada, 1992.

[52] J. H. M. Lee and M. H. van Emden. Adapting CLP(R) to floating-point arithmetic. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 996–1003, 1992.

[53] Wm Leler. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley Publishing Compnay, Inc., 1988.

[54] M. R. Levy and R. N. Horspool. Translator-based multi-paradigm programming. *Journal of Software and Systems*, 1993.

[55] H. R. Lewis and L. Denenberg. *Data Structures and Their Algorithms.* Harper Collins Publishers, 1991.

[56] J. D. C. Little, K. G.M urty, and D. W. Karel. An algorithm for the travelling salesman problem. *Operations Research*, 11, 1963.

[57] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Kaleidoscope: A constraint imperative programming language. In Brian Mayoh, editor, *Constraint Programming, NATO ASI Series*, pages 313–329. Springer-Verlag, 1994.

[58] Brian Mayoh, Ann Tyugu, and Tarmo Uustalu. Constraint satisfaction and constrain programming: A brief lead-in. In *Constraint Programming, NATO ASI Series, Series F: Computer and System Science Vol. 131.* Springer-Verlag, 1994. Ed. Brian Mayoh.

[59] R. E. Moore. *Interval Arithmetic and Automatic Error Analysis in Digital Computing.* PhD thesis, Stanford University, October 1962.

[60] R. E. Moore. A test for existence of solutions to nonlinear systems. *SIAM J. Numer. Anal.*, 14(4):611–615, September 1977.

[61] R. E. Moore. Method and applications of interval analysis. *SIAM Studies in Applied Mathematics. SIAM, Philadelphia*, 1979.

[62] R. E. Moore. Global optimization to prescribed accuracy. *Computers Math. Applic.*, 21:25–39, 1991.

[63] R. E. Moore, E. Hansen, and A. Leclerc. Rigorous methods for parallel global optimization. In A. Floudas and P. Pardalos, editors, *Recent Advances in Global Optimization*, pages 321–342, Princeton, N.J., 1992. Princeton Univ. Press.

[64] Ramon E. Moore. *Interval Analysis.* Prentice-Hall, 1966.

[65] W. J. Older. CLP(BNR) algorithm for travelling salesman. Technical report, Bell-Northern Research Computing Research Laboratory, August, 1993.

[66] W. J. Older and A. Vellino. Constraint arithmetic on real intervals. In *Constraint Logic Programming: Selected Research.* MIT Press, 1993. Eds. F. Benhamou and A. Colmerauer.

[67] William Older. Constraints in BNR Prolog. Bell-Northern Research Technical Report, 1993.

[68] William Older and André Vellino. Constraint arithmetic on real intervals. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming*, pages 175–195. MIT Press, 1993.

[69] William J. Older. Application of relational interval arithmetic to ordinary differential equations. In Jean Jourdain, Pierre Lim, and Roland H.C. Yap, editors, *Workshop on Constraint Languages and their Use in Problem Modelling*, pages 60–69, Ithaca, New York, November 1994.

[70] W.J. Older. The application of relational arithmetic to X-ray diffraction crystallography. Technical report, Bell-Northern Research Computing Research Laboratory, 1989.

[71] W.J. Older. Interval arithmetic specification. Technical report, Bell-Northern Research Computing Research Laboratory, 1989.

[72] W.J. Older. Using interval arithmetic for non-linear constrained optimization. Technical report, Bell-Northern Research Computing Research Laboratory, 1993.

[73] W.J. Older. Note on computing square-well spectra using interval arithmetic. Technical report, Bell-Northern Research Computing Research Laboratory, 1994.

[74] H. Ratschek and J. Rokne. *New Computer Methods for Global Optimization.* Ellis Horwood/John Wiley, 1988.

[75] Klaus Schittkowski. *More Test Examples for Nonlinear Programming Codes.* Springer-Verlag, 1987. Lecture Notes in Economics and Mathematicl Systems 282.

[76] Alexander L. Semenov. Solving optimization problems with help of the UniCalc solver. In *Applications of Interval Computations.* Kluwer Academic Publishers, 1996. Eds. R. Baker Kearfott and Vladik Kreinovch.

[77] M. Simonnard. *Linear Programming.* Prentice-Hall, Inc., Englewood Cliffs, N.J, 1966. translated by W.S.Jewell.

[78] S. Skelboe. Computation of rational interval functions. *Nordisk Tidskrift for Informationsbehandling (BIT)*, 14:87–95, 1974.

[79] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, Second edition, 1994.

[80] I. Sutherland. *Sketchpad: a Man-Machine Graphical Communication System.* PhD thesis, Dept. of Electrical Engineering, MIT, 1963.

[81] A. Torn and A. Zilinskas. *Global Optimization.* Springer-Verlag, Berlin Heidelberg New York, 1989. Lecture Notes in Computer Science.

[82] M. H. van Emden. The compatibility operator for real intervals, herbrand universes and finite domains. Technical report, Department of Computer Science, University of Victoria, 1994.

[83] M. H. van Emden. Canonical extensions as common basis for interval constraints and interval arithmetic. In *Proceedings of the Sixth French Conference on Logic and Constraint Programming,* Orléans, France, 1997.

[84] M. H. van Emden and H. M. Chen. Interval constraints for unconstrained global optimization. *International Journal of Reliable Computing (formly Interval Computations)*, 1995. Submitted.

[85] M.H. van Emden. The compatibility operator in numerical computation. In preparation.

[86] M.H. van Emden. An algorithm for interpreting Prolog programs. In *Implementations of Prolog*, pages 93–110. Ellis Horwood, 1984.

[87] M.H. van Emden. Algorithmic power from declarative use of redundant constraints. Technical report, University of Victoria, Canada, 1998.

[88] G. R. Walsh. *Methods of Optimization.* John Wiley and Sons Ltd., 1975.

[89] D. Waltz. Understanding line drawings in scenes with shadows. In Patrick Henry Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[90] David H. D. Warren, Luis M. Pereira, and Fernando Pereira. Prolog – the language and its implementation compared to lisp. In *Proc. Symp. on AI and Programming Languages*, 1977. SIGPLAN Notices vol. 12.

[91] D.H.D. Warren. Implementing prolog — compiling predicate logic programs. D.A.I. Research Report 39–40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[92] D. Xue and Z. Dong. Improvement and testing of local and global optimization programs for NGFT project. Technical report, Department of Mechanical Engineering, University of Victoria, 1995.