FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Extending the Gecode Framework with Interval Constraint Programming

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Ivan Ivezić
Matrikelnummer 1128638

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Univ.Lektor Dr. Luca Di Gaspero

Wien, 15. Oktober 2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Extending the Gecode Framework with Interval Constraint Programming

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Ivan Ivezić
Registration Number 1128638

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Univ.Lektor Dr. Luca Di Gaspero

Vienna, 15. October 2012 _____ _____
(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Ivan Ivezić
Rapska 3, 42000 Varaždin, Kroatien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

This thesis introduces the reader to the basics of constraint programming, including the main concepts such as search space, variables and their domains, and constraints. Furthermore, the constraint satisfaction problem modeling process, and the general procedure required to solve it is introduced. Constraint satisfaction problem solving concepts such as propagation and branching are explained for a general constraint satisfaction problem as well.

Interval constraint programming, a subclass of constraint programming where the domains of variables in the problems are intervals, is then introduced. Then, basic concepts of interval arithmetic needed for interval constraint programming are shown. Afterwards, the peculiarities of interval constraint satisfaction problems, as opposed to general constraint satisfaction problems are highlighted. Furthermore, generic consistency notions, namely, node and arc consistency are introduced. They are followed with the description of hull consistency and box consistency, which are the two consistency notions relevant to interval constraint programming. A method for enforcing both hull and box consistency is given in detail.

The C++ constraint programming framework Gecode is then briefly presented. An extension of Gecode supporting interval constraint programming, that was developed alongside this thesis, is described in detail. The implementation relies on the Boost Interval library to handle intervals. To implement the box consistency propagator, an additional library, namely, SymbolicC++ was used, and had to be extended as well. The necessary extensions of SymbolicC++ library are described as well.

The implemented extension was tested on various scalable benchmarks, namely Broyden Banded, Broyden Tridiagonal and Brown, each having its unique properties testing, and highlighting a particular feature of the system. Experiments on Broyden Banded show that SymbolicC++ may have been a suboptimal choice for the extension, as it suffers from relatively high constraint initialization time. Broyden Tridiagonal evaluates the performance of box consistency propagation, whereas Brown evaluates hull consistency propagators.

The final test of the extension is the 3D reconstruction problem. The formal description of the problem is given, and the results of the 3D reconstruction obtained with the extension are shown, both statistically and graphically.

Kurzfassung

Die vorliegende Arbeit führt den Leser zunüchst in die Grundlagen von Constraint Programming sowie hierfür relevante Konzepte wie Suchräume, Variablen, ihre Domänen und Constraints ein. Weiters wird beschrieben wie reale Probleme als Constraint Satisfaction Modelle dargestellt und gelöst werden können. Grundprinzipien wie Constraint Propagation und die Baumsuche werden skizziert.

Interval Constraint Programming ist eine Unterklasse von Constraint Programming, in der die Domänen der Variablen Intervalle sind. Um diese näher zu betrachten werden zunächst die Grundlagen der Intervall-Arithmetik vorgestellt. Danach wird auf Besonderheiten der Interval Constraint Satisfaction Probleme eingegangen. Neben Konsistenzbegriffen wie Knoten- und Bogenkonsistenz haben nun Hüllen- und Boxkonsistenzen eine große Bedeutung. Algorithmen um die beiden letztgenannten Konsistenzen zu erreichen werden im Detail beschrieben. Gecode ist eine C++ Constraint Programming Entwicklungsumgebung, für die in dieser Diplomarbeit entsprechende Erweiterungen für Inverval Constraint Programming entwickelt wurden. Für die Intervallarithmetik wurde hierfür auf die Boost-Library sowie SymbolicC++ zurückgegriffen.

Die implementierte Erweiterung wurde auf verschiedenen skalierbaren Benchmark-Instanzen getestet, nämlich Broyden Banded, Broyden Tridiagonal und Brown. Jede dieser Benchmark-Instanzen hat spezielle Eigenschaften. Die Experimente mit Broyden Banded zeigen, dass SymbolicC++ eine Schwachstelle der Erweiterung sein könnte, weil es zu langen Constraint - Initialisierungszeiten führt. Broyden Tridiagonal wertet im Speziellen die Leistung der Boxkonsistenz, während Brown primär die Leistung in Bezug auf die Hüllenkonsistenz aufzeigt. Weiters wurde die Erweiterung auf einem komplexeren 3D-Rekonstruktionsproblem erfolgreich getestet.

Contents

CHAPTER 1

Introduction

Constraint programming is a programming paradigm for solving problems defined through a set of constraints between variables [2]. The idea is to specify a set of constraints that a solution must satisfy, and then the constraint solver can reason about the possible properties of a solution, while constantly minding the constraints a solution has to satisfy.

It is a form of declarative programming, as the user specifies a set of variables along with a set of constraints between them, without specifying the sequence of steps that have to be carried out to find the solution.

A problem specified in such a way, usually called constraint satisfaction problem, can then be solved through various means. If the constraints have a particular form, namely, if they are linear inequalities, the Simplex algorithm [16] may be employed, which has "almost" polynomial run time, that is, its run time is polynomial in most practical cases, even though its worst case runtime is exponential. For constraints having a more general form, such as the ones that the system described in this work is able to solve, the Simplex algorithm is, is general, not applicable. In such cases, various methods may be applied, often specific to the form of the constraint. However, most approaches for solving constraint satisfaction problems, are solved with some form of tree search and propagation.

Software frameworks are available for solving constraint satisfaction problems. One of such constraint programming frameworks is Gecode, which is designed to be easily extensible [19]. It allows the user to solve problems with integer, boolean, and set variables, but is currently unable to reason about interval variables, which are variables whose domains represent real values (usually represented as floating-point values). This paper describes an extension of Gecode, in which interval variables are added, along with the other necessary additions to make problem-solving with interval variables possible.

Furthermore, this functionality is demonstrated on a 3D reconstruction problem. The task in this problem is to find a set of vertices, given a bounding box for each vertex, as well as geometrical constraints that their vertices and faces have to satisfy (e.g. orthogonality or parallelism). Since the constraints in the problem consist of non-linear constraints, such as trigonometric constraints, 3D reconstruction problem cannot, in general, be solved through methods such as

Simplex algorithm. Thus, a method such as interval constraint programming (ICP) is required to solve this problem.

1.1 Constraint Programming Basics

A general constraint satisfaction problem consists of a definition of the variables and the constraints between them. Each variable is defined through its initial domain, whereas constraints may be defined in any form which is supported in the solver used.

Variables

Every useful constraint satisfaction problem has *variables* $V = \{v_1, v_2, \dots v_n\}$, where $n \geq 0$. Each variable v_i has a domain, that is, a set of admissable values, D_i associated to it. A constraint satisfaction problem includes the definition of an initial domain D_i for each variable v_i, but the domains may subsequently change (namely, shrink) during the solving process.

A *domain* of a variable v_i is denoted by D_i. Each domain is a set of values that are permissable for the respective variable. As expected, variables used in constraint programming can have numeric domains. One can limit the numeric domain to have only, for example, values from the set of complex numbers, or reals, or integers, within some range. However, beside numerical values, variables can also represent anything else that can be given a domain - for example, letters, sets, and so on.

Furthermore, mathematically, a domain is a set of all admissable values for a variable. However, in practice, a domain will usually be represented only through its bounds. Thus, due to large domains, there will often be no practical way to assert that a value between the bounds is not in a domain. This is, though, not a serious limitation in most cases.

A sample of integer variables is $\{x \in \{1, 2\}, y \in \{1, 2, 3, 4, 624\}\}$. An example set for the interval variables is $\{x \in [1, 2.3], y \in [3, 4] \cup [7, 15]\}$.

Search Space

A space defined by the domains of variables as $D_1 \times D_2 \times \cdots \times D_n$ is called *search space*. When no ambiguity is present, term *space* may be substituted for *search space*. A *subspace* S_s of a space $S = D_1 \times D_2 \times \dots D_n$ is defined as $S_s = D'_1 \times D'_2 \times \dots D'_n$, where, for every i, $D'_i \subseteq D_i$. Furthermore, a space S_p is a *proper subspace* of the space S if and only if S_p is a subspace of S, and there exist j such that $D'_j \subset D_j$, thus forcing subspace S_s to omit at least one element from one domain of space S. In symbols, $S_s \subseteq S$ means that S_s is a subspace of S, whereas $S_p \subset S$ means that S_p is a proper subspace of S.

Furthermore, S_1 is a (proper) *superspace* of S_2 if and only if S_2 is a (proper) subspace of S_1. Union of two spaces contains all the points in either space, while their intersection contains all points included in both spaces.

When a space contains no points, it is *empty*, denoted by $\emptyset$. Otherwise, the space is *nonempty*.

Intuitively, a solution to a constraint satisfaction problem is a search space that is a subset of the initial space (given by the constraint satisfaction problem), and that satisfies all the con-

straints. The goal of the solving process is to eliminate infeasible values from the search space (according to the constraints), thus obtaining a solution, or to discover that no solution exists.

Constraints

A central notion of constraint programming are constraints, imposing a relation that must hold among the variables. Every solution has to satisfy all constraints in the model, for any combination of values for each of its variables from their domains. When a solution maps every variable to exactly one value, then a solution gives a simple valuation for a set of variables - and all constraints need to hold for that valuation.

Formally, let c denote a particular constraint in a particular constraint satisfaction problem with a search space S. Furthermore, let $D_i^c(S)$ denote the projection of the domain of the ith variable D_i with respect to the constraint c and the space S, where $D_i^c(S) \subseteq D_i$. This can be written as

$$D_i^c(S) = \{d \in D_i \mid c(d) \text{ is satisfied}\}. \tag{1.1}$$

Then, constraint space $c(S)$ with respect to the search space S is defined as

$$c(S) = D_1^c(S) \times D_2^c(S) \times \cdots \times D_n^c(S), \tag{1.2}$$

that is, a space having a dimension for every variable in the problem, and being a subspace of the space S. Note that the space defined by the constraint is a function of the search space, as different constraint spaces may be induced for different search spaces with the same set of constraints.

Note that some authors use this definition of the constraint space as the definition of the search space.

If one has a set of m constraints $c_1, \ldots, c_m$, and a problem with some initial search space S, the problem is *solvable* (conversely, S' is the *solution* of that problem), if and only if there exists empty space S' such that

$$S' \subseteq c_1(S) \cap c_2(S) \cap \cdots \cap c_m(S). \tag{1.3}$$

In other words, space S' is a *solution* if it satisfies all the constraints. On the other hand, if

$$c_1(S) \cap c_2(S) \cap \cdots \cap c_m(S) = \emptyset, \tag{1.4}$$

the problem has no solution (for the initial space S).

A step towards solving the constraint satisfaction problem can be expressed as

$$S_{i+1} = c_1(S_i) \cap c_2(S_i) \cap \cdots \cap c_m(S_i) \tag{1.5}$$

where S_0 is the initial search space. This process is iterated until such j is found, for which S_j is either a solution, or is an empty set (which means no solution exists). The process towards finding the solution in this way is called *constraint propagation*, or just *propagation*.

A solution S_j found in this manner is a maximal solution, that is, a solution such that no convex solution that is its superset exists. Note that this does not mean that all solutions are

subsets of the maximal solutions, as there may be multiple disjoint maximal solutions[1]. Note that this process is not guaranteed to converge - for example, it will not converge when multiple maximal solutions exist, as the process is not able to decide on which to converge.

However, this process guarantees that, for every j, all solutions, if they exist, are contained within S_j. Non-convergence can be detected by finding a natural number k, such that $S_k = S_{k+1}$. Then, S_{k+2}, and all subsequent spaces will not be different from S_k either.

Even though it is simple to describe a solution (even maximal solution) in terms of the above intersection, solving a constraint satisfaction problem is far from trivial. The biggest obstacle is possible non-convergence of the procedure. Nevertheless, under some assumptions, namely the assumption of the function implementing the constraint being *inflationary*[2] and *monotonic*[3], any propagation procedure is guaranteed to converge [1].

Even in the case when the above procedure converges, the difficulty still lies not only in the (possible) vastness of the search space, but also in the fact that one often cannot transform arbitrary-form constraints to a space, which can then be directly intersected with the initial space, as well as the fact one needs to iteratively repeat this process. Furthermore, finding a maximal solution is rarely feasible, but in practice finding any solution often suffices. If a particular quality of a solution is desired, this can be enforced through the addition of further constraints.

The need for the iterative repetition of the process comes from the dependence of the constraint space on the search space. Because of this, if $S_{i+1} \neq S_i$, it might also be the case that $C(S_{i+1}) \neq C(S_i)$.

Examples of constraints are simple linear inequalities between the variables, such as $x < 5$ or $x \leq y$, but can also be more complex relationships, such as $\sin x = \sqrt{y}$. As expected, the first two constraints could be applied to both integer and interval case, whereas the last one is natural only in the interval case (as the results of the functions are real numbers), so it has only limited applicability (if any) in the integer case.

Consider a initial search space in the interval case $\{x \in [1, 2], y \in [1, 3]\}$, and a constraint $x < y$. Finding the space this constraint defines (which is dependent on the search space), is non-trivial. The space defined by the constraint would be $\{x \in [1, 2], y \in (1, 3]\}$, i.e. as the initial search space, but with value of 1 removed as a possibility of y. Thus, for any chosen x, there would then exist an y satisfying the constraint, and vice versa. If 1 were not removed from y, there would be no $x \in [1, 2]$, such that $x < 1$. Furthermore, note that space defined by the constraint is a function of the search space. For example, suppose another constraint limited x to the interval $x \in [1.5, 2]$. This would limit options for y further, namely, to $y \in (1.5, 3]$. This demonstrates that solving constraint satisfaction problems is usually not possible with cheap computational tricks, even with very simple constraints, but has to be done through elaborate computation process.

[1] Consider a constraint such as $x \neq 0$, on $x \in \mathcal{R}$. Here, maximal solutions are both $x \in (-\infty, 0)$ and $x \in (0, \infty)$.

[2] Consider a partial order $(D, \sqsubseteq)$. A function f on D is called *inflationary* if $x \sqsubseteq f(x)$ for all x.

[3] Consider a partial order $(D, \sqsubseteq)$. A function f on D is called *monotonic* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ for all x, y.

Modeling

In order to solve an abstractly stated constraint satisfaction problem, one first has to model it - that is, translate a generic problem into constraint programming terms. That involves deciding what the variables will be, as well as choosing their initial domains. Furthermore, modeling involves laying out the constraints that need to hold between the variables.

An example of the distinction between the problem itself, and its model can be seen in the 3D reconstruction problem - whose model has been described in detail in Chapter 5. Here, the problem is the idea of finding a matching coordinates of a 3D object, given certain geometric constraints. The model, on the other hand, consists of the concrete variables holding the coordinates of the points (and some other variables, described in detail in Chapter 5), as well as the concrete mathematical equations and inequalities that take place of the more abstract ideas such as parallelism or equal angles.

In this paper, when not ambiguous[4], terms constraint satisfaction problem and its model are used interchangeably.

An Example Model of Sudoku

Sudoku is a combinatorial number placement puzzle. The objective of the puzzle is to fill the blank slots in the 9×9 grid. The 9×9 grid consists of 3×3 regions of 3×3 slots each. Every slot may contain a single number from 1 to 9, and every 3x3 region has to contain every of those 9 numbers exactly once. Furthermore, every row and every column of the whole grid has to contain every of those 9 numbers exactly once.

There exist other variations of Sudoku, with various grid sizes, but these dimensions and constraints are by far the most common.

When solving the puzzle, user is given a partially filled grid, and is expected to fill rest of those slots with the allowed numbers $1, 2, \ldots, 9$, while satisfying said constraints. Being in essence a constraint satisfaction problem, Sudoku is naturally suited to be solved by constraint programming.

To model Sudoku as a constraint satisfaction problem, one must transform the rules of the puzzle into variables, and formal constraints. Since a single number may be fitted in each slot, it makes sense to have one variable per slot, representing the number in the slot. The grid, and the associated variables with each slot are shown in Figure 1.1. The only constraint imposed by the rules of Sudoku is that all numbers in some set (either row, column, or region) be all different. Thus, one can introduce *alldifferent*($\mathbf{V}$) constraint, over a set of variables $\mathbf{V} = v_1, \ldots v_n$. This constraint holds if all the variables in the set $\mathbf{V}$ have different values. Formally, *alldifferent*($\mathbf{V}$) can be defined as

$$alldifferent(\mathbf{V}) \Leftrightarrow \forall i \in \{1, \ldots, n\} \ \forall j \in \{1, \ldots, n\} \ \ (i \neq j \Rightarrow v_i \neq v_j) \tag{1.6}$$

where $v_i \neq v_j$ denotes the fact that v_i and v_j contain different values.

[4]This is unambiguous when considering problems that can be translated directly to models, such as, for example, $x \in [1, 3], y \in [1, 3], x < y$

Figure 1.1: Sudoku grid with associated variables

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}
x_{19}	x_{20}	x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}
x_{28}	x_{29}	x_{30}	x_{31}	x_{32}	x_{33}	x_{34}	x_{35}	x_{36}
x_{37}	x_{38}	x_{39}	x_{40}	x_{41}	x_{42}	x_{43}	x_{44}	x_{45}
x_{46}	x_{47}	x_{48}	x_{49}	x_{50}	x_{51}	x_{52}	x_{53}	x_{54}
x_{55}	x_{56}	x_{57}	x_{58}	x_{59}	x_{60}	x_{61}	x_{62}	x_{63}
x_{64}	x_{65}	x_{66}	x_{67}	x_{68}	x_{69}	x_{70}	x_{71}	x_{72}
x_{73}	x_{74}	x_{75}	x_{76}	x_{77}	x_{78}	x_{79}	x_{80}	x_{81}

Then, variables for Sudoku may be expressed as

$$x_i \in \{1, 2, \ldots, 9\}. \tag{1.7}$$

Equation 1.8 expresses the idea, for every row, that it has to hold all different values. The same idea is expressed for every column through the Equation 1.9, and for every region through the Equation 1.10.

$$\begin{aligned} &\mathit{alldifferent}(x_{1+k}, x_{2+k}, x_{3+k}, x_{4+k}, x_{5+k}, x_{6+k}, x_{7+k}, x_{8+k}, x_{9+k}), \\ &\qquad \forall k \in \{0, 9, 18, 27, 36, 45, 54, 63, 72\} \end{aligned} \tag{1.8}$$

$$\begin{aligned} &\mathit{alldifferent}(x_{1+k}, x_{10+k}, x_{19+k}, x_{28+k}, x_{37+k}, x_{46+k}, x_{55+k}, x_{64+k}, x_{73+k}), \\ &\qquad \forall k \in \{0, 1, 2, \ldots, 8\} \end{aligned} \tag{1.9}$$

$$\begin{aligned} &\mathit{alldifferent}(x_k, x_{k+1}, x_{k+2}, x_{k+9}, x_{k+10}, x_{k+11}, x_{k+18}, x_{k+19}, x_{k+20}), \\ &\qquad \forall k \in \{1, 4, 7, 28, 31, 34, 55, 58, 61\} \end{aligned} \tag{1.10}$$

Furthermore, as already stated, one's objective usually is to complete partially filled Sudoku puzzle. In order for this constraint program to be able to deliver a solution to a partially filled Sudoku, further constraints in the form

$$x_i = v_i \tag{1.11}$$

have to be introduced, for each of the values already present in the puzzle. Here, v_i represents the value in the already filled, ith slot. Constraints given by Equation 1.11 merely restrict the variables associated to pre-filled slots to that values.

1.2 Constraint Programming Solving Basics

Propagation

Usually, a constraint programming solver works by beginning with an initial search space, and then iteratively attempts to reduce the search space, by eliminating values that may never be

part of the solution. Such values are said to be inconsistent in the constraint programming terminology. If some variable were to take an inconsistent value, then there exist values in domains of other variables for which at least one constraint would be violated. Removal of inconsistent values is called *constraint propagation*, or just *propagation*, which is equivalent to the definition from the definition from the Section 1.1.

In order for constraint propagation to be possible, constraints have to be built with operators (functions) which have to fulfill certain requirements. However, having enough knowledge about the constraint to be able to remove inconsistent values through reasoning about its properties is not necessary to use constraint programming. It is, though, necessary to be able to determine whether a constraint is satisfied in a space or not, at least for spaces that consist of only one point (spaces that have exactly one value in the domain of each variable). Intuitively, if such a test did not exist, constraint's validity could never be checked, and solutions could not be distinguished from non-solutions.

Nevertheless, one often attempts to use constraints for which more advanced reasoning is possible, namely, the removal of inconsistent values rather than simply deciding satisfiability. The reason for this is that constraint propagation is crucial for performance of the solver.

As already stated, when solving a constraint satisfaction problem, one is often satisfied with one, or at most, finitely many point solutions. In some applications, though, one may distinguish between solutions of higher and lower quality. This is the case in problems such as, for example, travelling salesman problem. There, a solution must represent a route that visits each city exactly once, but shorter routes are better (assuming minimization) than longer ones. However, the following presentation will assume that no minimization or maximization is desired, that is, that any solution is (equally) satisfactory.

Generic procedure for constraint propagation is a relaxation of the method described in Section 1.1. Again, let S_0 be the initial search space. Furthermore, let $c_j(S)$ denote the space defined by the jth constraint in the model of the problem for the space S, and let $T_j(S)$ be an arbitrary superspace of $c_j(S)$, i.e. $T_j(S) \supseteq c_j(S)$. Then, the next space S_{i+1} can be chosen, starting from the space S_i by satisfying the conditions

$$S_{i+1} \subseteq S_i \tag{1.12}$$

and

$$S_{i+1} \supseteq T_1(S_i) \cap T_2(S_i) \cap \cdots \cap T_m(S_i). \tag{1.13}$$

This form uses, for jth constraint, superspace T_j of the space induced by the constraint c_j, instead of c_j directly. Such relaxation is necessary as it is often impossible, or infeasible, to calculate a particular $c_j(S)$ correctly. As

$$c_1(S_i) \cap c_2(S_i) \cap \cdots \cap c_m(S_i) \subseteq T_1(S_i) \cap T_2(S_i) \cap \cdots \cap T_m(S_i) \tag{1.14}$$

no solution included in the method described in Section 1.1 will be left out by this method, so it is indeed a relaxation. Furthermore, it might be impossible or infeasible to represent exactly the

space $T_1(S_i) \cap T_2(S_i) \cap \cdots \cap T_m(S_i)$[5], so any of its superspaces is admissable here, as long as it is a subspace of the space S_i. The latter condition is necessary, as it ensures that every step has at most as many inconsistent values as the previous. In practice, solvers will take the smallest representable space for S_{i+1}.

Since this procedure is a relaxation of the stricter one, convergence is not guaranteed either - even, there might exist cases in which the original method converges, where the relaxed variant does not. Thus, solvers must be able to detect non-convergence. This is done analogously to the detection method from Section 1.1, namely, if there exists some k so that $S_k = S_{k+1}$, any further S_i will be equal to S_k[6]. In this case, to ensure convergence, this procedure has to be complemented by another, for example branching as described in Section 1.2. Complementary procedure will then yield $S'_k \subset S_k$ (in any particular branch, no ambiguity is introduced as branches are disjoint), from which the procedure can continue on.

This process is repeated, until either a solution is found, or it is proven none exists.

It should be noted that solvers are free to choose each of $T_j(S_i)$, as long as it is a superspace (not necessarily proper) of $C_j(S_i)$. This means that solvers are permitted to use different methods of calculating them in different iterations. Also, they are allowed not to propagate on some, jth constraint[7].

Thus, propagation using even only one constraint is possible. Often, it is in fact desirable not to do all the possible propagation at once for performance reasons, as different constraints might have different computational complexity associated to their propagation. Thus, it is often beneficial to attempt to reduce the space with the cheapest operators as much as possible first, and only then proceed to the more computationally expensive ones. In fact, many constraint satisfaction problem solvers propagate only on one constraint at a time, an example of which is in fact Gecode [19].

For example, consider a problem with two variables x and y, with initial domains $x \in [0, 100], y \in [1, 20]$, and two constraints, $x \leq 10$ and $x \geq y$. From the first constraint $x \leq 10$, one can infer that the domain of x can be at most $[0, 10]$, thus yielding a new search space $x \in [0, 10], y \in [1, 20]$. From the second constraint, one can infer that x must be greater than or equal to 1, since it is greater than or equal to y, and 1 is the minimum value y might take. Furthermore, one can infer that y must be less than or equal to 10, as 10 is maximal value for x. Thus, the solution is then $x \in [1, 10], y \in [1, 10]$. This means that, upon choosing any value for

[5]In big, or even infinite domains, it is infeasible to track, for every value, whether it is in a domain. In practice, one would often represent the domain through the lower and upper bound. For example, if the intersection $T_1(S_i) \cap T_2(S_i) \cap \cdots \cap T_m(S_i)$ gives $[1, 2] \cup [3, 4]$, solver might decide to represent that as a hull of the two intervals, namely, $[1, 4]$, and use that value for S_{i+1}, to avoid representing the values through possibly very big stack of intervals representing unions. This is, in fact, the approach taken in the extension described in this work. Nevertheless, more exact representations can still be used, and the solver is welcome to use the smallest space it can to represent the result of the intersection.

[6]This is the case with the assumption that the solver will, from a particular right-hand side $T_1(S_i) \cap T_2(S_i) \cap \cdots \cap T_m(S_i)$, and S_i, always calculate the same S_{i+1}. Given this assumption, and that $S_i = S_{i+1}$, one can infer $S_{i+1} = S_{i+2}$, and so on.

[7]To fit this within an algorithm, one would simply put $T_j(S_i) = S_0 \cup C_j(S_i)$, where S_0 is the initial search space. Then, $T_j(S_i) \supseteq C_j(S_i)$, and, $T_j(S_i) \supseteq S_0$. As, for any i, $S_i \subseteq S_0$, $T_j(S_i)$ has no effect on the intersection defining S_{i+1} - it need not even be considered in calculation.

x from its domain, there exists at least one value for y that would satisfy all the constraints, and vice versa.

However, suppose a constraint $x \neq 5$ were added to the system. Then, the strict domain for x would clearly be $x \in [1, 5) \cup (5, 10]$. But, if the solver used were unable to represent the union of intervals, it would be unable to find a solution with propagation alone. Inability of a solver to represent the intervals as unions is, though, not a severe restriction in practice, and it might be crucial to performance.

Branching

Often, the program will not be able to compute a solution just with constraint propagation (it will not converge to a solution). Moreover, it might be the case that some inconsistent values are present in the current space, yet, it is impossible to remove them through propagation due to infeasibility to represent the domain without those values.

Because of this, an constraint solver will often branch - that is, attempt to find solution by cases. This means it will split the search space which it currently examines in multiple subspaces, such that their union yields the initial search space. It is necessary that the union be the original space, so no solution can be skipped. Normally, one would also attempt to split the space in such a way to minimize the intersection of the subspaces, to minimize the amount of duplicate work. The solver will then examine each subspace - that is, try to carry out constraint propagation and check whether it has obtained a solution. If not, it will recursively repeat this procedure until it has found a solution, determined there is none, or met some other stopping criterion. Such a procedure is inspired [18] by the Davis-Putnam procedure for SAT [7].

This process of splitting spaces into subspaces, called branching, is an important part of most constraint solvers. Usually, the solver decides on how to split into subspaces, and examines the first subspace attempting to find a solution there. If none exists, or the user has requested more solutions than were found in the first branch, the solver would explore the second branch, and so on. Thus, branchings form a *search tree*.

Formally, branching can be carried out by introducing complementary constraints to the model, such as $x \leq a$ on one branch, and $x > a$ on the other branch, where a is some value in the domain of variable x, usually its midpoint. Then, constraint propagation is carried out in every branch, with the old constraints and the newly introduced ones.

It is usually most beneficial to branch in two branches, as branchings increase the overall number of spaces that have to be considered. It is, thus, usually better to carry out as much propagation as possible, and only when propagation becomes stuck for a particular branch, branch in the minimal number of branches, namely two. Moreover, it can be proven that binary and k-ary branching are equivalent.

For example, consider a problem $x \in [0, 10], x \neq 3$, and a solver which is unable to propagate this constraint to be $x \in [0, 3) \cup (3, 10]$. Since the initial space is not a solution, and no propagation is possible, the solver could branch the space in the midpoint of the domain of x, namely, to create two spaces $S_l = [0, 5]$ and $S_r = [5, 10]$[8]. If the solver carried out a breadth-first

[8]Careful reader will note that the intervals S_l and S_r are not disjoint. Normally, one would strive to make the domains disjoint, to minimize the amount of duplicate work that needs to be done in both branches. However,

search, it would immediately notice that S_l is not a solution, and that it needs further branching, whereas S_r is a solution. On the other hand, if it carried out (left-first) depth-first search, it would not get to considering S_r, before exhausting S_l.

When considering S_l, a solver would branch it to $S_{ll} = [0, 2.5]$ and $S_{lr} = [2.5, 5]$. It would then notice that S_{ll} is a solution, whereas S_{lr} still needs exploring. The branch covering number 3 may, in theory, be expanded *ad infinitum*, however, solvers might dismiss a sufficiently small interval around the number 3.

A further example of branching alongside propagation is shown in Figure 1.2. There, blue arrows denote propagation, whereas red arrows denote branching. Every node denotes a search space throughout the solving process. The image assumes all the solutions are desired. White nodes denote undecided nodes, while subsumed spaces (solutions) are shown in green, and failed spaces are shown in red. The final, quadratic constraint $(x-2.4)(x-5) > 0$ is a parabola having zeroes at points 2.4 and 5. Thus, it disallows values for x from the interval $[2.5, 5]$. The image assumes that it is impossible to propagate using this constraint, but that it is possible whether the constraint certainly holds, or is certainly violated on some given interval. Note that, in this example, binary branching is carried out in such a way that no values are repeated in both branches.

1.3 Interval Constraint Programming

If constraint programming is to be used for variables whose domains are subsets of a set of real numbers, then their domains are usually represented as intervals within which their permissable values lie. Because of this, constraint programming with real-numeric variables is called interval constraint programming [4]. Interval bounds are usually internally represented as floating point values. In this representation, only finitely many values may be represented in any given interval.

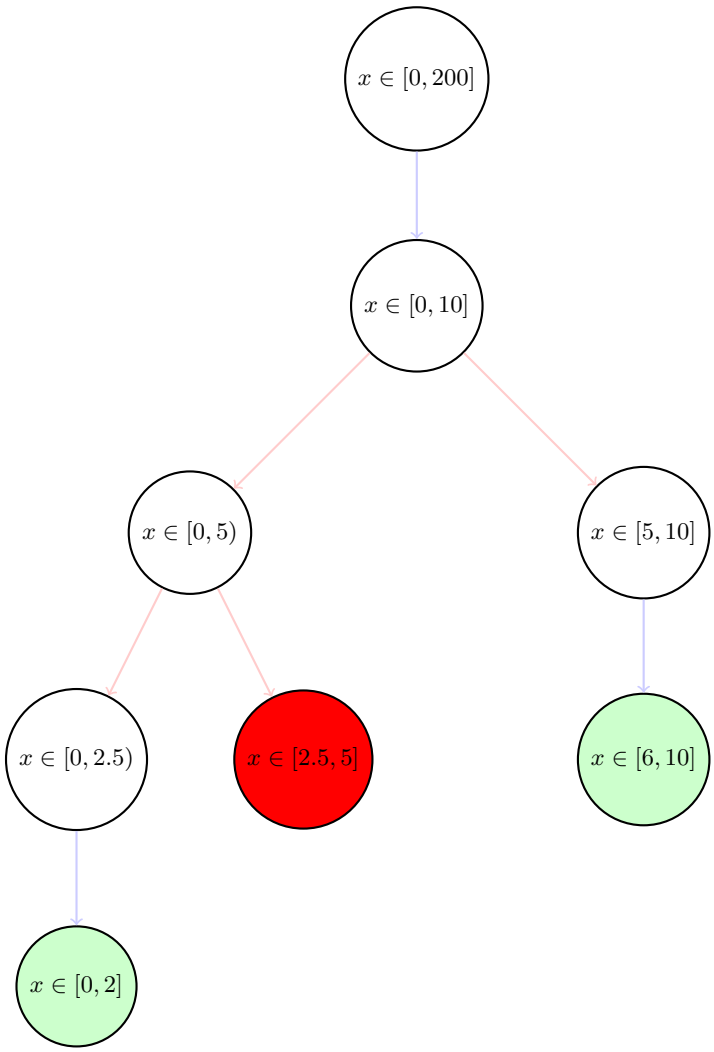
Constraints usually take the form of arithmetic statements involving variables, but may also describe more complex relationships.

An example of a constraint satisfaction problem solvable with interval constraint programming would be

$$x \in [0, 2.4] \tag{1.15}$$
$$y \in [0, 15.1] \tag{1.16}$$
$$z \in [0, 10] \tag{1.17}$$
$$x < y \tag{1.18}$$
$$x^2 + y^2 = z \tag{1.19}$$
$$e^x + \sqrt{y} + z^3 = 1. \tag{1.20}$$

sharing one value between the two branches will usually not significantly affect performance, but may make the implementation easier to understand and more straightforward.

Figure 1.2: Propagation and Branching Tree for the Constraints $\{x \leq 10, x \leq 2 \vee x \geq 6, (x - 2.4)(x - 5) > 0\}$, where initially $x \in [0, 200]$



1.4 Constraint Programming Systems

There are many software frameworks for constraint programming. Many are designed to facilitate constraint programming to the user, by relieving the user of implementing the details that are general to all or most of the constraint programming problems, and allowing the user to focus on specifying the problem at hand. Some of the commonly used constraint programming systems and libraries are Comet [17], Gecode [19], Prolog [14], ECLiPSe [8], IBM ILOG CPLEX CP Optimizer [13].

From those systems, Comet and Gecode focus on the constraint programming with constraints in arbitrary form (as long as the implementation for a particular constraint form exists) - Comet is a programming language, while Gecode is a framework for C++. Prolog is a logic programming language, focusing on logical inference, and ECLiPSe is its superset. IBM ILOG CPLEX is a mathematical programming framework, allowing only constraints in a particular form (namely, linear and quadratic inequalities). It is able to exploit the specific facts about those forms to use very efficient algorithms (such as Simplex) to solve the problems.

Gecode

Gecode (Generic Constraint Devolopment Environment) is a toolkit for developing constraint-based systems and applications. It is written as a library in C++, to be used also in C++. However, interfaces to other languages, such as Python, also exist. It is open for extensions, portable to different operating systems, and efficient (it won various awards for performance [5]). Its extension to interval constraint programming is the focus of this paper.

1.5 3D Reconstruction Problem

Interval constraint programming is suitable for solving the 3D reconstruction problem, which is used as a demonstration for the described extension.

Three-dimensional reconstruction (in the sense used in this paper) is the search for the coordinates for some points in space, whose locations are constrained by some constraints. For each vertex (point), approximate location is known (within a certain bounding box). Furthermore, it is known, for every triple of vertices, whether it forms a face. Additionally, geometric constraints to the faces and vertices are known - an example of such constraint may be that two faces are parallel, or that two pairs of faces enclose equal angles.

This problem is interesting because it is possible to extract such constraint information from a series of two-dimensional images of some object, taken from different angles [9]. Then, combined with interval constraint programming, a three dimensional reconstruction of an object can be derived.

Moreover, this problem has proven itself to be excellent test case for the developed extension, as one can easily verify the correctness of the results, both visually and computationally. Furthermore, instances, in particular synthetic ones, are an excellent benchmark of the system as well.

CHAPTER 2

Interval Constraint Programming

This chapter explores the unique features of the *Interval Constraint Programming*. Mathematical fundamentals of interval constraint programming are the mathematical concept of *interval*, and *interval arithmetic* which enables one to do calculations with intervals. Those concepts are explained in Section 2.1.

Furthermore, unique features of an interval constraint program are explored in Section 2.2. Then, constraint propagation methods for interval constraint programming are explored in Section 2.3, and a method to solve interval constraint satisfaction problems is shown in 2.4.

Finally, Section 2.5 introduces the consistency notions. Here, more general *node consistency* and *arc consistency* are introduced, which are applicable to a general constraint satisfaction problem. Then, it is shown how the notion of arc consistency is relaxed to define *hull consistency* and *box consistency*, the two consistency notions normally associated with interval constraint programming. Moreover, a formal method for enforcing both of those consistency notions is given.

2.1 Interval Arithmetic

A *closed interval* is a set of real numbers, defined by its lower and upper bounds. Exactly the set of real numbers that are greater than or equal to the lower bound, and less than or equal to the upper bound are part of the set. An *open interval* is the set defined like its closed counterpart, but not containing its bounds. Furthermore, it is possible that one bound be open, whereas the other is closed. When not specified otherwise, *interval* will denote a closed interval.

Interval arithmetic is a method of calculation such that the result of some expression, is an interval, representing a range of possibilities. Such a calculation is interesting in cases where one does not know the exact value, but within some margin of error - or, to keep track of the possible floating-point error that can be accumulated in the series of floating-point calculations.

Then, for some mathematical expression, or a function, if its arguments are given as intervals, it is possible to define the value of the expression as the interval including all the values expression could take for any combination of the values from the domains of its arguments.

Suppose that there are two intervals $\mathbb{X} \in \mathbb{I}$ and $\mathbb{Y} \in \mathbb{I}$, where $\mathbb{I}$ denotes the set of all real intervals. Then, for any binary operation between real numbers $\circ : \mathcal{R}^2 \to \mathcal{R}$,

$$\mathbb{X} \circ \mathbb{Y} = \{z \mid \exists x \in \mathbb{X}, \exists y \in \mathbb{Y} \text{such that } z = x \circ y\}. \tag{2.1}$$

This defines the smallest possible set that defines all the possible results of the operation $\circ$ between any combination of arguments from $\mathbb{X}$ and $\mathbb{Y}$. Note, however, that the above set is not necessarily an interval (this will be the case if the function $\circ$ is not continuous). This means that this set may, in general, consist of arbitrarily many discontinuous values - which might not be representable. Thus, instead of using this set directly in implementation, its *hull*, that is, the smallest interval containing all the values in the above set are used, as it can be defined using only two values. This relaxes the notion somewhat, however, this trade-off is necessary in order to be able to efficiently handle the arithmetic operations.

From this point on, when not otherwise stated, method using the hull rather than the narrowest possibility set will be meant. Nevertheless, since most of the common operators are continuous, the narrowest set and its hull often coincide (notable common exception to this being the division operation, when the interval of the divisor contains 0). Hull operation is usually denoted by $\square$.

As mentioned above, the most common use of interval arithmetic is to track floating point errors, and to handle uncertainty with measurements. When measuring some physical quantity, instead of assuming that some measured value (measured with some instrument introducing some error) is exactly as read from the instrument, one could take the value to be in some interval containing the measured value, where other values in the taken interval account for the possible errors introduced by the instrument. If one were to use interval arithmetic in further calculations with this interval, one would, in the end, obtain an interval containing all possible values of the expression being calculated, without ignoring the introduced error.

Other common use is in tracking floating point errors. While computing with floating point values, numeric errors are often introduced - but their maximum magnitude can always be tracked. Thus, an expression with floating point values can be evaluated using interval reasoning to yield all possible values occuring due to floating point errors, rather than an exact number.

In constraint programming, it is interesting to know whether a constraint is satisfied. Since the domains of variables are represented as intervals, interval arithmetic has to be used to calculate their values and to check this.

Singleton

An interval represents an underlying set of real numbers. However, it may also be the case that the interval represents only one value. Such an interval is called a singleton.

Due to floating-point errors, an interval with bounds very close together (tolerance being determined by a particular implementation) is usually treated as a singleton.

2.2 Interval Constraint Satisfaction Problem

An interval constraint satisfaction problem is a set of variables whose domains are intervals, along with a set of constraints on them.

The constraints have to be defined using such variables and real constants, and there has to exist a procedure that can decide whether the constraint is satisfied in a space that is reduced to a single point. When a space has more than a single point, decision procedure of the constraint is free to report unknown.

Restriction on the decision procedure is, though, not a severe one - it simply means that a constraint has a meaning. There would be little use of using a constraint on some variables, without being able to check whether it is satisfied for some concrete numbers.

2.3 Interval Constraint Propagation

As stated above, every constraint must at least have a procedure to decide whether it is surely satisfied in a space containing only a single point. However, if it is able to do more, then the procedure may be able to eliminate large number of points from a space as infeasible. This is beneficial from the performance point of view.

If a procedure can report that a constraint is (certainly) satisfied in spaces that have more than one point, then it would have eliminated the branching that would be required to generate spaces with just one point, which are subspaces of the current space. An example of this would be a constraint $x < 3$, with $x \in [0, 2]$. There can exist a decision procedure that would recognize this constraint as satisfied for any of the many (mathematically infinitely many, but finitely many in machine representation) values in the domain of x (namely, in this case, checking whether upper bound of x is less than 3). In this case, the tree would be pruned by accepting a space, on which branching could be performed without that observation.

Furthermore, if a procedure can detect that some case is unsatisfiable in some space, exploring further subspaces is unnecessary, and the tree can be pruned by discarding all subspaces of the current space. An example of this case would be a constraint $x < 3$, with the domain of x being $[5, 19]$.

A step further is that such procedure (called *propagator* in following text) associated to the constraint is able to eliminate subspaces from the current space, consisting of points that surely violate the constraint. This would prune a search tree by eliminating every space between the currently explored space and the one generated through variable domain subset elimination (*propagation* in following text).

Consider, for example, the constraint $x < y, x \in [1, 3], y \in [1, 2]$. Values larger than 2 for x will result in the violation of this constraint regardless of the value of y (taken from its domain), so this subrange may safely be removed from the solution space, as no solution can assign a value to x from that range and still satisfy all constraints.

By shrinking the search space as much as possible, one speeds up the search - thus, this is a very important duty of any constraint programming system. The reason for the speed-up is that pruning shrinks the search space, and in turn, the branching tree, thus limiting the number of nodes that will have to be visited in a tree. Intuitively, it is beneficial to remove values that

cannot be in the solution as soon as possible - if they are removed later, work will be duplicated in each of the subspaces. However, from the correctness point of view, duty of a propagator is merely to decide satisfiability when the considered space consists only of a single point (all variables are *assigned*) - and, if any pruning is carried out, no value may be pruned for which there exist values of other variables such that the constraint would be satisfied for them. In all other cases, it is free to report unknown. It is not necessary for a propagator, if it prunes, to prune all values that can correctly be pruned. Its duty is (in order to remain correct), though, to merely refrain from pruning the values that may satisfy the constraint along with any other combination of values for other variables. For example, consider a simple constraint satisfaction problem defined as follows

$$x \in [0, 5] \tag{2.2}$$

$$x \leq 2. \tag{2.3}$$

Here, a propagator can easily infer that $x \in [0, 2]$. However, a propagator is allowed to make a weaker inference[1] - for example, it can infer that $x \in [0, 2.5]$, without compromising correctness. It is, though, not allowed to infer $x \in [0, 1.5]$ since such an inference would leave out admissable values from the interval $(1.5, 2]$.

2.4 Solving Interval Constraint Satisfaction Problems

A constraint satisfaction problem is, in general, solved with the tree search and propagation procedure. This is, in essence, a divide-and-conquer procedure. Generally, one starts with a search space, and attempts to prune the values from the space that cannot be part of the solution, thus getting a subspace of the original search space. This is the propagation part. Then, when no further propagation is possible, one would branch the propagated space into multiple subspaces. Then, the propagation, and again branching would be recursively applied to the branches, until a solution is found, or some other stopping criterion is satisfied.

2.5 Consistency Notions for Interval Constraint Programming

Introduction

A consistent system is one that does not contain a contradiction. Generalized to constraint programming, local consistency is a property of a constraint satisfaction problem that can be enforced through transformations that change the search space without changing the solution space. Such transformations are the transformations pruning such values from the search space that are certain to violate at least one constraint.

There exist various different consistency notions, the most well known in constraint programming in general being node and arc consistency. However, those consistency notions are not generally applicable to interval constraint programming.

[1]The reason for making a weaker inference would, normally, be the computation cost, or the infeasibility of making the strongest possible inference, usually because of the complexity of the underlying operation. In this example, strongest inference can, of course, be made just as easily as any weaker inference.

Node Consistency

Node consistency is a property that may hold between a constraint and a variable. It requires that every unary constraint on a variable is satisfied by all values in its domain. Enforcing it involves removing the values from the domain of a variable which do not satisfy the constraint. Usually, node consistency can be enforced in polynomial time. Formally, variable x, with domain D_x is *node consistent* to the unary constraint $c(x)$ if and only if

$$\forall x \in D_x \; c(x) \tag{2.4}$$

where $c(x)$ is an unary predicate.

For example, consider a variable $x \in \{1, 2, 3, 4\}$, and a constraint $C := x < 3$. Variable x is not node consistent with the constraint C, but it can be made node consistent by removing the values 3 and 4 from the domain of x.

Arc Consistency

Arc consistency is a property that may hold between a constraint and a pair of variables in the constraint. It requires that, for every value in the domain of one varible, there exist a value in the domain of the other variable so that the constraint is satisfied between those two values. Formally, two variables x, y, with domains D_x and D_y respectively, are *arc consistent* with respect to constraint $c(x, y)$ if and only if

$$\forall x \in D_x \; \exists y \in D_y \text{ s.t. } c(x, y) \quad \wedge \quad \forall y \in D_y \; \exists x \in D_x \text{ s.t.} c(x, y) \tag{2.5}$$

where $c(x, y)$ is a binary predicate.

For example, consider a constraint $x < y$, over variables $x \in \{1, 2, 3\}$, $y \in \{1, 2, 3\}$. This setup is not arc consistent, since, if $x = 1$, there is no smaller value for y. But, by removing value 1 from the domain of x, this setup will become arc consistent.

It is possible to generalize arc consistency to k-ary relations. This generalization is usually called hyper-arc consistency, or simply generalized arc consistency.

Hull Consistency

In interval constraint programming, domains of the variables are, in principle, infinite (since every interval contains infinitely many real numbers). In practice, the domains are not infinite, as there are only finitely many representable real numbers within some interval. Furthermore, it is impractical to track, for every possible number, whether it is in the domain. Therefore, an interval floating point representation is normally used for variables representing real numbers (interval variables). As a consequence, the representing intervals may be slightly wider than the actual interval to be represented, as well as that discontinuous intervals are disallowed (arrays of subintervals that are interpreted as an union and would allow for discontinuous intervals are normally not used in interval constraint programming).

Because of this, and the fact that not all numbers can be represented with a floating point machine representation, it is normally infeasible to enforce arc consistency when working with intervals.

Hull consistency is introduced as a relaxation of arc consistency with which it is easier to work (than with arc consistency) in case of interval variables. It is the direct approximation of arc consistency for floating point values. Arc consistency is not computable in general as the actual arc consistent bound might not be present in the set of numbers used for machine representation. However, if the arc consistent domain is approximated with the smallest enclosing machine-representable hull, hull consistency is obtained. Two variables are hull consistent with respect to a constraint if their domains are the hulls of the arc consistent domains with respect to the same constraint. Formally, two variables x, y, with domains D_x and D_y respectively, are *hull consistent* with respect to constraint $c(x, y)$ if and only if

$$\forall x \in \square(D_x)\ \exists y \in \square(D_y) \text{ s.t. } c(x,y) \quad \wedge \quad \forall y \in \square(D_y)\ \exists x \in \square(D_x) \text{ s.t.} c(x,y) \tag{2.6}$$

where $c(x, y)$ is a binary predicate, and $\square$ is the hull operation.

Due to very large number of possible machine-representable numbers in the domains, it is normally not feasible to compute hull consistent intervals by directly applying the definition, that is, by iterating the values. Rather, reasoning about the functional building blocks in the constraint is usually used to calculate hull consistent intervals efficiently.

For example, consider a constant $x+y = z$, with $x \in [0, 1], z \in [0, 100]$. One could discover that z is not hull consistent - through interval arithmetic, one can discover that the expression on the left hand side $(x + y)$, for current domains, is $[0, 2]$ because of the properties of addition. Furthermore, due to properties of equality operator, right hand side has to also be from $[0, 2]$. Therefore, hull consistent interval for z would be $[0, 2]$ (a subset of its previous interval).

Enforcing Hull Consistency

Suppose an expression of the form $x_1 \circ x_2 \circ ... \circ x_n = z$, where $\circ$ is a computable operator, and $x_1, ..., x_n$ and z are variables, with known domains.

The usual algorithms for enforcing hull consistency are *HC3* and *HC4*. HC3 [18] is an algorithm similar to *AC-3* [15] (an algorithm for enforcing arc consistency), which tackles the more complex user constraints by decomposition to simpler constraints. HC4 [3] is an extension of HC3, which is able to handle user constraints directly, rather than decomposing them to simpler constraints - but is otherwise similar to HC3. Pseudocode of the canonical HC4 algorithm is given in the Algorithm 1.

A procedure for enforcing hull consistency that is applicable for interval constraint programming and implementation in Gecode on a constraint denoting this expression is slightly different than the procedure described in Algorithm 1. The main part of the canonical HC4 algorithm is a loop that executes the *HC4revise* function, but in Gecode, Gecode kernel takes care of propagator scheduling. Therefore, there is no possibility to control the loop by the programmer. Thus, to adapt HC4 algorithm to Gecode, one only needs to implement the appropriate *HC4revise* routine for the constraint being implemented, and the kernel will take care of calling it appropriately. Nevertheless, such *HC4revise* routine should follow the outline given in Algorithm 2, as this will yield the strongest inferences possible. Thus, the following text focuses only on the revision process, or equivalently, the enforcement of hull consistency for a particular constraint.

Enforcement of hull consistency (HC4 revision) can be divided in two steps, forward evaluation and backward propagation. Forward evaluation is the reasoning about the necessary domain

Algorithm 1 HC4 algorithm

Input: List of real constraints $c_1, c_2, \ldots, c_n$, search space $S = I_1 \times I_2 \times \cdots \times I_m$
Output: Pruned space S_p

```
S_p ⇐ S
C ⇐ {c_1, c_2, ..., c_n}
while S_p ≠ ∅ ∧ C ≠ ∅ do
    c ⇐ choose one c_i ∈ C
    S'_p ⇐ HC4revise(c, S_p)
    if S'_p ≠ S_p then
        C ⇐ C ∪ {c_j | ∃x_k ∈ Var(c_j) ∧ I'_k ≠ I_k}
        S_p ⇐ S'_p
    else
        C ⇐ C \ c
    end if
end while
```

Algorithm 2 HC4revise

Input: real constraint $c = r(x_1, x_2, \ldots, x_n)$, search space $S = I_1 \times I_2 \times \cdots \times I_m$, $n \leq m$
Output: Pruned space S_p

```
S_p ⇐ S
for all x ∈ x_1, x_2, ..., x_n do
    ForwardEvaluation(x, S_p)
end for
BackwardPropagation(x, S_p)
S_p ⇐ □S_p
```

of the right-hand side, given the knowledge about the domains of the variables on the left hand side. Backward propagation is the reasoning about the domains of variables on the left-hand side, given the knowledge about the domain of the right hand side.

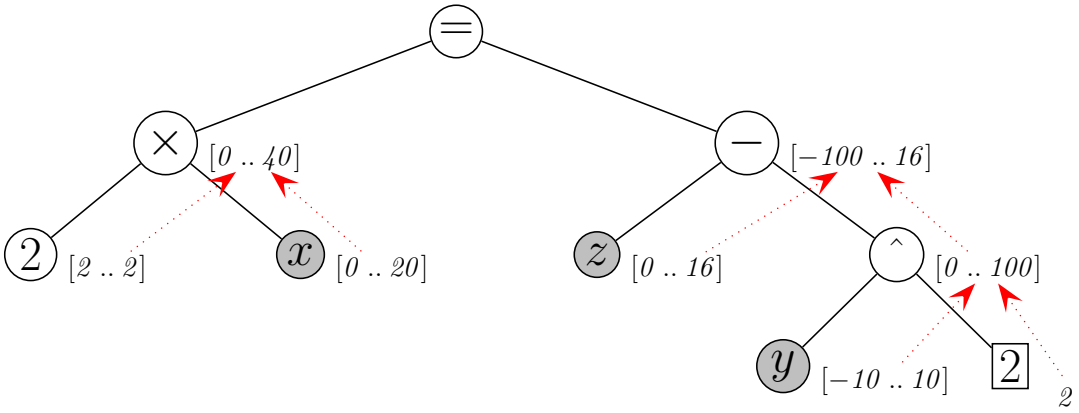
Operator $\circ$ and Interval Arithmetic

As already stated, when reasoning about the operator $\circ$ (which is a placeholder for a concrete operator), given intervals $\mathbb{A}$ and $\mathbb{B}$, the result of $\mathbb{A} \circ \mathbb{B}$ should be such that, for any choice of values a from $\mathbb{A}$ and b from $\mathbb{B}$, their result $a \circ b$ will be in the interval $\mathbb{A} \circ \mathbb{B}$. However, this interval may contain other values. It is, though, beneficial for the interval to contain as little other values as possible, as other values weaken the inferences that can be made (but do not compromise correctness).

Forward Evaluation

Forward evaluation step is carried out by calculating $\mathbb{L} := x_1 \circ x_2 \circ ... \circ x_n$. Note that, here, $\mathbb{L}$ is by definition equal to the original expression, with $\mathbb{L}$ taking the place of z. However, $\mathbb{L}$ is an

Figure 2.1: Annotated tree for the forward evaluation of the constraint $2x = z - y^2$ (taken from [3])



interval, which is dependent exclusively on the domains of the variables $x_1, ..., x_n$, whereas z is a variable, potentially having its own domain (before propagation) different than $\mathbb{L}$. Then, z is set to the intersection $z \cap \mathbb{L}$. The rationale behind this step is that z may not take values that were not in its original domain, but through the constraint, z is also constrained to take values only from $\mathbb{L}$, so intersection is necessary. An example annotated tree for the forward evaluation of the constraint $2x = z - y^2$ is given in figure 2.1.

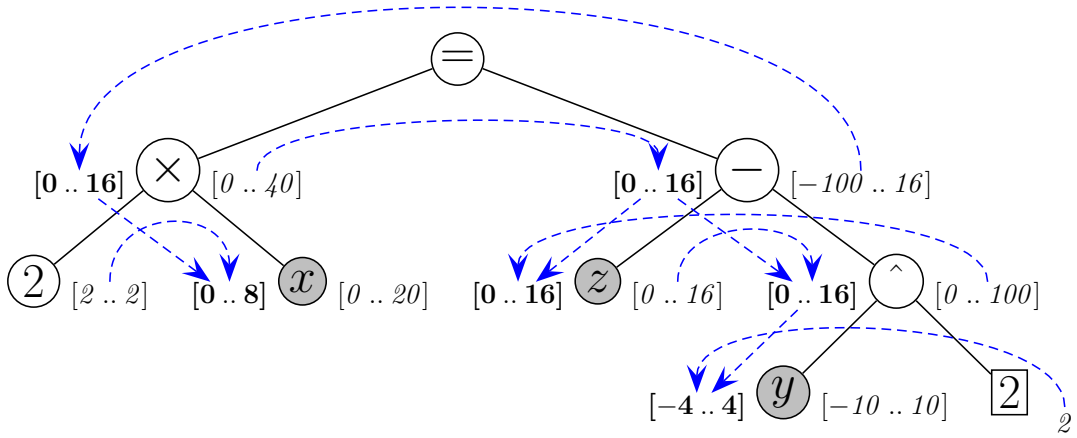
Backward Propagation

If the domain of z were set just to $\mathbb{L}$, backward propagation in this sense would be redundant, as the domains of the variables on the left-hand side would be used to infer $\mathbb{L}$, which would then be used to infer the new domains on the left-hand side. However, since other constraints (or initial domain) might have narrowed z to be narrower than $\mathbb{L}$, often, this new knowledge of the domain of z may be used to remove inconsistent values from the domains of the variables on the left-hand side.

In order for backward propagation to be possible in this sense, the operator $\circ$ should have an inverse of its every operand. These requirements imply an operator $\circ^{-1}$, such that if, for intervals $\mathbb{A}$, $\mathbb{B}$ and $\mathbb{C}$, if $\mathbb{A} \circ \mathbb{B} = \mathbb{C}$, then $\mathbb{A} \in \mathbb{B} \circ^{-1} \mathbb{C}$ and $\mathbb{B} \in \mathbb{A} \circ^{-1} \mathbb{C}$. Note that stronger inverses, $\circ_s^{-1}$, for which $\mathbb{A} \circ \mathbb{B} = \mathbb{C}$ would imply $\mathbb{A} = \mathbb{B} \circ_s^{-1} \mathbb{C}$ and $\mathbb{B} = \mathbb{A} \circ_s^{-1} \mathbb{C}$ are usually not possible in interval arithmetic. Weaker version can still add strength to inferences, with stronger inferences being possible with the inverse returning smaller intervals (of course, while not violating the necessary conditions).

Note that if the operator $\circ$ is not commutative, there may exist two inverses $\circ_l^{-1}$ and $\circ_r^{-1}$. For the expression $a \circ b = c$, the equivalent statements with the corresponding inverses are $a = b \circ_l^{-1} c$ and $b = a \circ_r^{-1} c$. If the operator $\circ$ is not commutative, when using backward propagation, respective inverse has to be used for every operand, instead of the common one. If only one of those inverses exists, backward propagation might still be possible, albeit only

Figure 2.2: Annotated tree for the backward propagation of the constraint $2x = z - y^2$ (taken from [3])



for some operands (namely, the leftmost, or the rightmost, depending on whether $\circ_l^{-1}$ or $\circ_r^{-1}$ is available).

Backward propagation then consists of a step for every variable on the left-hand side. Consider the step for variable x_i. First, let $\mathbb{T}_i := x_1 \circ x_2 \circ ... \circ x_{i-1} \circ x_{i+1} \circ x_{i+2} \circ ... \circ x_n$, that is, be equal to the operator $\circ$ being folded over every variable on the left-hand side except for x_i. Then, given commutativity of $\circ$, one obtains $x_i \circ \mathbb{T}_i = z$. Given the necessary properties of the inverse operator $\circ^{-1}$, one can infer $x_i \in z \circ^{-1} \mathbb{T}_i$. Thus, the domain of x_i can be set to $x_i := x_i \cap \mathbb{T}_i$.

When the number of variables in a constraint is large, it may seem tempting to optimize the above operation by saving $\mathbf{L}$ from forward propagation, and calculating every $\mathbb{T}_i$ from it as $\mathbb{T}_i = \mathbb{L} \circ_s^{-1} x_i$. While this would be correct, given the existence of the strong inverse operator $\circ_s^{-1}$, it need not produce non-redundant inferences if weak inverse is used in its place (and strong inverse often does not exist). Naturally, other optimizations are possible (for example, saving the intermediate values between backward propagation for different variables).

An example annotated tree for the backward propagation of the constraint $2x = z - y^2$ is given in figure 2.2.

Other constraint forms

For some operators, especially non-commutative ones such as subtraction or division, arbitrarily many operands are not useful - such constraints would have the form $x \circ y = z$, which is a subcase of the aforementioned general case.

If the operator $\circ$ is not commutative, the commutative backward propagation procedure given above is not applicable, but in such cases constraints are usually given with only two operands on the left hand side (for example, $a/b = c$). In those cases, backward inference can easily be made that the domain of a is equal to the domain of $b \circ^{-1} c$. For the example $a/b = c$, one

can easily obtain that the backward propagation procedure might use the equivalent statement $a = bc$ to shrink the domain of a.

Furthermore, it is interesting to consider relations through hull consistency (example being $a < b$). Those could, with some care, be rewritten in the general form, however, they can also be handled directly.

Consider the constraint $x \leq y$. Then one can simply remove all values from x that are greater than the upper bound of y, and all values from y that are less than the lower bound of x. Having this operator available, one can write $x < y$ as $x \leq y + \delta$, where δ is a infinitesimal value (or next floating-point value in case of machine representation). $x \geq y$ can be defined as $y \leq x$, and analogously for greater than. Equality ($x = y$) can also be optimized by noting that domains of both variables must be the intersection of the domains.

It may be possible to allow some further, more complex constraints with hull consistency. However, their usefulness would be doubtful. Adding the possibility of more operators on the left-hand side would make backward propagation weaker, while adding more possibilities on the right-hand side would weaken the forward evaluation. Such propagation might not be able to prune some inconsistent values that simpler variant could prune.

However, disallowing more complex constraints is not saying that hull consistency cannot tackle them. When a more complex constraint is posted, it can be decomposed to simpler constraints, that can be directly handled by hull consistency. Such a decomposition is, in fact, the main idea for the algorithm HC3 [18].

For example, for a constraint $x^2 + y = z$, one would create new constraint, $x_{sq} = x^2$, and then post $x_{sq} + y = z$, which is equivalent to the initial constraint. Such a decomposition can be carried out for every complex constraint, however, when some variable occurs many times within a constraint, there might exist a more efficient approach than hull consistency, namely, box consistency.

Note that HC4 algorithm is, in itself, capable of directly handling more complex constraints. Yet, for any constraint form that one wishes to support, one must usually write a specialized propagation procedure. That, combined with possibly only weaker inferences being possible, makes more complex constraints less compelling to implement.

Box Consistency

Box consistency is a relaxation of hull consistency, in which reasoning about the functional building blocks is replaced by a refutation test over the bounds of an interval [18]. This means that, in order to find the box consistent bounds for some interval, one should first verify that the current bounds are not box consistent. If this is not the case, one would consider infinitesimally smaller interval, and check whether new interval is box consistent on both bounds. Then, one would keep repeating this until an interval is found which is box consistent from both sides. It is important, in the general case, that the step is infinitesimal, as otherwise some values may be left out.

However, considering an infinitesimally smaller interval is both impossible in floating point representation, and would yield in an infinite number of steps. It would, though, be possible (and sufficient, since one is limited to machine-representable values) to interpret infinitesimal step as a step to the next number in the direction of the other bound to obtain box consistent bounds.

But this would be very inefficient, so other methods have to be employed. Naive methods, such as binary search for the interval bounds are not correct in general, as they might miss some values, if the underlying function for a constraint is not monotonic. However, there exists a general method, namely, the Newton method, that iteratively converges around the desired interval, without missing any solutions.

It normally requires more computation to compute box consistent bounds in comparison to hull consistent bounds. This is because the refutation test in box consistency requires comparable computational effort to the whole forward evaluation phase in hull consistency, and many refutation tests are usually necessary to find box consistent bounds. However, it is not possible to apply hull consistency directly to more complicated constraints, namely, to those in which a single variable appears more than once. To get around this, one could decompose such constraint into simpler ones for which hull consistency can be enforced, but it is sometimes nevertheless more efficient to use box consistency.

Interval Newton Method

Newton method (also known as *Newton-Raphson method*) is a numerical method used for finding succesively better approximations of the zeroes of a real-valued function. This has as a consequence the restriction that constraints, for which box consistency needs to be computed, be in the form $c(\vec{x}) = 0$, where $\vec{x}$ is the variable vector. This is, however, not a significant restriction, as most functions can usually be rewritten to be in this form through basic arithmetic transformations. Inequalities can be rewritten as equalities by introducing slack variables.

The method in one variable for real numbers is derived the mean value theorem, which can be expressed as

$$\exists c \in [a, b] \text{ s.t. } f'(c) = \frac{f(b) - f(a)}{b - a} \tag{2.7}$$

where $f(x)$ is a continuous function on the interval $[a, b]$, where $a < b$, and differentiable in the open interval (a, b). Since one seeks zeroes of the function $f(x)$, one can assume that $f(a) = 0$ (if this is not the case, Newton method will not converge). Then, through algebraic manipulations, one obtains

$$f'(c) = \frac{f(b)}{b - a} \tag{2.8}$$

$$f'(c)(b - a) = f(b) \tag{2.9}$$

$$b - a = \frac{f(b)}{f'(c)} \tag{2.10}$$

$$b - a = \frac{f(b)}{f'(c)} \tag{2.11}$$

$$a = b - \frac{f(b)}{f'(c)} \tag{2.12}$$

where a is the sought zero, and b is a known point. By transforming that into a series, i.e. by replacing a with x_{n+1}, b with x_n, and c with x_n as well, one obtains the defining formula for

the series of the Newton method. It is a series giving successively better approximations of the function $f(x)$, defined as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.13}$$

where x_0 is an initial guess for a zero of the function $f(x)$ whose zero is sought, and $f'(x)$ is its derivative. The process is repeated, until a sufficiently accurate approximation is reached.

However, when working with intervals, Newton method is defined slightly differently. Interval variation of the Newton method is known as the *Interval Newton method*. For an univariate function $f(x)$ with derivative $f'(x)$, with $\mathbb{I}_0$ signifying the initial guess (an interval), series of successively better approximations is given as

$$\mathbb{I}_{j+1} \leftarrow \mathbb{I}_j \cap \left(\gamma(\mathbb{I}_j) - \frac{f(\gamma(\mathbb{I}_j))}{f'(\mathbb{I}_j)} \right) \tag{2.14}$$

where $\gamma(\mathbb{I})$ is a function returning arbitrary value from the interval $\mathbb{I}$. Initial interval $\mathbb{I}_0$ needs to be large enough to surely contain a solution, as every next step can only narrow the interval as it returns an intersection with an interval from a previous step.

Problems when $0 \in f'(\mathbb{I}_j)$

In a Newton step, Equation (2.14), $f'(\mathbb{I}_j)$ is a denominator, and it evaluates to an interval since its argument is an interval $\mathbb{I}_j$. Thus, it may not contain 0, as the division with 0 is undefined. This requirement is equivalent to the statement that f is monotonic on the interval $\mathbb{I}_j$. If this requirement is violated, further Newton steps in as given by Equation (2.14) cannot be taken. In those cases, instead of a Newton step, a different approach has to be followed.

This is often a significant problem, as Newton step is normally significantly faster than workaround approaches. Furthermore, many constraints used in practice violate the requirement that $0 \notin f'(\mathbb{I}_j)$.

An approach to tackle this problem is suggested in [12]. However, the implementation developed along with this thesis uses a somewhat different approach, which is described in Section 3.2.

Enforcing Box Consistency by Shaving

A state of the art algorithm for enforcing box consistency is the *Box Consistency by Shaving* algorithm [10]. Here, constraints in the form $c(\mathbf{x}) = 0$ are assumed, where $\mathbf{x}$ is an interval variable.

Often, though, a constraint is a function of more than one variable. However, a single iteration of bound revision by shaving (or other algorithms, such as BC3) affects only a single variable (one chooses arbitrarily at the beginning of the iteration), Therefore, in this section, it is assumed that $c(\mathbf{x})$ is an univariate function. This is because all other variables are treated as constants in a single iteration (their values are intervals, representing all the values for each variable).

Its pseudocode is given in the Algorithm 3. The following process describes narrowing of the bounds for a single constraint.

Firstly, one should check if the constraint is already surely violated, which is the case if and only if $0 \notin c(\mathbf{x})$. If so, then the current space is dismissed.

Then, one checks if the left bound is consistent. This is done by taking a very small interval (possibly infinitesimal, although some optimizations suggest slightly larger intervals) L, and evaluating the constraint for it. If $0 \notin c(L)$, then, left bound is inconsistent, and the interval can be narrowed from the left. Left half of the original interval is then taken, and checked for satisfiability. If it is satisfiable (constraint evaluated for the left part contains 0), a Newton step is carried out, and the result is stored as a result for the left-hand side. Otherwise, an empty interval is given as a result for the left-hand side.

One then repeats the analogous process for the right bound. Resulting interval is then the hull of the result for both sides. Note that the result is not necessarily box-consistent, but can be made box-consistent by repeating the process as long as the process modifies anything.

Algorithm 3 Enforcing Box Consistency by Shaving

Input:

$g : \mathbb{I} \rightarrow \mathbb{I}$ *# A function implements a constraint which is*
 # satisfied for some value x*, if* $0 \in g([x, x])$.

$\mathbf{I} \in \mathbb{I}$ *# An interval on which the largest box consistent interval w.r.t* g *is sought*

Output:

$\mathbf{I}_{bc} \in \mathbb{I}$, s.t. $\mathbf{I}_{bc} \subseteq \mathbf{I}$ and $\mathbf{I}_{bc}$ is box consistent w.r.t. g

(*left_consistent, right_consistent*) $\Leftarrow$ (*false*, *false*)
$\mathbf{I_{bc}} \Leftarrow \mathbf{I}$
while $\mathbf{I_{bc}} \neq \emptyset \wedge (\neg left\_consistent \vee \neg right\_consistent)$ **do**
  $(\mathbf{I_l}, \mathbf{I_r}) \Leftarrow split(\mathbf{I})$ *# normally a binary split*
  if $\neg left\_consistent$ **then**
    # Check if left bound is box consistent w.r.t. g*. This is the case if* $0 \in g(\underline{\mathbf{I_l}})$.
    # However, to prevent not detecting box consistent bound, small (epsilon)
    # environment is used ($[\underline{\mathbf{I_l}}, \underline{\mathbf{I_l}}^+]$ *instead of just* $\underline{\mathbf{I_l}}$*)*
    if $0 \notin g([\underline{\mathbf{I_l}}, \underline{\mathbf{I_l}}^+])$ **then**
      $\mathbf{I_l} \Leftarrow [\underline{\mathbf{I_l}}^+, \overline{\mathbf{I_l}}]$ *# Bound not box consistent, discard it from consistent interval*
      if $0 \notin g(\mathbf{I_l})$ **then**
        $\mathbf{I_l} \Leftarrow \emptyset$ *# Try to dismiss whole left half, if inconsistent*
      else
        # Choose a point for the numerator in the Newton step,
        # normally the bound in Enforcing Box Consistency by Shaving
        $P \Leftarrow \underline{\mathbf{I_l}}$
        $\mathbf{I_l} \Leftarrow \mathbf{I_l} \cap \frac{\mathbf{I_l} - g([P,P])}{g'(\mathbf{I_l})}$ *# One Newton step*
      end if
    end if
  else
    $left\_consistent \Leftarrow true$
  end if

continued ...

Algorithm 3 Enforcing Box Consistency by Shaving (continued)

 # This is the right subinterval part, analogous of the left case above.
if $\neg right\_consistent$ **then**
 # Check if right bound is box consistent w.r.t. g
 # This is the case if $0 \in g(\underline{\mathbf{I_r}})$
 # However, to prevent not detecting box consistent bound, small (epsilon)
 # environment is used ($[\overline{\mathbf{I_r}^-}, \overline{\mathbf{I_r}}]$ *instead of just* $\overline{\mathbf{I_r}}$*)*
if $0 \notin g([\overline{\mathbf{I_r}^-}, \overline{\mathbf{I_r}}])$ **then**
$\mathbf{I_r} \Leftarrow [\underline{\mathbf{I_r}}, \overline{\mathbf{I_r}^-}]$ *# Bound not box consistent, discard it from consistent interval*
if $0 \notin g(\mathbf{I_r})$ **then**
$\mathbf{I_r} \Leftarrow \emptyset$ *# Try to dismiss whole right half, if inconsistent*
else
$P \Leftarrow \underline{\mathbf{I_r}}$
$\mathbf{I_r} \Leftarrow \mathbf{I_r} \cap \frac{\underline{\mathbf{I_r}} - g([P,P])}{g'(\mathbf{I_r})}$ *# One Newton step*
end if
end if
else
$right\_consistent \Leftarrow true$
end if
$\mathbf{I_{bc}} \Leftarrow \square(\mathbf{I_l} \cup \mathbf{I_r})$ *# Box consistent interval found is the hull of left and right parts*
end while

CHAPTER 3

Implementation

Alongside this thesis, a working extension of *Gecode* for interval constraint programming was developed. This chapter gives the most important technical information about the developed implementation.

Section 3.1 gives an outline of the Gecode architecture, the abstract principle that describes how Gecode functions, and the main Gecode concepts that have to be extended in order to extend the implementation. The next Section, 3.2, explains how the system was extended to implement the concepts given in Chapter 2.

3.1 Gecode Architecture

Gecode is structured into its kernel that must always be included and used when working with it, and other add-ons which are not essential. These include support for some variable types, namely, integer and integer sets, and other extensions such as graphical library *Gist* used to display the search data, *FlatZinc*, a parser for a low level modeling language, and a command-line driver. Even though none of the add-ons are essential to compiling a program using Gecode, little can be modeled without including at least one variable type. Figure 3.1 schematically shows the (original) Gecode architecture. This work introduces a new variable type, namely, floating point, or interval variables (sometimes used interchangeably, because intervals are internally used to represent ranges of floating point values). Along with this, it introduces all the necessary propagators and branchers (which will be explained in the next paragraph). Figure 3.2 schematically shows the Gecode architecture with the interval extension.

Gecode enables the user to model and solve constraint satisfaction problems through the use of variables, propagators, branchers and search engines. All of these can also be easily programmed to extend Gecode itself, as it was done for the extension described in this paper.

General Gecode concepts will be briefly outlined. A detailed instruction for Gecode can be found in the document Modeling and Programming with Gecode [19].

Figure 3.1: Original Gecode Architecture (from [19])

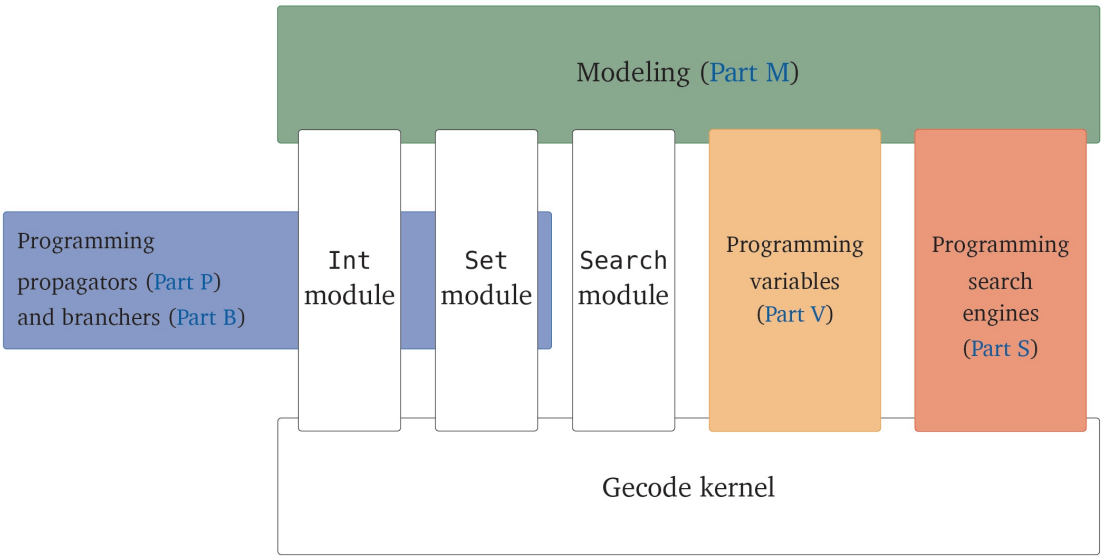
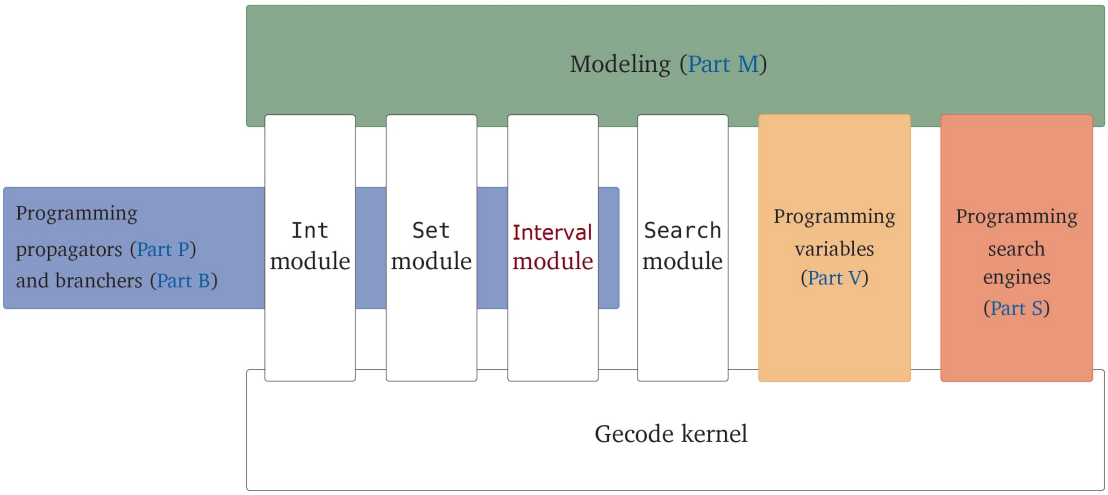


Figure 3.2: Gecode Architecture extended with ICP



Space

Space is a concept in Gecode that denotes the set of constraints, along with domains for all variables. There is a minor difference between the meaning of the term *space* previously in this thesis, and in Gecode terminology. In the thesis, space includes the space defined by the set of variables through their domains. However, in Gecode, class *Space* and its subclasses actually keep track of the propagators along with the variable domains. However, this distinction is mostly technical, as it makes no difference in the discussion how the propagators are represented internally. Thus, it will be clear from the context to which meaning the term space refers.

Variables

A variable is the unknown value in the constraint system. It has a domain - a set of values from which it can take its values. User's objective will typically be to find the set of values for some, or all, the variables in a given system, such that all of those values satisfy all the constraints. Examples of variable types are integer variables, interval variables and integer set variables (variables whose value is a set of integers).

Each variable has a class that represents its implementation, whose name usually has the suffix *VarImp* (for example, integer variable implementation class is called *IntVarImp*). This class handles all the technical details of the variable.

On top of this, Gecode offers variable class, which is effectively an interface to the variable implementation, and is used to pass variables to functions. It is designed to be lightweight, so that it can be passed by value to functions efficiently - and it contains virtually only the pointer to the underlying variable implementation, as well as some accessor methods.

When writing propagators, domains of variables often have to be modified. Usually, an user will not work directly with *VarImp* classes, and *Var* interfaces do not offer modification facilities. Because of this, there is also view class for every variable type, normally suffixed with *View* (e.g. IntView). View interface should never be used for modeling - it should be reserved to be used in implementations of propagators.

Propagators

Propagators are the implementations of the constraints in Gecode. Their task is to infer whether the constraint is satisfied by all of the values in the domain, for each constraint, given a domain of each variable. Furthermore, they should prune all the values from the domain that are certain to violate the implemented constraint.

If, during the pruning, whole domain of the variable is pruned, then, obviously, no value satisfying the constraint exists, so that solution can be dismissed. On the other hand, if all of the values in all of the variables can satisfy a constraint, then the propagator is said to be *subsumed*, and it can be *disposed* (for performance reasons), as it will never propagate again.

The propagators interface to variables through the View classes.

From the propagation method of the propagator (called *propagate*), a propagator may return that the constraint is certainly satisfied (*subsumed* in following text), or failed (not satisfiable). Furthermore, as previously stated, propagator does not need to give a conclusive answer (unless

all variables are assigned). In such cases, it may return that it has either reached a fixpoint, or not. Then, a propagator lets the kernel know that it cannot be conclusively determined that the current state certainly satisfies, or fails the constraint. If subsequent propagation with this propagator could further affect the search space, the propagator is not at fixpoint, and it is at fixpoint otherwise. If a propagator returns that it is not at fixpoint, it will be scheduled to be executed again.

Propagators that reach a fixpoint after every execution are called *idempotent*. Every propagator could be made idempotent by simply running its propagation code over and over again and tracking whether it has changed any of its variables, until no change can be made any more. However, in some cases (especially when a propagation step is computationally expensive), it is beneficial for a propagator to only do a fraction of the work it can do. Then, opportunity can be given to other, computationally cheaper, propagators, while delaying the more expensive one until all the cheaper ones have been executed.

When created, the propagators in Gecode subscribe to at least one of their views (which refer to a variable). Through subscribing, Gecode kernel is notified that a propagator needs to be scheduled for execution whenever at least one of the variables to which it is subscribed is changed. A subscription may have different propagation conditions, scheduling a propagator for execution on any change, or only, for example, on lower bound change. Possible subscription types depend on the particular variable implementation.

The Gecode kernel, upon a change of a variable, will schedule the relevant propagators for execution.

However, the order of their execution is dependent on the Gecode kernel. Gecode offers the user the possibility to implement a cost function in the implementation of a propagator, whose idea is to approximate the computational complexity of that particular propagator.

If the propagator implements a cost function, the kernel may attempt to execute less computationally expensive propagators first. However, one should not rely on any particular execution order - but the kernel guarantees that all propagators that have been scheduled will have been executed before the next branching step.

Such approach to propagation makes some traditional consistency-enforcing algorithms unsuitable to be directly implemented within the propagation method of a propagator. More specifically, algorithms are often given as a loop, which attempts to find an inconsistent value within a domain of some variable, and revise the domains. In such algorithms, this process is repeated until nothing more can be revised.

In Gecode, however, when the propagation method is executed is dependent on the variable subscriptions, and a single propagator does not have access to all the variables within the space. Thus, traditional consistency-enforcing algorithm has to be adapted slightly. The adaptation usually involves making the propagate method only responsible for the revision step, while the propagator scheduling by Gecode kernel takes the place of the outer loop. Algorithm 4 gives the outline of what Gecode kernel does.

Branchers

Branchers are the parts of the system responsible for dividing the domains of the variables. Usually, the initial domains for all the variables will not satisfy at least some of the constraints

Algorithm 4 Abstract Propagator Scheduling and Pruning Routine

Input: List of propagators $c_1, c_2, \ldots, c_n$, search space S
Output: Report a required number of solutions, or assert this many do not exist

```
S_p ⇐ S
# Propagator may be in four states, namely, subsumed, failed, at_fixpoint, and
# not_at_fixpoint. Subsumed means that the propagator is surely satisfied, while
# failed means sure failure. Propagator is at fixpoint if it is unable to decide
# whether it is satisfied, but cannot make any further inferences in this space,
# whereas a propagator not at fixpoint might still be able to prune some values
# from a space.
while ∃i  state(c_i) = not_at_fixpoint do
    c ⇐ choose c_i ∈ C, such that state(c_i) = not_at_fixpoint
    S'_p ⇐ propagate(c_i, S_p)
    for all c_j ∈ c_1, c_2, ..., c_n, state(c_j) = at_fixpoint
      ∧ c_j subscribed to at least one variable with values in S_p \ S'_p do
        state(c_j) ⇐ not_at_fixpoint
    end for
    S_p ⇐ S'_p
end while
# Now no propagator can propagate again.
# All propagators are now either subsumed, failed, or at fixpoint.
if S_p = ∅ then
    report no solution
    stop
else
    if ∀j state(c_j) = subsumed then
        report S_p is a solution
    else
        S_1, S_2, ..., S_m ⇐ branch (S_p)
# Different search engines will choose S_j in different orders
        for S_j ∈ S_1, S_2, ..., S_m do
            Abstract Propagator Scheduling and Pruning Routine((c_1, c_2, ..., c_n), S_j)
            if stopping conditions satisfied then
                stop
            end if
        end for
    end if
end if
```

(otherwise, the problem would have already been solved). This is likely to be true even after the initial propagation. Thus, a procedure is needed that will nevertheless look for the solution. While exhaustively trying all the possible values for all variables may be possible in some cases (those with finite many possibilities), it is not a particularly appealing approach from an efficiency point of view.

Therefore, Gecode relies on the fact that sometimes whole groups of values can be dismissed. For example, in $x \leq 2$, all values in the interval $(2, \infty)$ for x are not feasible.

A brancher is an algorithm that, given a space, gives possible options how the space could be split in multiple subspaces, so that the union of all the subspaces would be the original space. It determines the shape of the search tree.

Choice of a brancher is a very important one, primarily because of performance reasons.

Internally, for any space, Gecode requires that a brancher be able to calculate its possible choices of subspaces. It is furthermore required that those choices can be serialized in such a way so the choices can be repeated without the space for which they are computed necessarily being the same. This is important because of possible need for backtracking, and Gecode's recomputation methods.

Search Engines

Search engines apply branchers and propagators to spaces in order to find subspaces that would hopefully satisfy all the constraints. They determine order in which the search tree is explored, and may make inferences (such as bounds pruning) to not explore all the possible options. The goal of the search engine is to find a feasible solution, or multiple feasible solution, or to report that no feasible solution exists after the search space is exhausted.

3.2 Extending the System

To implement interval reasoning, the first step was implementing the classes needed for interval variables themselves. However, since nothing can be modelled only with variables and without constraints, propagators needed to be implemented for them too. Furthermore, branching used for other types of variables could not be used either, so a brancher was implemented as well. Existing search engines were functional for this problem, so a new search engine was not needed. When floating-point values is mentioned, double-precision floating-point representation is meant if not specified otherwise.

Interval Arithmetic

In order to implement the system, the *Boost Interval* library [11] was used. This library handles virtually all interval arithmetic. It overloads most of the common arithmetic operators used in the language of implementation (C++), and makes the correct reasoning about the possible intervals in any expression as effortless as in expressions with ordinary numbers.

Boost Interval library is a library designed specifically to facilitate interval arithmetic. The main class in the library is the *interval* class (*boost::numeric::interval*). This class defines lower and upper interval bounds, but type of the bounds is templated, so the user can use own type

to represent the bounds (for example, arbitrary precision numbers). However, for performance reasons, in this implementation, C++ *double* data type was used for the bounds.

Furthermore, the class is templated with additional policies parameter. The policies consist of two classes, namely, rounding and checking. Rounding policies handle rounding matters, while checking policies deal with empty intervals, detecting infinite numbers or invalid values and similar matters. Thus, the library is highly configurable, as the user can specify exactly how the system should behave in handling those delicate issues. However, the library already gives some pre-defined policies, so that most users must not implement their own.

Additionally, library offers possible and certain equality and inequality tests. Possible (in)equality test functions return true if there exist values within the compared intervals for which the test could be true, while certain tests return true only if the test is true for any pair of values taken from each interval, respectively. It offers explicit functions for this functionality, but also offers overloaded operators. User may choose whether overloaded operators are certain or possible by a proper namespace declaration.

Interval data type

The *Interval* data type is the fundamental building block of the extension. It is a type defined as a particular case of Boost *interval* class. Its definition follows.

```
typedef boost::numeric::interval
    <
        double,
        boost::numeric::interval_lib::policies
        <
            boost::numeric::interval_lib::save_state
            <
                boost::numeric::interval_lib::rounded_transc_opp<double>
            >,
            boost::numeric::interval_lib::checking_base<double>
        >
   >
  Interval;
```

Policy choices, however, have only a marginal influence on the extension, as the check for whether an interval is assigned has to be performed manually, within a given tolerance. Furthermore, policies offer choices on how to handle *NaN* values, but the extension does not rely on those values. The reason not to use simpler, default, policies, was because with them explonentiation functions incorrectly (gives a compile time error).

Comparing floating-point values

Due to accumulation of floating-point errors, comparing floating-point values through equality may cause incorrectness of the program. Therefore, a comparison routine *fp_eq* was used to determine if two floating-point numbers are equal. It would return true if the numbers were

different by at most some tolerance[1], and false otherwise. The check verifies whether their absolute difference is lower than the threshold. Relative comparison is often considered superior, due to floating exponent in floating-point values, however, in this case, it was inapplicable as it did not recognize imporatant cases within a Newton step.

This relation is used throughout the extension for the bounds of the interval, to see whether it is assigned (a singleton). An interval is considered a singleton if its bounds are floating-point equal to one another, in the manner defined above. When not otherwise stated, comparison in this manner is meant whenever two floating-point values are compared in the remainder of the work.

SymbolicC++ Library and respective modifications

To make the system more accessible to the user, it should be possible to construct more complex arithmetic expressions directly. Furthermore, a Newton step (see Equation (2.14)) requires the derivative of an arbitrary function (namely, the function used to represent a constraint).

To accomplish both of these requirements, some kind of computer algebra system has to be used. Such a system should make use of operator overloading, so that the user can specify expressions such as, for example, $2xy + z\hat{}2$ directly in code. Then, compiler could, because of overloaded operators, be able to construct the expression tree from the expression automatically, making it much less tedious for the user. Furthermore, the system should be able to symbolically differentiate an arbitrary expression, as required for the Newton step.

In this implementation, *SymbolicC++* library [22] was chosen, which satisfies the above requirements. It is a lightweight computer algebra system that comes only as header files, and is as such easily portable, and practical to modify. All of these facts made it the preferred choice for this implementation. Its most basic building block is a symbol, which may be a constant, a variable or a function. The symbol class was extended, making it possible to attach a Gecode interval variable (as described in the next section, 3.2) to a symbol.

Furthermore, expressions are represented as a more general, *Symbolic* class, used to denote an arbitrary expression. Interval evaluation and interval variable retrieval methods were added to the *Symbolic* class. The system uses a class for every operator type it supports, where this class is a subclass of the *Symbolic* class. Thus, a specialization of the interval evaluation and variable retrieval methods had to be written in many different cases.

When using this extension, every variable in a constraint expression should have an associated interval variable. Even though it is technically possible to construct an expression not satisfying this, it would normally not be useful with this extension. In further text, it will thus be assumed that all *SymbolicC++* variables have associated Gecode interval variables.

Interval evaluation method will return an interval for each expression, corresponding to the hull of possible values the expression may take. The system is able to infer this, since every variable has a known domain, obtainable through the associated interval variable. Overloaded variant of this method involves similar logic, but with the exception that it takes a variable and a value as a parameter. It then evaluates the expression in the analogous manner, but replaces the

[1]This tolerance is, by default set to 10^{-8}. However, user is free to modify this value to obtain different resolutions.

variable given as a parameter (if present) with the value given as a parameter - i.e. evaluates the expression, given a single substitution.

Variable enumeration methods list all the variables in the expression. This is immediately not obvious from the expression, as it may be intertwined with various operators, so the methods were written recursively, with cases for every supported expression type. A method returning all the variables, and a method returning only all unique variables are offered.

Symbolic expressions are used to post box consistency constraints as described in the Section 3.2.

Variables

A new variable implementation had to be made in order to implement the interval reasoning. To implement new variable type, one has to compile Gecode from its source code. A configuration must be set, and most of the code required is automatically generated by Gecode build script. However, some additional logic had to nevertheless be implemented.

Naming convention used for other Gecode variables, especially IntVar was followed, so the new variable type was named *FloatVar*. This is due to the fact that, even though it uses interval reasoning, the variable actually represents floating point values, much in the same way as IntVar represents an integer, even though it holds information about multiple possible integers. Thus, new variable implementation was named *FloatVarImp*, while new variable and view interface were named *FloatVar* and *FloatView*, respectively. Those classes offer the expected functionality. Furthermore, some derived views were implemented.

FloatVarImp

FloatVarImp represents the implementation of the variable. It keeps a variable of type Interval, representing the current domain of the variable. It offers a constructor through the Interval type, or through specifying the interval bounds through two floating-point values, which then constructs the interval.

Variable implementation offers the operations to modify the current domain. Among that are the functions retaining only the values greater than (gq) and smaller than (lq) some value, which work as in the integer variant. Furthermore, an intersect function is present, which intersects the current domain with the interval parameter (or an overload with two bounds). Domain replacement functions are not present, as the propagator is not allowed to expand a domain of a variable.

Finally, accessor functions to get the lower bound (min()), upper bound (max()), interval width (width()) and whether interval is assigned (assigned()) are provided. A copy of the interval itself can be obtained with the interval() function.

FloatVar

FloatVar class offers an interface, and functionality similar to the one offered by the IntVar class, only with bounds being double-precision floating point values, instead of integers. As

with IntVar, this class offers accessor functions to getter functions from the *FloatVarImp* class, and it merely forwards them to the underlying interpretation.

FloatView

As with *FloatVar*, *FloatView* is an interface to the underlying variable implementation, but view classes can modify variable implementations as well. It does this by merely forwarding the function calls to the variable implementation class.

Derived Views

MinusView, *ConstSingletonView*, and *OffsetView* are offered. *MinusView* and *OffsetView* work exactly like their integer counterparts, but offsets can be floating-point values. *ConstSingleton-View* is an analogy of *ConstView* from the integer variant - it is in essence a constant interval view consisting just of a singleton.

Subscription Propagation Conditions

Propagation conditions are defined analogously as for *IntVar*s. Every variable modification operation is able to distinguish between cases when it has changed only lower, or upper bound, when both have been changed, or when the domain has been reduced to a singleton. Furthermore, variable modification operations are able to detect if nothing has been modified through the operation.

Propagators

Various propagators were implemented for the system. They can be divided into hull and box consistency propagators. One propagator needed to be implemented for every supported operation for hull consistency, whereas there is only one propagator for box consistency. The following description describes the propagation step for each propagator. Whenever at least one of variables changes (to which the propagator is subscribed), it is scheduled for execution. Then, the propagation procedure will be surely execute before the next branching (possibly after other propagators).

Other methods of a propagator are straightforward - the constructor creates the subscriptions, for all variables given as arguments. Post function calls the constructor, possibly after it has determined that the constraint is already not surely violated by a simple test.

Disposal method cancels the subscriptions, and calls the dispose method of the Propagator base class - as prescribed in the Gecode documentation [19].

Cost functions return the combination of the number of variables (binary, ternary or linear) according to the number of variables in every propagator. Furthermore, additional information may be given whether the propagation cost is low or high. Hull consistency propagators return low here, whereas box consistency propagator returns high as it usually has more work per propagation iteration.

Hull Consistency Propagators

Three different forms of hull consistency propagators have been implemented. The propagators implemented are relational propagators in form $x \sim y$, where $\sim$ is a binary relation, general-form arithmetic operations in the form $x_1 \circ ...x_n = z$, namely, addition and multiplication, and other arithmetic operations with form $x \circ y = z$.

Addition and Multiplication In case of the most complex form, $x_1 \circ ...x_n = z$, propagator is implemented exactly as described in the Section 2.5. First, the domain of the right-hand side is shrinked to the intersection of its previous domain, and the domain of the operation on the left-hand side (addition, or multiplication, respectively). Then, domain for each x_i is shrinked to the intersection of its previous domain and the domain of the right-hand side, from which one subtracts, or divides, respectively, the sum, or the product, of the left-hand side without x_i.

For example, given three *FloatVar* variables, x, y and z, one can post a constraint equivalent to $x + y = z$ with the command

```
sum(home, x, y, z);
```

where *home* refers to the current Home space (a familiar Gecode concept).

Relational Operators Relational operators are the operators in the form $x \sim y$, where $\sim$ is an arbitrary binary relation. As specified in the Section 2.5, equality is implemented by specifying that the domain of both x and y is equal to the intersection of the both on every step. Less than or equal relation is defined as in the Section 2.5, and other operators are accordingly defined.

Binary Operations Subtraction, multiplication, exponentiation, and inverse exponentiation have been implemented in a binary form, $x \circ y = z$. This is a special case of the $x_1 \circ ...x_n = z$, as described in the Section 2.5. For example, given two *FloatVar* variables, x and y, one can post a constraint equivalent to $x \leq y$ with the command

```
leeq(home, x, y);
```

where *home* refers to the current Home space.

Symbolic Expressions for Hull Consistency Constucting hull consistency constraints from symbolic expressions (see 3.2) is not possible. The reason for this is that virtually any constraint that has a more complex form than those postable directly by the available functions would, because of the Gecode structure, require decomposition into multiple constraints. Thus, it might be the case that the user inadvertently posts many constraints, thinking only one will be posted, possibly compromising performance. Because of this, the decision was made to split the post functions for the two consistency options, so that the user will always be aware which is used.

Box Consistency Propagator

There is only one box consistency propagator, able to propagate an arbitrary box consistency constraint.

Specifying Constraints As stated in the Section 2.5, box-consistency constraints can only be given in the form $c(\mathbf{x}) = 0$. Thus, the propagator class internally stores only the left-hand side of the expression, and equality to zero is implied.

To construct an expression for the box consistency propagator, the extended *SymbolicC++* library (as presented in Section 3.2) is used.

To post a constraint, an user is given various variants of the post function, the simplest one being only giving the left-hand side expression (where an expression is an instance of *Symbolic* class from *SymbolicC++* library). In this case, the posted constraint is that the given expression is equal to 0.

However, an user may also specify both left and right-hand side expressions, along with a relation (equality, less than, less than or equal, greater than, greater than or equal). In this case, a new expression is constructed internally for the actual propagator, namely, by subtracting the right-hand side from the left-hand side. Furthermore, slack variables are introduced where applicable (inequalities).

This frees the user from having to know the required form for box consistency constraints, but nevertheless, forces the user to post constraints in the form $A \sim B$ giving A, B and the operator $\sim$ as three parameters to the post function, where the operator $\sim$ is specified through an enumeration in the propagator class (options being equal, not equal, less than, less than or equal, greater than and greater than or equal).

Considerations when Posting Constraints through Symbolic Expressions The symbol $\wedge$ represents the *XOR* operation in *C++*, which is a low precedence operation. However, in *SymbolicC++*, $\wedge$ represents the exponentiation operator, which, mathematically, has greater precendence than other common operators such as $+$. As a consequence, user should mind that, when creating expressions, even though $\wedge$ signifies the exponentiation operator, its precedence is equal to that of the standard *C++ XOR* operation. Thus, to create the constraint $x^2 + y$, user should write $(x\wedge 2) + y$, as writing just $x\wedge 2 + y$ would create the expression x^{2+y}.

Furthermore, *SymbolicC++* has $==$ equality symbol, which is still allowed in the constraints. However, this operator in fact creates an expression $A == B$, which will evaluate to 1 if A is (certainly) equal to B, to 0 if A is certainly different than B, and to an interval $[0, 1]$ (hull of $[0, 0]$ and $[1, 1]$) if nothing can be inferred. This means that posting $A == B$ would actually post the disequality constraint between A and B (namely, $(A == B) = 0$), while $(A == B) - 1$ would post the equality constraint between them (note that the right-hand side is implicitly equal to 0). Disequality can also be posted by putting $A == B$ on the left-hand side, and stating equality as a relation with 1 (namely, $(A == B) = 1$) on the right-hand side, which then works as expected.

Certain and Possible Constraint Satisfaction A constraint $c(\mathbf{x}) = 0$ is *certainly satisfied* if all of its variables are singletons, and $0 \in c(\mathbf{x})$. Furthermore, even if the variables are not all assigned, but $c(\mathbf{x}) \subset [-\epsilon, \epsilon]$, the constraint is also certainly satisfied, where ϵ is some small threshold value.

A constraint is *satisfiable* (*possibly satisfied*) if $0 \in c(\mathbf{x})$ and not all variables have been assigned (if all variables are assigned, we can determine whether the constraint is actually satisfied or not).

A constraint is *unsatisfiable* if it is not satisfiable.

Propagator Creation To post a constraint, the user uses the post function which transforms the constraint in the form $c(\mathbf{x}) = 0$ (as described in the Section 3.2). Then, post function of the propagator checks satisfiability, and trivially fails the space if the constraint is already unsatisfiable. Then, it checks if it is certainly satisfied (if so, propagator has nothing to do), and if this is not the case, it posts the propagator.

In the constructor, a subscription to all variables in the expression is created - the same subscriptions are cancelled in the destructor.

Propagation Initially, the propagation method first checks unsatisfiability, and fails the space if the constraint is unsatisfiable. Then, if a constraint is certainly satisfied, the propagator is dismissed as subsumed. If neither is the case, box consistency by shaving algorithm is carried out.

The function considers unique unassigned variables in the expression. For every such variable, it proceeds with a step of the box consistency by shaving enforcement algorithm step.

If the step has modified the variable, the propagation method terminates, returning that it has not reached a fixpoint. This notifies the kernel that it may still have some work to do, but leaves the possibility that other, less computationally expensive propagators may first propagate.

If the step has not modified the variable, the procedure proceeds to the next unassigned variable. If no such variable exists, propagator returns subsumption - as all variables have been assigned, and unsatisfiability has not been caught.

Handling the Case When $0 \in f'(\mathbb{I}_j)$ As stated in Section 2.5, Newton step cannot be taken if $0 \in f'(\mathbb{I}_j)$. In that case, this implementation attempts to eliminate inconsistent part of an interval, either from the left side, or the right side. This implementation first attempts to narrow from the left, and, afterwards, from the right. The process to eliminate inconsistent values from the right side is symmetrical, but otherwise analoguous to the process from the left, so the following presentation shall assume one attempts to narrow from the left. Note that here the function f is the function implementing the constraint, value of which is 0 if a constraint is satisfied, and is nonzero otherwise. Its derivative can no longer be used, since a Newton step cannot be applied.

The narrowing process is, basically, a binary search for an interval that is inconsistent from the left of the interval (assuming left narrow). It is not a search for the largest inconsistent interval. The implementation bisects the interval $\mathbb{I}_j$ into left and right halves, $\mathbb{I}_j^l$ and $\mathbb{I}_j^r$, respectively. Then, the implementation checks if $0 \in f(\mathbb{I}_j^l)$. If this is the case, one cannot dismiss the interval $\mathbb{I}_j^l$ as inconsistent, as it is sure to contain at least one consistent value. In this case, the procedure is recursively carried out on the left half of the interval, $\mathbb{I}_j^l$. If, however, $0 \notin f(\mathbb{I}_j^l)$, one can immediately note that $\mathbb{I}_j^l$ is inconsistent, and the left bound of $\mathbb{I}_j^r$ is a result of the left binary

narrowing. Note that the result of this procedure is a number that might (does not necessarily have to) be a left consistent value. However, the result will be less than or equal to the smallest left consistent value (it will not dismiss consistent values), but different to the left bound of the original interval $\mathbb{I}_j$. The (left) binary narrowing procedure is given in Algorithm 5.

Note that this procedure possibly terminates before it has done all the narrowing that it is capable of doing. This is intentional, as the binary narrowing procedure is normally much more computationally heavy than a Newton step. Thus, it is important that some, minimal narrowing be done so that the implementation does not become stuck. However, it is not vital that the full extent of the narrowing be done immediately, as this gives a chance to propagation of other constraints, or the same constraint on a different variable. Furthermore, it is also possible that the pruned values will cause that f' evaluated on a narrowed interval no longer contains 0, thus opening the possibility of further Newton steps.

Then, the final result is the hull of results that are obtained by left and right binary narrowing.

Algorithm 5 Binary Narrowing Procedure from the Left - *binary_narrow*

Input: Function f implementing a constraint, Interval $\mathbb{I}$, Original bound O
where the original bound O should be set to to $\underline{\mathbb{I}}$, for use in recursive
calls to determine whether any progress has been made

Output: Number l s.t. $l > \underline{\mathbb{I}}$ and $l \leq L$
where $\underline{\mathbb{I}}$ and $\overline{\mathbb{I}}$ are the left and right bound of an interval $\mathbb{I}$, respectively,
and L is the smallest left consistent value in $\mathbb{I}$ w.r.t f

$(\mathbb{I}_l, \mathbb{I}_r) \leftarrow bisect(\mathbb{I})$
 # Left side of the interval is unsatisfiable
 # Just return the left bound of the right half (a half of an interval was dismissed)
if $0 \notin f(\mathbb{I}_l)$ **then**
    return $\underline{\mathbb{I}_r}$
else
    # Left side of the interval is satisfiable - this need not provide any useful info
    # Therefore, either split further, or leave it as is, in which case nothing new might
    # have been obtained. Split further only if no change was done thus far
    # (dismissing any number of inconsistent values from the left is satisfactory)
    if $O = \underline{\mathbb{I}}$ **then**
        return *binary_narrow*(f, $\mathbb{I}_l$, O)
    else
        return $\underline{\mathbb{I}_l}$
    end if
end if

Branchers

A branching step on interval variables is, as expected, bisecting the domain of a single variable, and exploring two subspaces with the variable domain being equal to the respective half in each,

with all other domains remaining unchanged.

The user may specify a branching as usual in Gecode, by calling the *branch* function, which was overloaded for the *FloatVar* variables. User may use the *FloatVarArgs* array to chain multiple branchings, or simply chain the calls.

A brancher in Gecode has three important functions to be implemented, namely, a choice function that chooses how to branch, a commit function that specifies how to implement a chosen branching, and a status function that checks whether a brancher has anything left to do.

Choice

The choice function loops through all the variables in the specified branching, and attempts to find the most suitable one by comparing their domains. Multiple criteria can be used to determine the most suitable variable, and the criterion used is specified by the user as the branching post parameter. An example of a criterion is to take the variable with the largest domain first. Then, the function creates a choice, which consists of an index of a variable which was chosen, along with two intervals corresponding to the two halves.

Commit

The commit function takes a choice as created above as a parameter, as well as whether to choose the left or the right option. Depending on the option, it extracts, respectively, the left or the right interval from the choice, as well as the variable. Then, the function simply reduces the domain of the variable to be the intersection of its previous domain with the extracted interval.

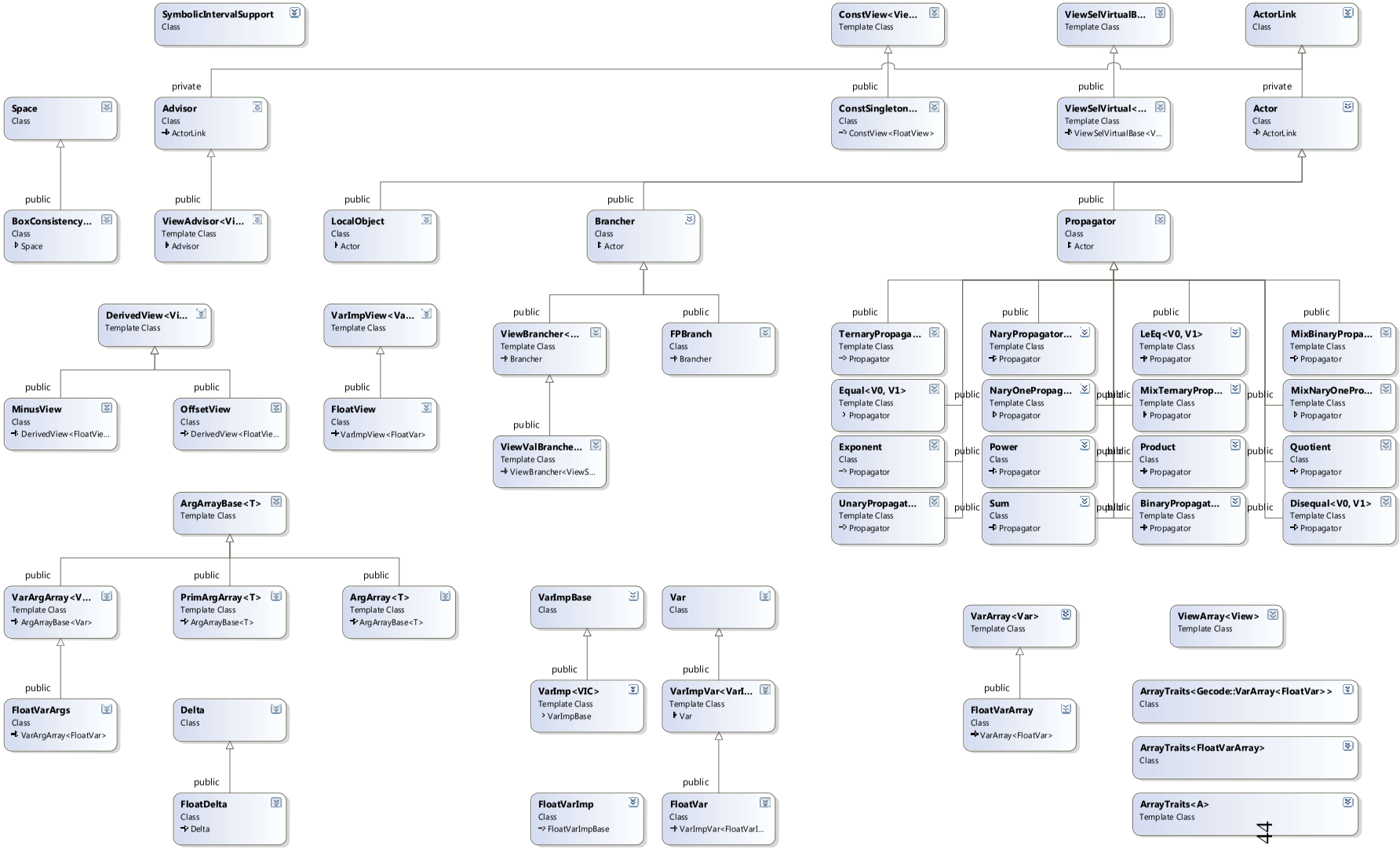
Status

Status function determines whether there is anything left to do. If there exist active propagators, and unassigned variables, the function will return that something is left to do (true), and otherwise false.

Other Functions

A brancher needs to implement an archiving method, which archives a choice. The implemented method simply takes the variable index, and the four bounds of the two intervals in a choice, and stores them in sequence. A complementary method for extracting choice reads them in sequence and reconstructs. Other methods, such as constructors and disposal method, were implemented exactly as specified in Gecode documentation [19].

Figure 3.3: Class Diagram of Gecode classes relevant to the extension, as well as extended classes



CHAPTER 4

Benchmarks

To test the correctness, and the speed of the system, known problems (from [6]) were modeled, and solved within the framework. Problems with multiple instances were chosen, in which one is able to choose a number of constraints defining the instance. Here, more relations (usually) means a more difficult instance. Solving increasingly difficult instances is a good benchmark for a framework, as it shows how a particular framework scales with the increasing difficulty of the instance.

All results given in this chapter, and in Chapter 5, were obtained on a computer with a Intel(R) Core(TM)2 Duo CPU T9300 @ 2.50 GHz, 4 GB of RAM, running a 64-bit Windows 7.

4.1 Broyden Banded

Problem Description

The first problem used to test the implemented extension is a classical problem for interval constraint programming, the *Broyden Banded* [6]. The problem is defined by the set of n equations, as follows:

$$x_i(2 + 5x_i^2) + 1 - \sum_{j \in J_i} x_j(1 + x_j) = 0 \tag{4.1}$$

where

$$J_i = \{j, j \neq i, \max(1, i - ml) \leq j \leq \min(n, i + mu)\} \tag{4.2}$$

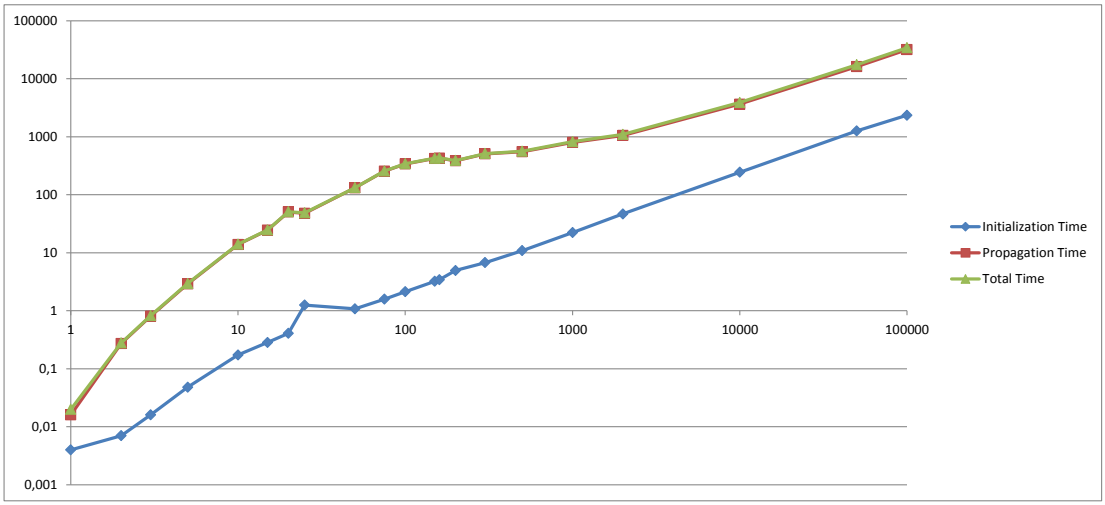
$$ml = 5 \tag{4.3}$$

$$mu = 1 \tag{4.4}$$

$$x_i \in [-100, 100], \forall i. \tag{4.5}$$

The above problem is solvable with box consistency alone, without branching [20]. Therefore, the problem was modeled only with box consistency constraints, with one constraint directly included in the model for each of the defining equations.

Figure 4.1: Broyden Banded - n vs Time (s)



Results

Table 4.1 shows the times (in seconds) the computer needed to solve a particular instance of the problem (in relation to a particular instance size n, that is, the number of defining equations). Epsilon value used for this experiment set is 10^{-9}. Initialization time is the time the system needed to set up all the constraints, that is, the work prior to actual propagation. This consists mainly of the overhead from the *SymbolicC++* library, since the construction of the constraint set itself is not computationally heavy. Propagation time is the time it takes to find the first (and only [6]) solution, after the initialization is complete. Initialization and propagation time are the only parts of the solving process, so, together, they amount to total solving time. The last column shows which proportion of the total time was spent in the actual propagation, as opposed to initialization overhead.

The data about initialization, propagation and total times are shown in Figure 4.1 (on a logarithmic scale), while the data about the proportion of time spent in propagation in relation to total time is graphed in Figure 4.1. Furthermore, Figures 4.3, 4.4 and 4.5 show the total time, initialization time, and propagation time, respectively, on a logarithmic scale. On those figures, linear interpolations of the result for the respective segment for $n = 100$ and $n = 100000$ are plotted as well, for comparison [1]

[1] Linear interpolation for the respective segment are the functions $f_{100}(n) = \frac{c_{100}}{100}\, n$ and $f_{100000}(n) = \frac{c_{100000}}{100000}\, n$, where c_{100} and c_{100000} are the actual values (of initialization, propagation, and total time, respectively) for $n = 100$ and $n = 100000$, respectively. In other words, f_{100} and f_{100000} represents how the function would behave if it were linear, and passing through the data point for $n = 100$ and $n = 100000$, respectively. Since the axis is logarithmic, linear function is no longer a straight line through the given points, so these serve to better put computation time in reference.

Table 4.1: Broyden Banded - Time to Solve with Box Consistency

n	Initialization	Propagation	Total	Propagation Time / Total Time
1	0,004	0,016	0,02	0,8
2	0,007	0,272	0,279	0,974910394
3	0,016	0,806	0,822	0,98053528
5	0,048	2,895	2,943	0,983690112
10	0,173	13,898	14,071	0,987705209
15	0,284	24,49	24,774	0,988536369
20	0,408	50,516	50,924	0,991988061
25	1,255	47,74	48,995	0,974385141
50	1,076	131,733	132,809	0,991898139
75	1,586	255,388	256,974	0,993828169
100	2,128	340,27	342,398	0,99378501
150	3,216	423,238	426,454	0,992458741
160	3,432	427,557	430,989	0,99203692
200	4,945	385,814	390,759	0,987345141
300	6,755	508,068	514,823	0,986878986
500	10,914	550,881	561,795	0,980572985
1000	22,247	801,81	824,057	0,973003081
2000	46,595	1055,125	1101,72	0,95770704
10000	244,965	3653,935	3898,9	0,93717074
50000	1264,53	16218,47	17483	0,92767088
100000	2342,37	31963,13	34305,5	0,931720278

Figure 4.2: Broyden Banded - n vs Proportion of Time Spent in Propagation

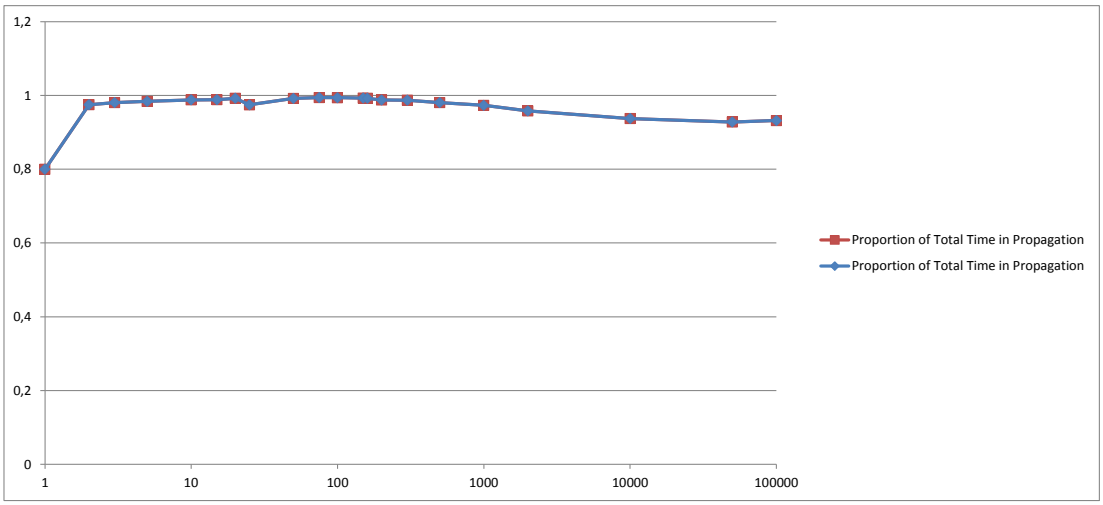


Figure 4.3: Broyden Banded - n vs Total Time (s)

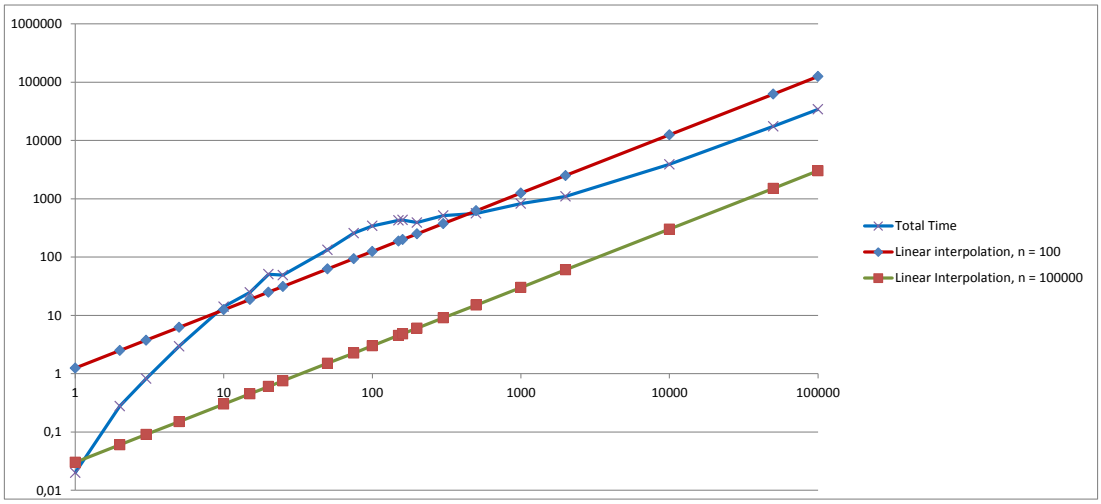


Figure 4.4: Broyden Banded - n vs Initialization Time (s)

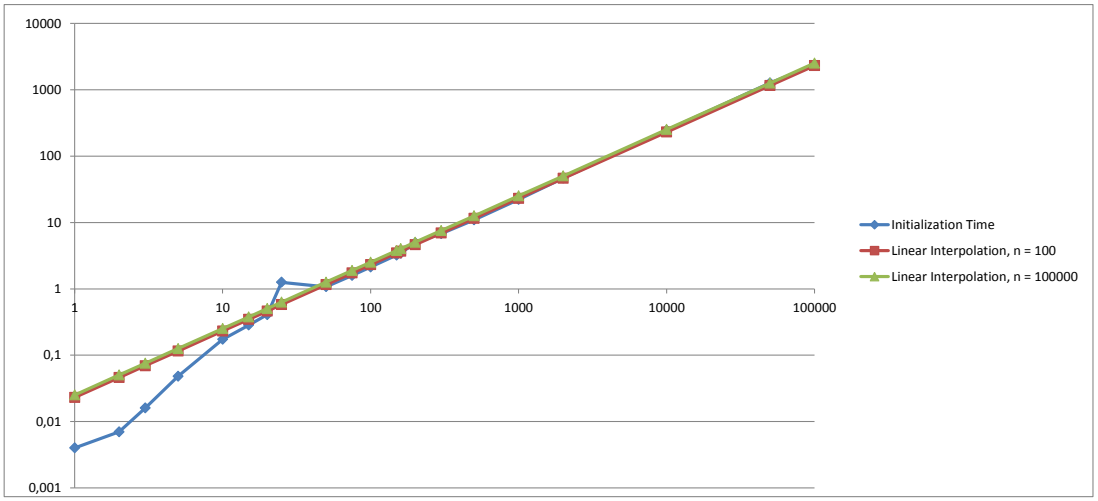
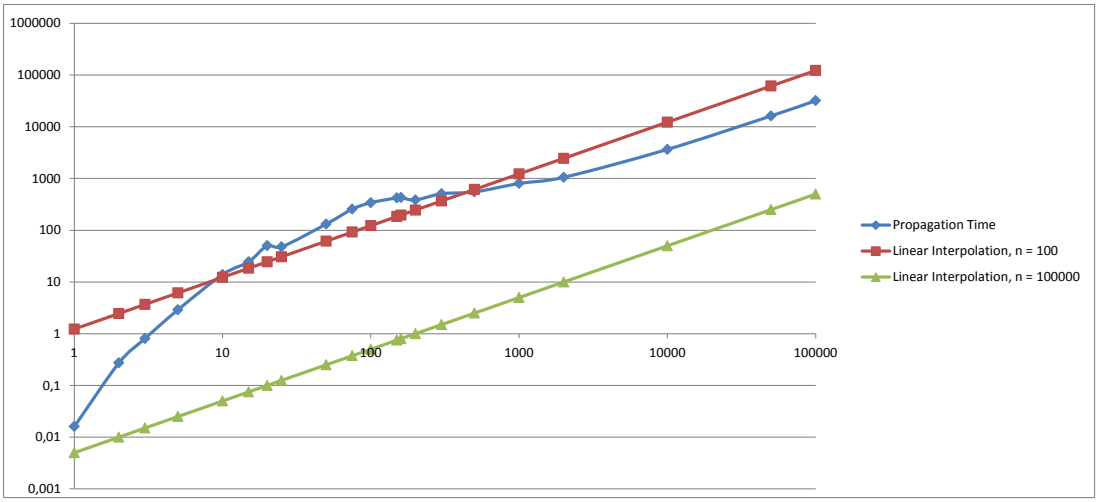


Figure 4.5: Broyden Banded - n vs Propagation Time (s)



Analysis of the Results

The implemented system was able to tackle even very large instances of the Broyden Banded problem within reasonable time. This is no doubt due to the fact Broyden Banded can be solved with propagation alone, without branching [20].

However, one can note that initialization time is significant compared to the propagation time. Furthermore, the initialization time was compared to the initialization time of a different interval constraint programming implementaton using a different library, and the initialization time in the other implementation was two orders of magnitude better.

This points out the fact that *SymbolicC++* may be suboptimal library for this application. Replacing *SymbolicC++* with some other library or design pattern such as expression templates may offer better constraint initialization time. However, *SymbolicC++* performs relatively well when evaluating expressions, and all other necessary functions with them - except for the initialization. As most real-life problems are not solvable by propagation alone, slow initialization need not necessarily be problematic.

Nevertheless, one can conclude that even bigger instances (e.g. for $n = 100000$) are still feasibly solvable with this method.

4.2 Broyden Tridiagonal

The next problem used for testing the system is the *Broyden Tridiagonal*, taken from [6]. The problem is defined by n equations, given by

$$(3 - 2x_i)x_i - x_{i-1} - 2x_{i+1} + 1 = 0 \tag{4.6}$$

where

$$x_0 = x_{n+1} = 0 \tag{4.7}$$

$$x_i \in [-100, 100], 1 \leq i \leq n. \tag{4.8}$$

The problem was modeled directly with box consistency constraints, with one constraint included for every constraint from the problem. $n + 2$ variables were used, with x_0 and x_{n+1} having only the number 0 in their domains. When branching, the variable with the widest domain was selected. For this problem, choosing the variable by degree is not a viable option, since all variables are present in all constraints (through their sum).

Results

Smaller instances of this problem have two solutions. For all considered instances, it was feasible to find the first solution, but the second was not found due to search requiring insufficient memory. In such instances, missing data is marked with a question mark.

Three different sets of experiments were carried out, varying the equality tolerance, and the *modification significance level* (MSL). Equality tolerance, or ϵ, is the minimum difference between two numbers that has to be present for the numbers to be treated as different, as described in Section 3.2. Larger ϵ values mean less resolution, as there are less possible numbers that can be considered, however, because of the same reason, search often proceeds faster.

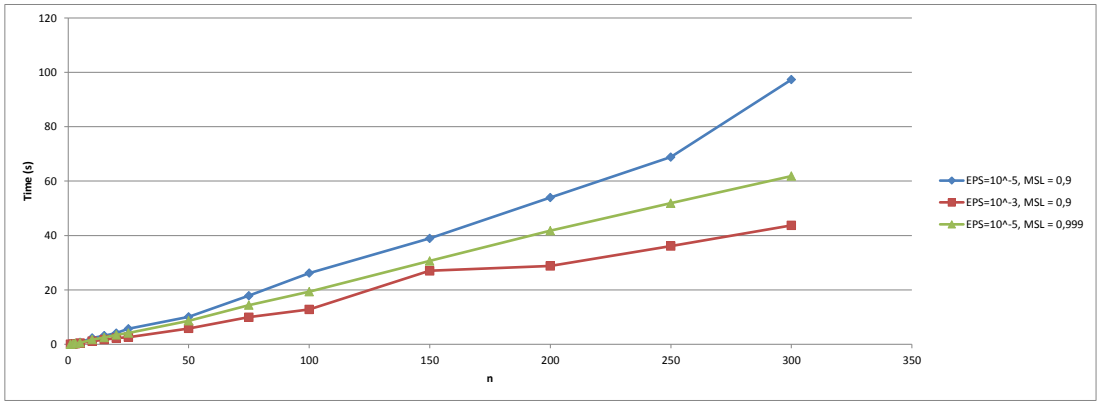
Moreover, especially when the derivative of a constraint contains 0, box consistency propagation is not able to make rapid changes to the variable domains. Thus, propagation is often repeatedly carried out, each time shrinking the domain quite insignificantly, perhaps even only by ϵ. Even though propagation is normally favored over branching, a new setting was introduced that, if a propagator was not able to shrink the domain of at least one variable to at least a fraction of its size (determined by modification significance level), it would notify the kernel that it is at fixpoint. This means that, if no other propagator is scheduled, the kernel would proceed with branching, rather than with more propagation, assuming that there are not many solutions, and that perhaps a whole branch may be dismissed in the next propagation round. Note that branching is still more computationally intensive (as well as memory intensive), so it should not be expected from a propagator to shrink the domain by too much. However, preventing very small shrinks and branching instead is beneficial for some instances.

Furthermore, due to reasoning with relatively high resolution, the difference between two possible values for a variable was quite small, so it often happened that a few adjacent values for a particular variable would satisfy all the constraints due to the tolerance. This yielded dozens of solutions that, in fact, all represented only a single solution within some small tolerance [2]

However, such solutions are the representations of the same actual solution, within floating point errors. Thus, a criterion was introduced that the solutions are only considered different solutions when the difference between any two solutions is at least 10ϵ for each variable.

[2]Not necessarily the ϵ tolerance, since any particular constraint is considered satisfied (for box consistency) if its left-hand side evaluates to 0 within the ϵ-environment. This means that the two solutios could differ by more. For example consider the constraint $\frac{1}{100}x - 1 = 0$. Here, two values for x could vary by as much as 100ϵ.

Figure 4.6: Broyden Tridiagonal - n vs Time to First Solution



Tables 4.2 - 4.4 show the results of the various instances solved with different ϵ and *modification significance level* (MSL) parameters. Times required to solve the instances are shown, as well as some statistics given by Gecode kernel. Initialization represents the time needed to initialize the constraints, without any propagation or branching work. Other columns (except for n and initialization time) have, in parentheses, whether the data therein corresponds to the first or the second solution.

The column *Time* gives the time required to solve the instance. Note that the solving time for the second instance includes the solving time for the first instance (and initialization), as solutions are found through a depth-first tree search, and second solution could not have been found without considering all the previous nodes. The column *Depth* gives the maximum depth of the search tree reached, while the column *Nodes* gives the number of nodes visited, and the column *Propagate* gives the number of times the *propagate* routine was called.

Times to first, and second solution are shown graphically in Figures 4.6 and 4.7.

Analysis of the Results

In all three cases, it can be noted that the time required to find the first solution rises with n. It is, in fact, almost linear to n. This is also the case with the depth (consequently, the number of nodes as well) and the number of propagations. This cannot be said for the second solution, which is sometimes much harder to find for smaller instances than for bigger ones (consider, for example instances with $n = 5$ and $n = 10$). In general, regularity does not follow from the experimental data for the second solution. Depth, number of visited nodes and number of propagations rise with n for the second solution as well, but required time does not follow the trend.

In the presented instances, lower resolution (higher ϵ) clearly outperforms the other two sets. This is expected, as lower resolution leaves a much smaller search space to be explored. Furthermore, it is interesting to note that, in both instances with $MSL = 0.9$, depth at which the solution is found is virtually identical, even though resolution differs significantly. Also, in both those cases, number of explored nodes when searching for the first solution is exactly greater

Table 4.2: Broyden Tridiagonal ($\epsilon = 10^{-5}$, $MSL = 0.9$) - Experimental Results

n	Initialization	Time (1)	Depth (1)	Nodes (1)	Propagate (1)	Time(2)	Depth (2)	Nodes (2)	Propagate (2)
1	0,004 s	0,058 s	1	2	27	0,061 s	3	8	48
2	0,005 s	0,157 s	2	3	68	34,438 s	6	14	120
3	0,008 s	0,335 s	4	5	112	89,852 s	6	16	322
5	0,014 s	0,714 s	9	10	237	185,709 s	10	39	847
10	0,029 s	2,354 s	18	19	691	4,402 s	21	777	4973
15	0,042 s	3,214 s	25	26	1039	3,887 s	32	23742	98395
20	0,055 s	4,129 s	35	36	1375	4,769 s	38	115685	483076
25	0,069 s	5,711 s	45	46	1724	?	?	?	?
50	0,152 s	10,124 s	97	98	3351	?	?	?	?
75	0,211 s	17,869 s	148	149	5668	?	?	?	?
100	0,288 s	26,196 s	197	198	7901	?	?	?	?
150	0,483 s	38,94 s	297	298	12451	?	?	?	?
200	0,591 s	53,959 s	397	398	17001	?	?	?	?
250	0,841 s	68,81 s	497	498	21551	?	?	?	?
300	0,947 s	97,325 s	597	598	26101	?	?	?	?

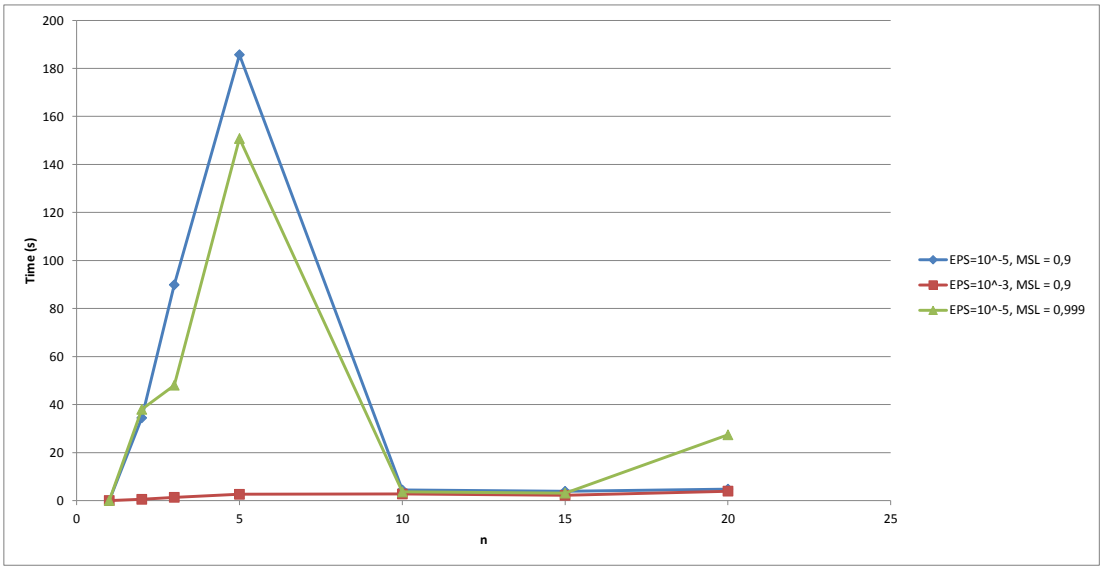
Table 4.3: Broyden Tridiagonal ($\epsilon = 10^{-3}$, $MSL = 0.9$) - Experimental Results

n	*SymbolicC++* Constraint Initialization	Time (1)	Depth (1)	Nodes (1)	Propagate (1)	Time(2)	Depth (2)	Nodes (2)	Propagate (2)
1	0,005	0,051	1	2	21	0,053	4	8	35
2	0,006	0,116	2	3	50	0,507	6	14	94
3	0,008	0,198	4	5	84	1,37	5	14	234
5	0,013	0,452	8	9	182	2,625	8	25	512
10	0,028	1,154	17	18	448	2,756	19	388	2392
15	0,044	1,743	28	29	715	2,196	30	12801	58463
20	0,061	2,295	37	38	892	3,925	40	263082	1140617
25	0,072	2,639	48	49	1076	?	?	?	?
50	0,147	5,855	97	98	2409	?	?	?	?
75	0,222	9,964	147	148	3875	?	?	?	?
100	0,278	12,85	197	198	5359	?	?	?	?
150	0,427	27,021	297	298	8309	?	?	?	?
200	0,602	28,817	397	398	11259	?	?	?	?
250	0,815	36,126	497	498	14209	?	?	?	?
300	0,894	43,738	597	598	17159	?	?	?	?

Table 4.4: Broyden Tridiagonal ($\epsilon = 10^{-5}$, $MSL = 0.999$) - Experimental Results

n	*SymbolicC++* Constraint Initialization	Time (1)	Depth (1)	Nodes (1)	Propagate (1)	Time(2)	Depth (2)	Nodes (2)	Propagate (2)
1	0,01	0,0601	1	2	30	0,0604	3	8	51
2	0,01	0,152	2	3	72	38,028	6	14	127
3	0,012	0,35	4	5	127	48,039	4	10	286
5	0,015	0,585	7	8	265	150,816	9	35	781
10	0,03	1,743	17	18	781	3,623	19	431	3925
15	0,042	2,51	26	28	1168	3,073	27	4967	22333
20	0,057	3,462	38	41	1640	27,4	42	346255	1377364
25	0,08	4,138	48	53	2015	?	?	?	?
50	0,149	8,645	108	126	4187	?	?	?	?
75	0,229	14,408	163	188	6876	?	?	?	?
100	0,31	19,333	227	267	9392	?	?	?	?
150	0,442	30,625	352	417	14578	?	?	?	?
200	0,613	41,768	477	567	19692	?	?	?	?
250	0,721	51,878	602	717	24878	?	?	?	?
300	0,848	61,866	727	867	29992	?	?	?	?

Figure 4.7: Broyden Tridiagonal - n vs Time to Second Solution



by 1 than the depth. This effectively means that the search was correctly directed towards the solution, without a single wrong turn on the way. This is, though, not the case with the higher *MSL*.

Comparing the two higher resolution sets with different *MSL*, one can observe that neither setting is consistenly outperforming the other. Lower *MSL* is better for some problem sizes, while greater *MSL* value is better for other sizes. It is also interesting that one value may be beneficial in finding the first solution, while other might be better for finding the second one.

Nevertheless, all three sets seem to have similar relative difficulty. If one instance takes more time to solve than some other on a particular setting, it will most likely also take more time to solve than the other one on some other setting. However, this is not the case for all instances.

4.3 Brown

Another benchmark, focusing on testing hull consistency, is *Brown*, again taken from the COPRIN examples [6]. The problem is defined by n equations as follows:

$$x_k + \sum_{i=1}^{n} x_i = n + 1, \quad 1 \leq k \leq n - 1 \tag{4.9}$$

$$\prod_{i=1}^{i=n} x_i = 1 \tag{4.10}$$

$$x_i \in [-10^8, 10^8], \quad 1 \leq k \leq n. \tag{4.11}$$

This problem, even though it has multiple occurences of a variable in every constraint (except for $\prod_{i=1}^{i=n} x_i = 1$), yields itself naturally to modeling through hull consistency.

To model $\prod_{i=1}^{i=n} x_i = 1$, one introduces a variable p representing the product of all variables, whose domain is just the number 1, and a hull consistency constraint $p = \prod_{i=1}^{i=n} x_i$.

Furthermore, to model the remaining set of constraints, one first introduces a variable $\sigma \in [-10^8 n, 10^8 n]$ that will hold the sum of all variables. Then, a constraint $\sigma = \sum_{i=1}^{n}$ is introduced. Now, for every remaining constraint, a variable $y_k \in [n-1, n-1]$ is introduced, and the constraint $x_k + \sum_{i=1}^{n} x_i = y_k$ posted.

Through this approach, very little excessive variables are introduced, and propagation through hull consistency can proceed.

Results

The table 4.5 shows the experimental results for a few instances of Brown. Considered instances have either two or three different solutions, depending on the instance. Measures desribed in Section 4.2 had to be taken to ensure only different solutions are given. Meaning of the columns is as in Section 4.2, with the parenthesized number representing whether the data is for the first, second or the third solution. Time is again cumulative, as, in order to find any solution, all of the previous ones (if they exist) have to be found first. Cells for the third solution are marked as *N/A* where no third solution exists. Note that in such cases it was proven no third solution exists (search space was exhausted), rather than the search being terminated due to exhaustion of resources. As hull consistency is used for all constraints, there is no (significant) constraint initialization time, as that represents the constraint initialization overhead introduced by *SymbolicC++*. Figure 4.8 shows the instance size against the time to find the first two solutions graphically. Third solution was omitted here as it does not exist for all instances.

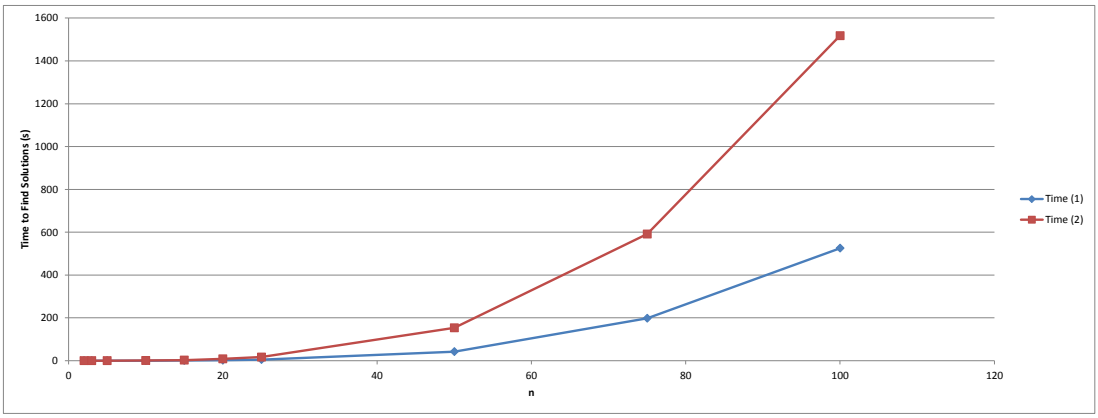
Analysis of the Results

It is interesting to note that the search proceeds to comparable depth in every of the considered instances except for $n = 2$, namely to depths between 100 and 109. Here, for all instances, finding the solutions for bigger instance requires more time than finding them for smaller instances (this is valid for both, or all three solutions). Increasing difficulty is also apparent through increasing number (with increasing n) of nodes and propagation calls, for all solutions. Second solution usually requires more time to be found after the first is found than what first one requires to be found in the first place. However, this is not always the case for the third solution, compared to the second.

Table 4.5: Brown - Experimental Results

n	Depth (1)	Nodes (1)	Propagate (1)	Time (1)	Depth (2)	Nodes (2)	Propagate (2)	Time (2)	Depth (3)	Nodes (3)	Propagate (3)	Time (3)
2	29	31	397	0,002 s	29	60	743	0,003 s	N/A	N/A	N/A	N/A
3	100	257	1507	0,008 s	103	641	3924	0,02 s	103	857	5413	0,027 s
5	101	613	4162	0,029 s	106	2290	15561	0,108 s	106	3342	23697	0,161 s
10	103	3212	34240	0,323 s	106	11131	120217	1,124 s	N/A	N/A	N/A	N/A
15	106	5245	77682	0,969 s	109	20382	311125	3,78 s	109	31827	490507	5,802 s
20	103	9136	173223	2,436 s	106	38118	727088	11,182 s	N/A	N/A	N/A	N/A
25	108	13747	316241	5,344 s	108	50921	1178599	22,123 s	108	77270	1783462	32,841 s
50	104	35125	1543369	42,221 s	109	152481	6683864	195,456 s	N/A	N/A	N/A	N/A
75	108	77327	5037387	197,898 s	109	288401	18734042	789,083 s	109	417030	26987053	1126,905s
100	108	122062	10537103	525,299 s	109	443701	38083240	2043,119 s	N/A	N/A	N/A	N/A

Figure 4.8: Brown - n vs Time to First and Second Solutions



CHAPTER 5

3D Reconstruction

One is often interested in a 3D reconstruction of some object. It is possible to obtain some information about the layout of points in space from a series of 2D images [9]. With this method, one can obtain a bounding box in which every of the points in the object lies, as well as some additional geometrical constraints. This gives a number of possible locations for different points, but only a few satisfy the geometrical constraints. Since the coordinates are real numbers, interval constraint programming is well-suited to solve this problem, and find a possible point locations satisfying the constraints.

5.1 Formal Statement of the Problem

For every point $P_i = (x_i, y_i, z_i)$, a bounding box $B_i = ([\underline{x_i}, \overline{x_i}], [\underline{y_i}, \overline{y_i}], [\underline{z_i}, \overline{z_i}])$ is given. Note that here P_i is unknown - the values of its components have to be found so that for every point i, $P_i \in B_i$.

Furthermore, set of faces given as a set of triples representing point indices is given. This set represents faces, where a face is a triangle. Three points form a face if and only if their indices are in the face set. Let a face $F_i = (v_1, v_2, v_3)$, where v_1, v_2, v_3 are point indices, dependent on i.

Moreover, some geometric constraints are given. These are divided into coplanarity constraints, orthogonality constraints, parallelism constraints and equal angle constraints.

A coplanarity constraint is given by four point indices, and represents the fact that those four points lie on a same plane. Each coplanarity constraint can be denoted as $C_i^{copl} = (v_1, v_2, v_3, v_4)$.

Orthogonality is a relation between faces, and each orthogonality constraint gives two face indices, and means that the faces given by those indices are orthogonal. It can be denoted as $C_i^{orth} = (f_1, f_2)$.

Parallelism constraints are given analogously as orthogonality, but represent parallel faces. It can be represented as $C_i^{par} = (f_1, f_2)$.

constraints are given as a quarduple - a pair of pairs of face indices. The meaning is that the faces pointed to by the first two indices enclose equal angle as the faces pointed to by the second two indices. It can be represented as $C_i^{ea} = ((f_1, f_2), (f_3, f_4))$.

5.2 Variable and Constraint Definition

Variables

For every point P_i, three variables corresponding to its coordinates are defined. Their initial domains correspond to the bounding boxes. This creates three arrays of variables in form x_i, y_i and z_i. Since the domains are initially limited by the bounding boxes, bounding boxes are not used afterwards.

For every face F_i, a normal must be calculated. Thus, four variable arrays are introduced, namely, nx_i, ny_i, nz_i and il_i, where the first three represent the normal vector coordinates, and the last one represents its inverse length. Since no conclusion can be made about the possible values of normals in advance, their domains should be set to some large range, possibly even $(-\infty, \infty)$, so that no normal values would be dismissed because of too restrictive domain. Inverse length cannot take negative values, so its domain can take that into account.

Normal Definition - Length

Constraints defining the normals need to be laid out. For every face F_i, its length needs to be exactly 1. Through the Pythagorean theorem, this can be stated as

$$il_i^2(nx_i^2 + ny_i^2 + nz_i^2) = 1. \tag{5.1}$$

This, in effect, sets up the relation between the normal components and its inverse length - which will be necessary to reason about angles afterwards.

Normal Definition - Components

A normal (with an arbitrary length, not necessarily equal to 1), to a face $F_i = (\vec{v_1}, \vec{v_2}, \vec{v_3}) = ((x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3))$ is given as

$$\vec{n_i} = (\vec{v_2} - \vec{v_1}) \times (\vec{v_3}, \vec{v_1}).$$

The constraints for nx_i, ny_i and nz_i are laid out by decomposing the above cross product component-wise.

Coplanarity Constraints

A plane through three points, given as $(\vec{v_1}, \vec{v_2}, \vec{v_3}) = ((x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3))$ is defined through the determinant

$$\begin{vmatrix} x - x_1 & y - y_1 & z - z_1 \\ x - x_2 & y - y_2 & z - z_2 \\ x - x_3 & y - y_3 & z - z_3 \end{vmatrix}$$

with free variables x, y and z. For a coplanarity constraint

$$C_i^{copl} = (\vec{v_1}, \vec{v_2}, \vec{v_3}, \vec{v_4}) = ((x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3), (x_4, y_4, z_4)), \tag{5.2}$$

by plugging the corresponding variables in the above determinant, and plugging (x_4, y_4, z_4) in place of the variables x, y and z, a arithmetic expression representing the corresponding coplanarity constraint is obtained. The final constraint can then be expressed as

$$\begin{vmatrix} x_4 - x_1 & y_4 - y_1 & z_4 - z_1 \\ x_4 - x_2 & y_4 - y_2 & z_4 - z_2 \\ x_4 - x_3 & y_4 - y_3 & z_4 - z_3 \end{vmatrix}$$

.

Orthogonality Constraints

Two planes (or faces) are orthogonal if their normals are orthogonal. This will be the case if the dot product of the normals is 0. Given a orthogonality constraint $C_i^{orth} = (f_j, f_k)$, we have that

$$\vec{n_j} \cdot \vec{n_k} = 0 \tag{5.3}$$

where $\vec{n_j}$ and $\vec{n_k}$ are the normal vectors corresponding to the faces f_j and f_k. This can be expanded to

$$nx_j nx_k + ny_j ny_k + nz_j nz_k = 0 \tag{5.4}$$

where variables correspond exactly to the variables in the system.

Parallelism Constraints

Two planes (or faces) are parallel if their normals are parallel. This will be the case if the dot product of the unit normals is 1. Given a orthogonality constraint $C_i^{par} = (f_j, f_k)$, we have that

$$\frac{\vec{n_j} \cdot \vec{n_k}}{|\vec{n_j}||\vec{n_k}|} = 1 \tag{5.5}$$

where $\vec{n_j}$ and $\vec{n_k}$ are the normal vectors corresponding to the faces f_j and f_k. This can be expanded to

$$(il_j\ il_k)(nx_j nx_k + ny_j ny_k + nz_j nz_k) = 1 \tag{5.6}$$

where variables correspond exactly to the variables in the system, and the inverse length factor $(il_j\ il_k)$ transforms the normals into unit normals.

Equal Angle Constraints

Two pairs of planes (or faces) enclose equal angles if the dot products of the pairs are equal. Given a equal angle constraint $C_i^{ea} = ((f_j, f_k), (f_l, f_m))$, we have that

$$\frac{\vec{n_j} \cdot \vec{n_k}}{|\vec{n_j}||\vec{n_k}|} = \frac{\vec{n_l} \cdot \vec{n_m}}{|\vec{n_l}||\vec{n_m}|} \tag{5.7}$$

where $\vec{n_j}$, $\vec{n_k}$, $\vec{n_l}$ and $\vec{n_m}$ are the normal vectors corresponding to the faces f_j, f_k, f_l and f_m. This can be expanded to

$$(il_j\ il_k)(nx_jnx_k + ny_jny_k + nz_jnz_k) = (il_l\ il_m)(nx_lnx_m + ny_lny_m + nz_lnz_m) \tag{5.8}$$

where variables correspond exactly to the variables in the system, and the inverse length factors $(il_j\ il_k)$ and $(il_l\ il_m)$ transform the normals into unit normals.

5.3 Experiments

Instances

The experiments with 3D reconstruction were carried out on five different regular instances, representing various objects. Furthermore, two different synthetic instances were constructed from the two different regular polyhedra with triagonal faces, namely, tetrahedron and octahedron. The synthetic instances were based on the polyhedron data as given by *Wolfram Mathematica 8* [21]. Bounding boxes were constructed in such a way that the midpoint of each bounding box corresponds to the coordinates given by *Mathematica* for the corresponding vertex. The sizes of the bounding boxes were made comparable to the regular instances. All objects were described through the bounding boxes of their vertices, and their faces, as well as constraints as described in Section 5.1. This data was loaded from a plaintext file for each of the instances.

Model

For all experiments, every normal length constraint was decomposed into simpler constraints, so that hull consistency enforcement could be applied to the new constraints. Reason for this is that, if box consistency were applied instead, in the Newton step a derivative containing 0 would be calculated, which would result in much slower convergence, as the implementation would have to resort to the binary narrowing procedure, as described in Section 3.2. Thus, every constraint of the form

$$il_i^2(nx_i^2 + ny_i^2 + nz_i^2) = 1 \tag{5.9}$$

was replaced with constraints in the form

$$ilsq_i = il_i^2 \tag{5.10}$$
$$nxsq_i = nx_i^2 \tag{5.11}$$
$$nysq_i = ny_i^2 \tag{5.12}$$
$$nzsq_i = nz_i^2 \tag{5.13}$$
$$lensq_i = nxsq_i + nysq_i + nzsq_i \tag{5.14}$$
$$lensq_i\ ilsq_i = 1 \tag{5.15}$$

with the appropriate variables introduced.

Table 5.1: Computation Time for Various 3D Reconstruction Instances

Instance	Number of Vertices	Number of Faces	Initialization Time	Solve Time	Total Time
tetrahedron	4	4	0.671 s	0.057 s	0.728 s
octahedron	6	8	2.368 s	0.124 s	2.492 s
casetta	9	14	7.237 s	0.244 s	7.481 s
boxwhole	16	32	19.423 s	0.677 s	20.1 s
lego	26	42	32.491 s	1.11 s	33.601 s
test	32	60	33.778 s	2.032 s	35.81 s
pozzo	49	92	51.463 s	2.712 s	54.175 s

All other constraints were declared in the model exactly as specified, and were posted with box consistency. Since all are polynomials, with the degree of all variables being at most 1, partial derivative on any variable is normally constant everywhere. This has as a consequence very fast convergence in a Newton step.

Results

Table 5.1 shows the initialization and solve time for each instance, as well as the size of the instance (through both the number of vertices and edges). Here, *initialization time* represents the time it takes to load the instance, initialize variables and construct the *Symbolic* objects used to represent constraint. The last factor is by far the most significant (at least by an order of magnitude). *Solve time* is, on the other hand, the time the implementation needs after the initialization to obtain the first solution, which is the solution shown in the result. *Total time* is the sum of the initialization and solve times.

Figures 5.1 and 5.2 show the reconstructions of the two synthetic instances, namely, tetrahedron, and octahedron. Furthermore, Figures 5.3 - 5.7 show the 3D reconstructions from various angles of the five different test instances. These representations are only one of many possible solutions, since, in those instances, every vertex has its bounding box. Thus, intuitively, one may be virtually free to choose the position for the first vertex.

All of the 3D reconstruction images were constructed by *Wolfram Mathematica 8* [21], from the result coordinates produced by the implemented Gecode extension.

Analysis of the Results

The shown instances were solved by this implementation, in a reasonable amount of time, as all of the shown instances were solved in under a minute. However, as well as the results shown in Section 4.1, results in this section emphasize disproportionally long initialization time, which is more than an order of magnitude greater than the time it takes to find the actual solution. In a comparable implementation using another computer algebra system, initialization times were two orders of magnitude faster. This further serves to highlight that *SymbolicC++* may not be the most appropriate library for interval constraint programming, especially when many constraints are necessary. However, it is interesting to note that, compared to creating *Symbolic*

Figure 5.1: 3D reconstruction of the synthetic instance *tetrahedron*

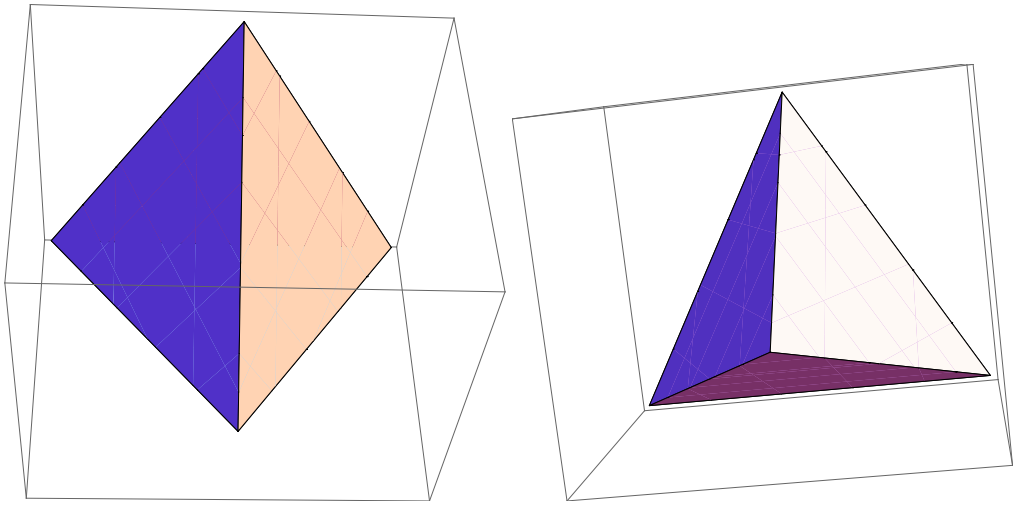


Figure 5.2: 3D reconstruction of the synthetic instance *octahedron*

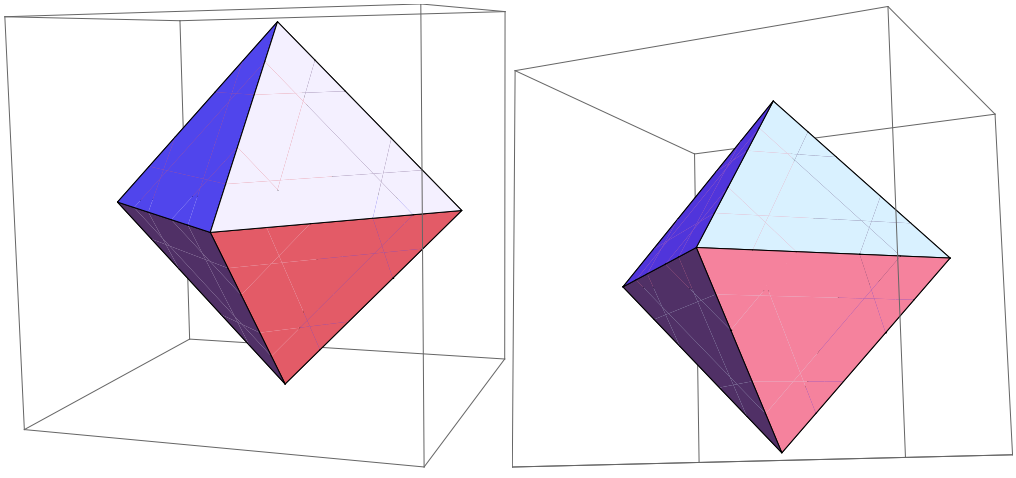


Figure 5.3: 3D reconstruction of the instance *boxwhole*

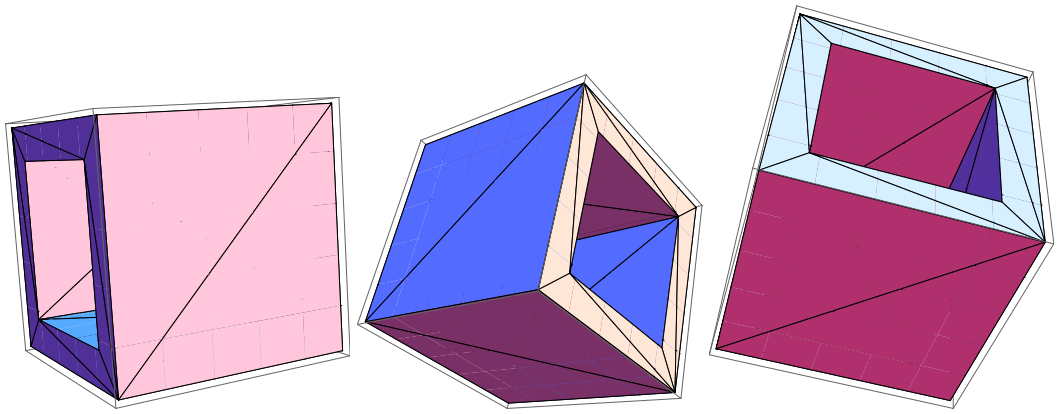
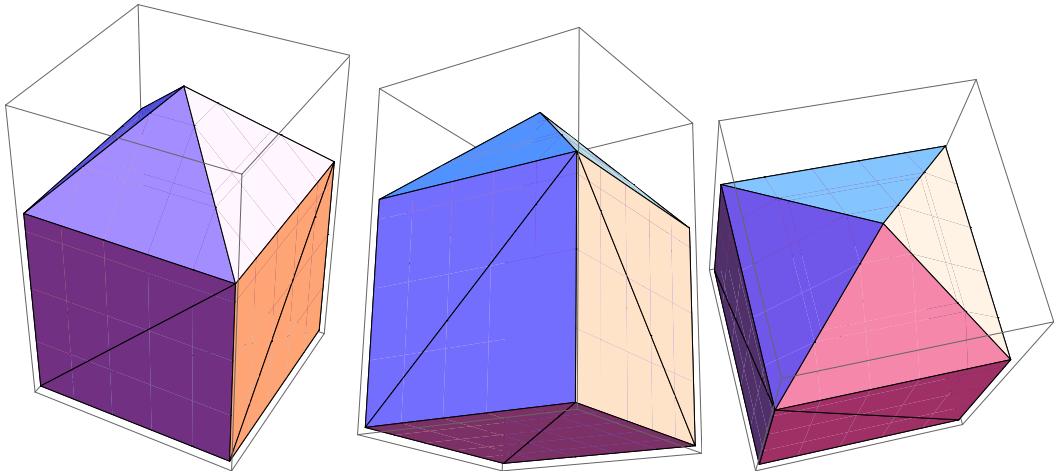


Figure 5.4: 3D reconstruction of the instance *casetta*



objects, working with them is not as slow, which is shown by the fact that the time to solve every of those instances is much less than the time it takes just to create the *Symbolic* objects corresponding to the constraints.

Figure 5.5: 3D reconstruction of the instance *lego*

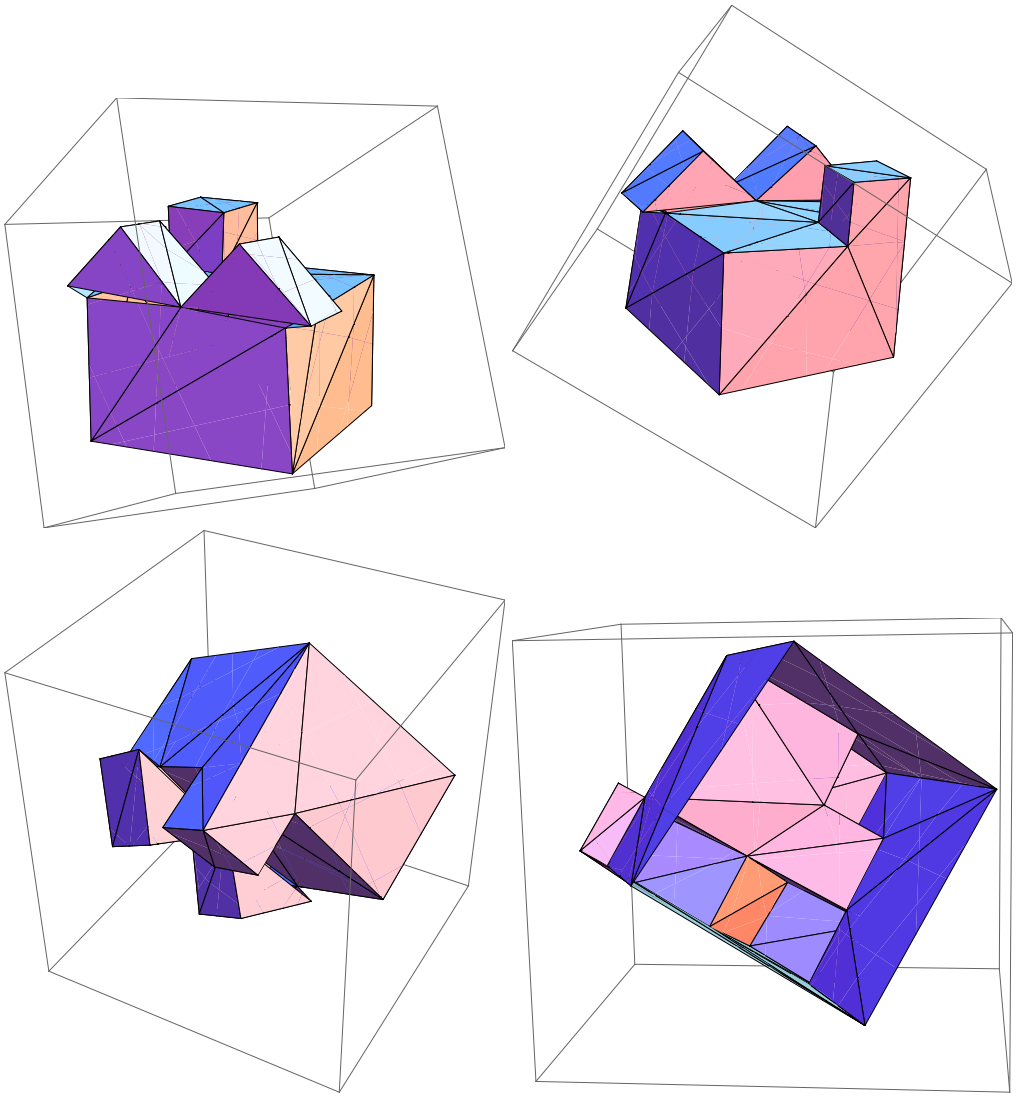


Figure 5.6: 3D reconstruction of the instance *test*

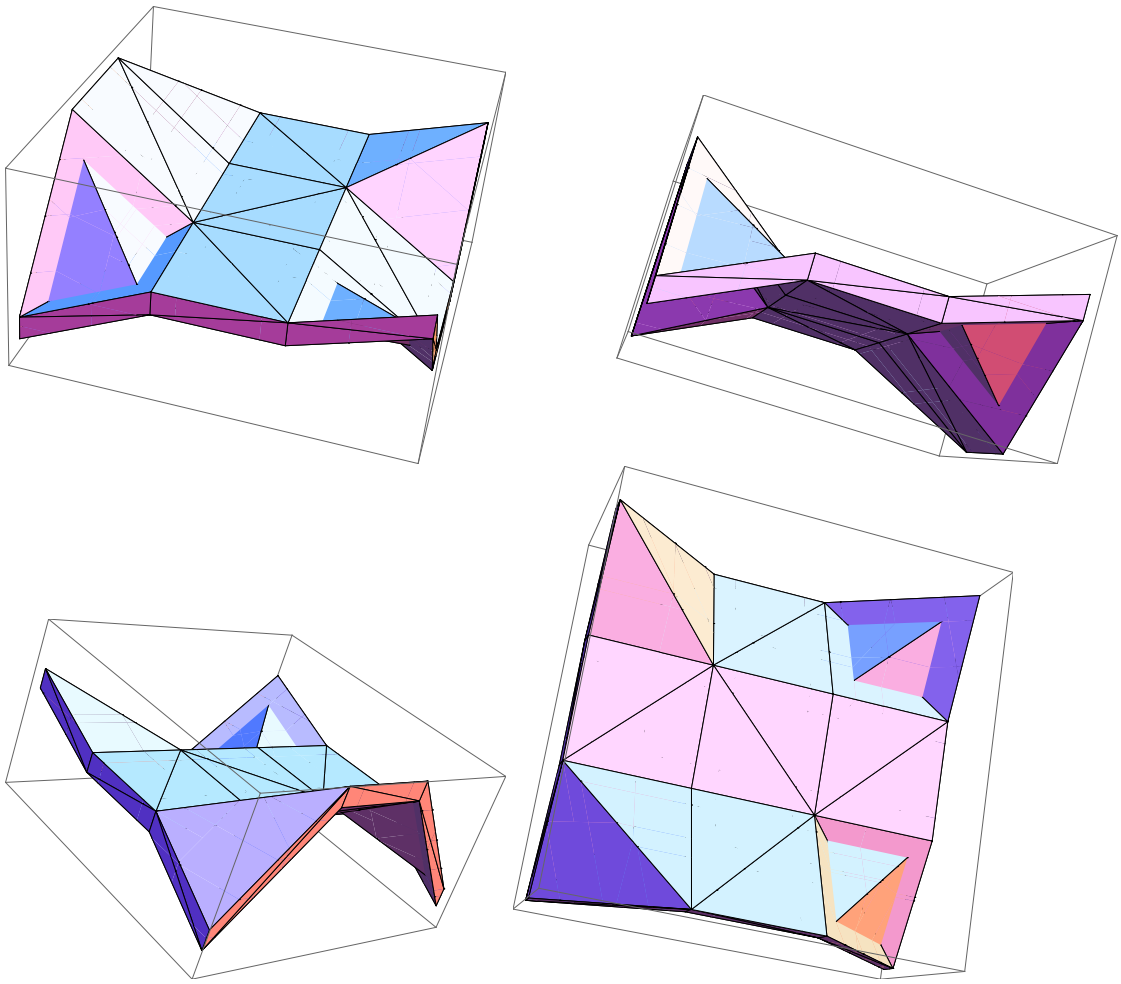
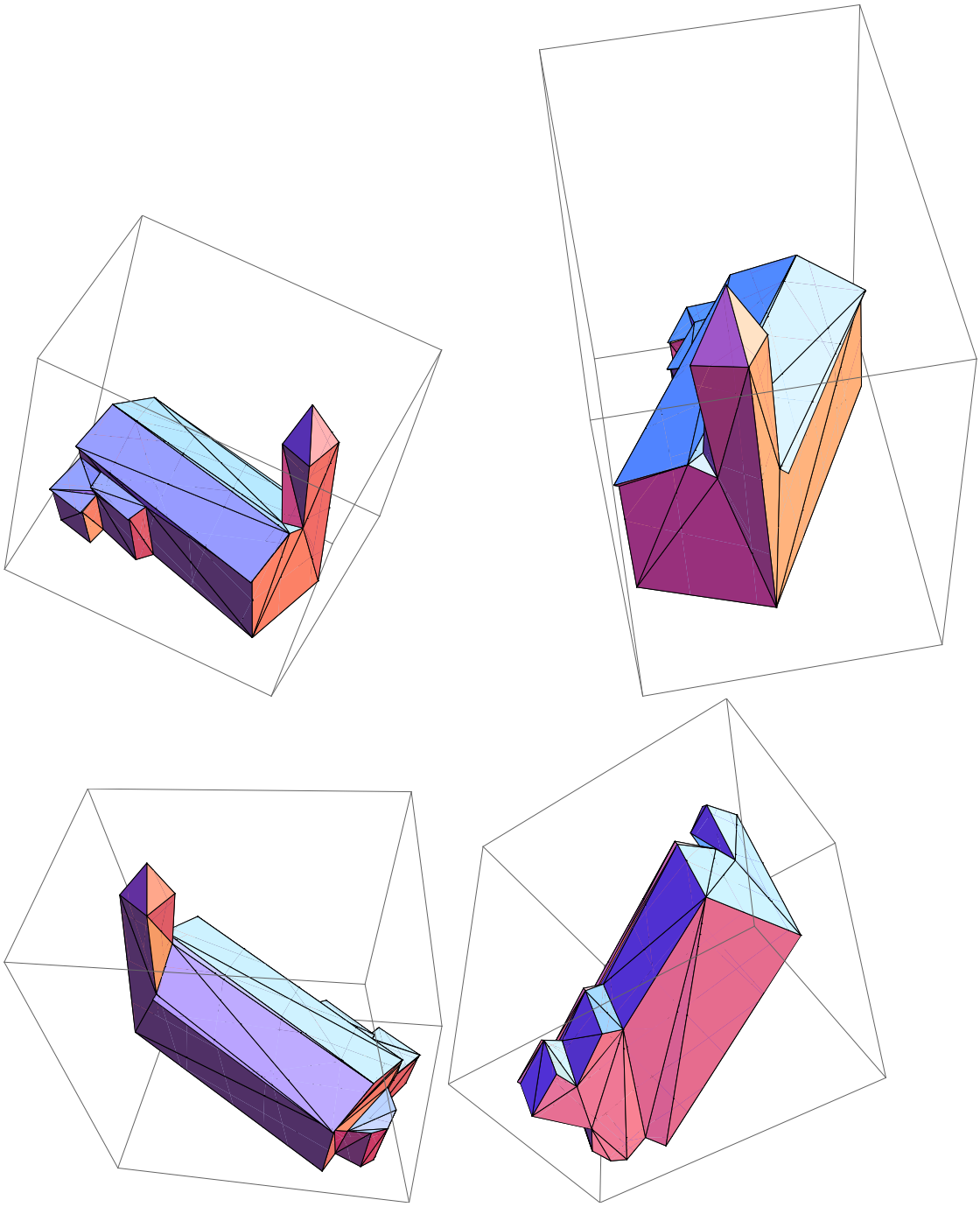


Figure 5.7: 3D reconstruction of the instance *pozzo*



CHAPTER 6

Conclusions and Future Work

The underlying concepts for Interval Constraint Programming were presented, and an extension of Gecode capable of such interval reasoning was implemented, and tested on various scalable benchmarks, as well as the 3D reconstruction problem. The implemented extension supports two consistency notions, namely, hull and box consistency.

In the implemented system, hull consistency propagators were implemented for the most common arithmetic operations. Hull consistency is the direct approximation of the arc consistency in the interval domain. The algorithm used for propagation of hull consistency constraints was HC4. Despite using the HC4 algorithm instead of the simpler HC3, hull consistency constraints are limited to a particular, relatively simple form, having only a single operation per constraint. Reason for this limitation is that, even though it may be possible to use HC4 to propagate more complex constraints, a particular procedure would have to be written for forward evaluation and backward propagation for each of the more complex forms that need to be supported. Nevertheless, user is able to add custom propagators to handle such cases, should such functionality be required.

The advantage of using hull consistency is that it is more efficient than box consistency, as long as every variable appears only once in every constraint. When this is not the case, hull consistency can still be applied, but box consistency may have performance advantages. The disadvantage of using hull consistency is, as was already stated, possibly lower performance when variables occur multiple times in a constraint. Furthermore, since only particular form of constraints can be specified in the implemented system through hull consistency, some constraint decomposition (although not to the extent algorithm HC3 requires) is necessary to use hull consistency if the user is unwilling to program own hull consistency propagators.

Aside from hull consistency propagators, a single box consistency propagator was implemented. Box consistency constraints have a function implementing a constraint that is 0 exactly for the values for which the constraint holds, and is nonzero otherwise. Box consistency is a relaxation of hull consistency, in which, instead of reasoning about the necessary domains of variables in an expression, one aims to eliminate the outermost values in an interval by showing that they are inconsistent, that is, that the function implementing the constraint is not 0 for such

values. To eliminate inconsistent values for a particular variable, the interval Newton method is used, if the derivative of the function implementing the constraint is nonzero when evaluated for the domain of that variable. If the derivative contains zero, workaround method have to be used. The binary narrowing procedure, described in Section 3.2, was used for this purpose in the implementation.

To implement box consistency, a method to represent symbolic expressions, and to differentiate them was required. The library *SymbolicC++* was chosen, and extended for this purpose. With the help of the library, the system was able to effectively propagate box consistency constraints. However, it presented its weaknesses in constructing the *Symbolic* objects, which represent the constraints internally. Such initialization time is unusually long, and this is particularly emphasized on problems like 3D reconstruction with a large number of constraints (see Section 5.3), or on problems like Broyden Banded that require extensive box consistency propagation (see Section 4.1). After a *Symbolic* object is created, the library performs its tasks comparatively more efficiently, but a different library or design pattern such as expression templates may still be a better choice. This all leads to a conclusion that this library may have been a suboptimal choice for this project.

Both of those consistency types were used to solve various scalable problems taken from the COPRIN example database [6], as well as the 3D reconstruction problems. This shows that the 3D reconstruction is feasible to solve with this method, and one can assume that even larger instances can be solved through this method too, as the time it takes to solve considered one is relatively small (excluding initialization time).

In conclusion, a working implementation of interval constraint programming for Gecode was implemented. Its weakness is the usage of the *SymbolicC++* library, which performs somewhat badly when initializing constraints. When working with the expressions, its performance is not as unsatisfactory, but other libraries or design patterns can offer better performance.

Furthermore, methods of handling the case when derivative contains zero, and a Newton step cannot be made could be improved. Instead of attempting to propagate through the binary narrowing procedure, an implementation could perhaps branch, thus creating more spaces to explore, but hoping it will be able to propagate with a Newton step in the newly created spaces. Here, Gecode is somewhat rigid by specifying that all propagation has to be done before a branching, because branching is usually more computationally heavy than propagation because it doubles the number of spaces that have to be explored. However, in some cases, such as in interval constraint programming, when a derivative contains zero, a more carefully controlled blend between propagation and branching may improve the performance of the implementation. Alternatively, an implementation could try to skip box consistency propagation on a certain variable when it is unable to make a Newton step, under some carefully devised conditions.

Bibliography

[1] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.

[2] Krzysztof R. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

[3] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-Francois Puget. Revising hull and box consistency. In *In Proceedings of the 16th International Conference on Logic Programming, Las Cruces, New Mexico (USA)*, pages 230–244, 1999.

[4] Frédéric Benhamou and Laurent Granvilliers. Continuous and Interval Constraints. In P. van Beek F. Rossi and T. Walsh, editors, *Handbook of Constraint Programming*, pages 569–601. Elsevier, 2006.

[5] Christian Schulte and Guido Tack and Mikael Z. Lagerkvist. Gecode website. `http://www.gecode.org`. Accessed: August 21, 2012.

[6] *COPRIN* Team. *COPRIN examples*. `http://www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html`. Accessed: August 7, 2012.

[7] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of ACM*, 7(3):201–215, 1960.

[8] ECLiPSe Team. *The ECLiPSe Constraint Programming System Library*. `http://www.eclipseclp.org/`. Accessed: August 2, 2012.

[9] Michela Farenzena and Andrea Fusiello. Stabilizing 3d modeling with geometric constraints propagation. *Computer Vision and Image Understanding*, 113(11):1147–1157, 2009.

[10] Alexandre Goldsztejn and Frédéric Goualard. Box consistency through adaptive shaving. In *In Proceedings of the 25th Symposium on Applied Computing, Sierre (Switzerland)*, pages 2049–2054, 2010.

[11] Guillaume Melquiond and Sylvain Pion and Hervé Brönnimann and Jens Maurer and Jeremy Siek and Maarten Keijzer. *Boost Interval Library*. `http://www.boost.org/doc/libs/1510/libs/numeric/interval/doc/interval.htm`. Accessed: August 5, 2012.

[12] Eldon Hansen and G. William Walster. *Global Optimization using Interval Analysis*. Marcel Dekker, second edition, 2004.

[13] IBM. *IBM ILOG CPLEX CP Optimizer*. `http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/`. Accessed: August 2, 2012.

[14] J.P.E. Hodgson. *ISO conformant version of the Prolog language*. `http://www.deransart.fr//prolog/docs.html`. Accessed: August 2, 2012.

[15] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[16] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[17] Pascal Van Hentenryck. Comet™. `http://dynadec.com/technology/`. Accessed: August 2, 2012.

[18] Francesca Rossi, Peter van Beek, and Toby Walsh (Editors). *Handbook of Constraint Programming*. Elsevier, first edition, 2006.

[19] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. *Modeling and Programming with Gecode, corresponding to Gecode 3.7.3*, 2012.

[20] Pascal Van Hentenryck, David McAllester, and Deepak Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34(2):797–827, April 1997.

[21] Wolfram Research. *Wolfram Mathematica*. `http://www.wolfram.com/mathematica/`. Accessed: August 2, 2012.

[22] Yorick Hardy and Willi-Hans Steeb and Tan Kiat Shi. *SymbolicC++* library. `http://issc.uj.ac.za/symbolic/symbolic.html`. Accessed: August 5, 2012.