Applications of Nested Search*

Peter Zoeteweij

CWI, P.O. Box 94079, 1090 GB Amsterdam, the Netherlands
p.zoeteweij@cwi.nl

Abstract. We show that a number of techniques for improving the efficiency of constraint solving can be expressed as applications of a single operator for nested search. We discuss the implementation of this operator, and provide experimental results to demonstrate that it yields a viable implementation of the techniques that we are interested in.

Keywords: Constraint propagation, nested search.

1 Introduction

Constraint propagation is usually implemented as the repeated application of a number of reduction operators, that enforce some form of local consistency, such as arc consistency, or an approximation thereof. In this context ‘local’ means that only individual constraints are considered when removing values from the domains of variables. For arithmetic constraints, these individual constraints can be the result of a decomposition of complex constraints into atomic constraints, for which the resulting form of consistency is weaker than for the original constraints.

In some cases the efficiency of constraint solving can be improved by using reduction operators that enforce a less local form of consistency. Such operators typically still update the domain of a single variable, but more than a single constraint is considered when removing values. Here we will show that two such operators, namely enforcing box consistency for arithmetic constraints, and shaving for scheduling problems, can be expressed as applications of a single operator for nested search. With nested search we mean that the computation performed by the reduction operator involves a search process. In addition we show that this operator has a third application: optimization by means of a bisection search on the domain for the outcome of an objective function.

An advantage of using the same operator for these different techniques is that we don’t need to implement dedicated operators for each of them. Instead it supports the composition of complex operators from a limited set of basic facilities. This is important in a hybrid solver, where we want to avoid fixing techniques to specific domain types and application areas. A disadvantage is that a generic operator is likely not as efficient as a dedicated implementation. We provide the

* This work is supported by NWO, the Netherlands Organization for Scientific Research, under project number 612.069.003.
results of experiments that show that even though we use almost a full constraint solver for the nested search, we still obtain a viable implementation.

The paper is structured as follows. The operator for nested search is defined in Sect. 2. In Sect. 3 optimization, box consistency and shaving are defined using this operator. In Sect. 4 the implementation of the operator is described, and Sect. 5 details the experiments that were performed to test the optimization and box consistency applications. Section 6 concludes the paper.

2 An Operator for Nested Search

2.1 Preliminaries

We extend here the standard concept of a constraint satisfaction problem (CSP, see e.g. [2]) with the notion of an auxiliary variable and of a solved form.

Consider a sequence of variables \( X := (x_1, \ldots, x_n) \) where \( n \geq 0 \), with respective domains \( (D_1, \ldots, D_n) \) associated with them. Let \( D \) denote the latter sequence. Below we will omit the brackets in the notation for sequences when this does not lead to confusion. Let \( \text{elem}(X) \) denote the set \( \{x_1, \ldots, x_n\} \) of elements of \( X \). By a constraint \( C \) on \( X \) we mean a subset of \( D_1 \times \ldots \times D_n \).

Given a subsequence \( Y := x_{i_1}, \ldots, x_{i_l} \) of \( X \), \( D[Y] \) denotes the sequence of domains \( D_{i_1}, \ldots, D_{i_l} \) associated with the variables of \( Y \), and given an element \( d := d_1, \ldots, d_n \) of \( D_1 \times \ldots \times D_n \), \( d[Y] \) denotes the sequence \( d_{i_1}, \ldots, d_{i_l} \).

A CSP consists of a finite sequence of variables \( X := x_1, \ldots, x_n \) with respective domains \( D := D_1, \ldots, D_n \), together with a finite set \( C \) of constraints, each on a subsequence of \( X \). We add to this a set \( A \subseteq \text{elem}(X) \), of auxiliary variables. Variables that are not in \( A \) are called decision variables. A CSP is now written as

\[
\langle C ; x \in D_1, \ldots, x_n \in D_n ; A \rangle.
\]

For empty \( A \), a CSP of the form (1) is written in the usual notation \( \langle C ; x \in D_1, \ldots, x_n \in D_n \rangle \), and for explicitly enumerated sets \( C \), a sequence notation is used.

By a solution to a CSP of the form (1) we mean an element of \( D_1 \times \ldots \times D_n \) such that for each constraint \( C \in C \) on a sequence of variables \( Y \) we have \( d[Y] \in C \). We call a CSP consistent if it has a solution and inconsistent otherwise. Two CSPs with the same sequence of variables are called equivalent if they have the same set of solutions. If for all \( 1 \leq i \leq n \) we have \( x_i \in A \) or \( |D_i| = 1 \), then a CSP of the form (1) that is consistent is said to be in solved form. Under the same condition, a CSP of the form (1) that is \( B \) consistent is said to be in \( B \) solved form, where \( B \) refers to some consistency notion, such as \( B = \text{arc} \) for arc consistency.

A CSP of the form (1) is handled by branch-and-propagate constraint solving, but branching never takes place on auxiliary variables. Instead of computing solutions to a CSP of the form (1), we are computing solved forms. In practice, determining that a solved form is consistent is as difficult as computing a full solution, which would still require search on the auxiliary variables. Instead we
stop the search when all decision variables have been given a value. On the resulting CSP we perform one last round of constraint propagation, to reach a form of local consistency that suits our purposes. As we shall see in Sect. 3, this last round of propagation is used also to communicate properties of the solved form back to the global CSP on $X$, that the solves constraint is part of.

Example 1. The CSP

$$\langle w \neq x, x \neq y, x \neq z, y \neq z ; w = 2, x \in \{0,1\}, y \in \{0,1\}, z \in \{0,1\} ; \{x,y,z\} \rangle$$

is not in solved form because it is inconsistent, but it is in arc solved form because it is arc consistent, while the domain of the only decision variable $w$ is a singleton set. The CSP

$$\langle x < y, y \neq z ; x = 0, y \in \{0,1,2\}, z = 0 ; \{y\} \rangle$$

is in solved form because it is consistent ($x, y, z = 0,1,0$ and $x,y,z = 0,2,0$ are solutions) and the domains of all decision variables are singleton sets. The CSP

$$\langle x < y, y \neq z ; x = 0, y \in \{1,2\}, z = 0 ; \{y\} \rangle$$

is in arc solved form because in addition to the properties of the previous example, it is also arc consistent. □

Some further notation: given two integers $l$ and $h$, let $[l..h]$ denote the set $\{i \in \mathbb{N} \mid l \leq i \leq h\}$, and let $[-\infty..h]$ denote the set $\{i \in \mathbb{N} \mid i \leq h\}$. Given two sequences $A$ and $B$, let $A, B$ denote their concatenation. If $A := a_1, \ldots, a_k$ and $B := b_1, \ldots, b_l$ are of the same length $l = k$, then for a relation $r$ between the elements of $A$ and $B$, we use $A r B$ as a shorthand for $a_1 r b_1, \ldots, a_k r b_k$. If $l = 1$, then we use $A r B$ as a shorthand for $a_1 r b_1, \ldots, a_k r b_1$.

2.2 Nested Search

Given a sequence of variables $X := x_1, \ldots, x_n$ with respective domains $D := D_1, \ldots, D_n$ associated with them, we introduce the constraint

$$\text{solves}(X_I, X_O, Y, D_Y, C_{X_I,Y}, A_{X_I,Y}).$$

In addition to the subsequence $X_I$ of $X$ that the constraint applies to, The constraint is parameterized by

- a subsequence $X_O$ of $X_I$, the output variables of the constraint. $X_I$ are called the input parameters of the constraint,
- a finite sequence $Y$ of variables that do not occur in $X$, and a sequence $D_Y$ of the same length $n_Y \geq 0$ of domains associated with them,
- a set $C_{X_I,Y}$ of constraints, each on a subsequence of $X_I, Y$, and
- a set $A_{X_I,Y} \subseteq \text{elem}(X_I,Y)$.
A reduction operator that enforces the constraint (3) works on the local CSP
\[ \langle C_{X,Y} ; X_I, Y \in D[X_I], D_Y ; A_{X_I,Y} \rangle. \]  

Let \( B \) refer to some notion of consistency, and suppose that there exists a CSP in \( B \) solved form
\[ \langle C_{X,Y} ; X_I, Y \in D'[X_I], D_Y' ; A_{X_I,Y} \rangle \]

such that \( D'[X_I], D_Y' \subseteq D[X_I], D_Y \). Now enforcing the constraint (3) can be described in the framework of [1] by the following reduction rule:
\[ \langle \text{solves}(X_I, X_O, Y, D_Y, C_{X,Y}, A_{X,Y}) ; x_1 \in D'_1, \ldots, x_n \in D'_n \rangle \]

where for \( x_i \notin \text{elem}(X_O) \) we have \( D'_i = D_i \), and for \( x_i \in \text{elem}(X_O) \), if the solved form (5) exists, we have \( D'_i = D''_i \). If the solved form (5) does not exist, then there exists an element \( x_i \in \text{elem}(X_O) \) for which \( D''_i = \emptyset \).

The reading of constraint (3) is “(5) solves (4).” We illustrate its usage by means of the following example:

**Example 2.** Consider the CSP
\[ \langle \text{solves}((x, y), (y), (z), \{(0, 1, 2\}), \{x < y, y \neq z\}, \{y\}) ; x, y \in \{0, 1, 2\} \rangle. \]

When the reduction rule (6) is applied to this CSP, an implementation of this rule will construct the CSP \( \langle x < y, y \neq z ; x, y, z \in \{0, 1, 2\} \rangle \). Suppose that, by means of a branch-and-propagate search on \( x \) and \( z \), the implementation of this operator finds the arc solved form (2) of example 1. The sequence of output variables of the solves constraint in this example consists of only the variable \( y \), so the result of applying the reduction rule is that the domain of \( y \) is reduced from \( \{0, 1, 2\} \) to \( \{1, 2\} \).

A reduction operator that implements the reduction rule (6) is awkward because it is not necessarily equivalence preserving, and because its functionality depends on the particular instance of the solved form (5) that is found. Still, as we shall see in the next two sections, it can be used to describe some powerful transformations of CSPs, and the implementation is straightforward.

### 3 Applications

#### 3.1 Optimization

As an alternative to branch-and-bound, in optimization we can perform a bisection search on the range of an objective function. In [4] this is used for scheduling problems. The idea is to split the range for the outcome of the objective function into two halves. Suppose we want to minimize the value for the objective function. First we search for a solution in the lower half. If it exists, we adjust...
the upper bound of the range, and continue the search in the resulting range. If it does not exist, we continue the search in the upper half of the split.

This scheme can be implemented using the operator of Sect. 2.2 as follows. Suppose that the CSP \( \langle C \mid X \in D \rangle \) defines the combinatorial problem, and that we want to minimize an (integer) objective function \( f(X) \) on the variables of \( X \), where \( l \) and \( h \) are obvious lower and upper bounds of \( f(X) \) for given \( D \). Now a bisection branching, combined with a depth-first leftmost-first first-solution search on the following CSP implements the optimization algorithm:

\[
\langle \text{solves}(X_I, X_O, Y, D_Y, C_{X_I}, Y, A_{X_I}, Y) ; \ x_o \in [l..h] \rangle,
\]

where

- \( X_I := x_o \), \( X_O := x_o \),
- \( Y := X, x'_o \), \( D_Y := D, ZZ \),
- \( C_{X_I, Y} := C \cup \{ x'_o := x_o, x_o \leq x'_o, x'_o = f(X) \} \), and
- \( A_{X_I, Y} := \{ x_o \} \),

and where \( x_o \) and \( x'_o \) do not occur in \( X \). This CSP consists of only one variable and one constraint. Branching on the variable \( x_o \) corresponds to trying different intervals for the outcome of the objective function. Propagation of the solves constraint corresponds to verifying that for a given interval for \( x_o \), a solution to the combinatorial problem can be found for which the outcome objective function falls within this interval.

The constraint \( x'_o := x_o \) denotes a right-to-left assignment, according to the following reduction rule.

\[
\langle x'_o := x_o ; \ x_o \in [a..b], x'_o \in [c..d] \rangle
\]

\[
\langle x'_o := x_o ; \ x_o \in [a..b], x'_o \in [c..d] \cap [a..b] \rangle
\]

It copies the domain of the one variable \( x_o \) of the global CSP to the CSP (4) that is subject to the nested search. The constraint \( x_o \leq x'_o \) is similar, and copies back the new bound for the objective function to the global CSP:

\[
\langle x_o \leq x'_o ; \ x_o \in [a..b], x'_o \in [c..d] \rangle
\]

\[
\langle x_o \leq x'_o ; \ x_o \in [a..b] \cap (\neg \infty..d], x'_o \in [c..d] \rangle
\]

Because \( x_o \in A_{X_I, Y} \), the latter constraint need not be enforced in the CSP (5), and any solved form will still yield a valid generate-and-test scheme. In practice, we require (5) to be in arc solved form (or an appropriate approximation thereof, e.g. hull solved form for a real valued objective function), in order that the domain of \( x_o \) is updated for any solution found.

**Example 3.** Let \( P := \langle C \mid X \in D \rangle \) be a combinatorial problem for which \( f(X) \in [0..100] \). Suppose the minimum value of \( f \) for any solution is 23. Now the branch-and-propagate search for this minimum, on the CSP (7), having \( l = 0 \) and \( h = 100 \) proceeds as follows. During the constraint propagation phase, reduction rule (6) is applied. This effectively checks that a solution to \( P \) exists for which \( f \) falls in the current range \([0..100]\) for \( x_o \). This is the case, so suppose that for the
particular solved form (5) found during the nested search \( f(X) \) equals 36. If the constraint \( x_o \leq x'_o \) is enforced, for example because we compute an arc solved form, then the domain of \( x_o \) is reduced to \([0..36]\). If we compute just a solved form, the domain is not updated. In both cases, the domain is not yet a singleton set, and we proceed by branching on \( x_o \). Suppose the constraint \( x_o \leq x'_o \) has been enforced. The branching yields subdomains \([0..18]\) and \([19..36]\). Because we do a depth-first leftmost-first search, we continue the branch-and-propagate search in the \( x_o \in [0..18] \) branch. It will turn out that no solution to \( P \) lies in this range. Then search proceeds in the \( x_o \in [18..36] \) branch, and so on, until finally the domain of \( x_o \) has been narrowed down to the set \{23\}. Because the search on (7) is depth-first leftmost-first, this is guaranteed to be the minimum.

A typical branch-and-propagate solver will apply the nested search in both branches, when splitting the domain of \( x_o \). This behaves slightly different from the scheme described in [4], which immediately splits the right branches again, and only tries to solve the combinatorial problem for left branches.

### 3.2 Box Consistency

Box consistency [5] refers to an approximation of arc consistency that is used for arithmetic constraints and an interval representation of the domains of numerical variables. It avoids the effect that for multiple occurrences of variables, a decomposition of an arithmetic constraint into atomic constraints yields a weaker form of consistency. Intuitively, a CSP is box consistent with respect to a constraint \( C \) on variables \( x_1, \ldots, x_n \) having respective interval domains \( D_1, \ldots, D_n \), if for every \( 1 \leq j \leq n \) the following condition holds: the values at the bounds of \( D_j \) satisfy the unary interval constraint obtained by replacing in the interval extension of \( C \) every occurrence of every variable \( x_i \) other than \( x_j \) by the constant interval \( D_i \).

For a representation of real numbers by means of floating point intervals, the values at the bounds of the domains are intervals that will not be split into further subdomains, either because the bounds are identical or consecutive floating point numbers, or because the width of the interval is smaller than the precision that we are interested in.

In [7] a very general algorithm for enforcing box consistency is given, where procedures \texttt{LeftNarrow} and \texttt{RightNarrow} search in the domain of a single variable for the leftmost and rightmost canonical interval that satisfy the unary interval constraint described above. Both operators can be described by instantiations of the constraint (3). For example the constraint that the leftmost canonical interval in \( D_j \) must satisfy constraint \( C \), with all other variables replaced by their respective domains can be formulated as

\[
\text{solves}_L(X_I, X_O, Y, D_Y, C_{X_I, Y}, A_{X_I, Y}),
\]
where
\[ X_I := x_1, \ldots, x_n, \quad X_O := x_j, \]
\[ Y := x'_j, \quad D_Y := \mathbb{R}, \]
\[ C_{X_I,Y} := \{x'_j/x_j\}C, \quad x'_j := x_j, \quad x_j \geq x'_j \}, \]
and
\[ A_{X_I,Y} := \{x_1, \ldots, x_n\}. \]

For the local CSP (4) all variables other than \( x'_j \) are auxiliary variables, so only the domain of \( x'_j \) is bisected. \( \{x'_j/x_j\}C \) denotes the constraint obtained by replacing in \( C \) all occurrences of variable \( x_j \) by a new variable \( x'_j \). The constraint \( \geq \) is similar to the constraint \( \leq \) defined above. \texttt{solvesL} indicates that a depth-first leftmost-first strategy is used to arrive at the solved form (5). The level of consistency that is enforced should ensure that by means of the constraint \( x_j \geq x'_j \) the lower bound of \( x_j \) is actually updated. In practice we require that (5) is in hull solved form, for \( C \) replaced by its decomposition into atomic constraints.

\textbf{Example 4.} Consider the constraint \( x^2 + x + y^2 + y = 0 \). Its decomposition into atomic constraints is \{\( x_2 = x^2, y_2 = y^2, x_2 + x + y_2 + y = 0 \}\}. For left-narrowing of \( x \), the constraint (8) is instantiated as follows: \( X_I := x, y, X_O := x, Y := x_2, y_2, x', D_Y := \mathbb{R}, \mathbb{R}, \mathbb{R}, A_{X_I,Y} := \{x, x_2, y, y_2\}, \) and
\[ C_{X_I,Y} := \{x_2 = x'^2, y_2 = y^2, x_2 + x' + y_2 + y = 0, x' := x, x \geq x'\}. \]

The local CSP (4) that the reduction rule (6) works on is then
\[ \langle C_{X_I,Y} ; x \in D_x,y \in D_y,x_2 \in \mathbb{R}, y_2 \in \mathbb{R}, x' \in \mathbb{R} ; \{x, x_2, y, y_2\} \rangle. \]

By means of the constraint \( x' := x \) the domain of \( x' \) is set to \( D_x \), initially. The nested search branches only on \( x' \), and because the search is depth-first leftmost-first, the hull solved form (5) will have the domain of \( x' \) set to the smallest canonical interval for which \( C_{X_I,Y} \) does not fail in presence of the current \( D_y \), or to the empty set if no hull solved form exists. By means of the constraint \( x \geq x' \) the lower bound of \( D_x \) is updated accordingly.

\[ \Box \]

\section{3.3 Shaving}

Shaving refers to a consistency technique used for solving scheduling problems. We refer to the description of this technique in [8], and use job-shop scheduling as an example.

A job-shop scheduling problem (JSSP) instance consists of a set of activities, and a number of machines. An activity is characterized by the machine that it must be processed on, and by a processing time. Activities are grouped in jobs, where all activities of a job have to be executed in a specified order. The problem is to find for each activity an interval in which it can be executed on the specified machine, such that no two activities require the same machine simultaneously, and such that the precedence constraints inside the jobs are respected. An optimal schedule minimizes the makespan of the schedule, i.e. the completion time of the activities that finish last.
A possible CSP formulation of the JSSP contains an integer variable (interval representation) for the starting time and for the completion time of each activity. Here we consider the procedure for the starting times. The procedure for the completion times is analogous. The idea is to remove (shave off) values for the lower bound of the starting time that will not lead to a feasible schedule. Let $\langle C ; X \in D \rangle$ be the CSP representation of a scheduling problem. Shaving the starting time $s_i \in \text{elem}(X)$ of activity $i$ can be described as enforcing the constraint

$$\text{solves}_L(X_I, X_O, Y, D_Y, C_{X_I,Y}, A_{X_I,Y}),$$

where

\begin{align*}
X_I & := X, & X_O & := s_i, \\
Y & := s'_i, & D_Y & := \mathbb{N}, \\
C_{X_I,Y} & := \{\{s'/s_i\}C, s'_i := s_i, s_i \geq s'_i\}, \text{ and} \\
A_{X_I,Y} & := \text{elem}(X),
\end{align*}

and $\{s'/s_i\}C$ denotes the set of constraints obtained by replacing $s_i$ by $s'_i$ in every constraint $C \in C$. Again we require that the solved form (5) complies to a form of local consistency that ensures that $s_i$ is updated through the constraint $s_i \geq s'_i$. In [4] it is suggested that for $C$ we use the level of consistency achieved by the edge finding algorithm.

Example 5. The ft20 benchmark JSSP consists of 20 jobs, each having 5 activities, that require the 5 different machines. To implement shaving for this problem, we need 200 operators: one for each of the 100 activity starting times, and one for each of the completion times. Consider the operator for one of the starting times. The nested search finds the smallest value for this starting time that has the property that if the activity is actually scheduled to start at that time, regular constraint propagation on the full problem, involving all 199 other starting times and completion times, does not lead to a failure. In the global CSP, all earlier starting time are removed from the domain of the variable. □

A depth-first leftmost-first bisection search on the domain of $s'_i$ will correctly update the lower bound of $s_i$, but in [8] a different branching scheme is described: first try to shave off a single value, and double the size of interval to shave off until an interval is found that allows for a feasible schedule. Then search for the lower bound in this interval by regular bisection.

The locality of the shaving operation is in between that of the other two applications. Box consistency performs nested search on a single variable, and considers only (the decomposition of) a single constraint. For optimization, the nested search is on all variables, considering all constraints. Like box consistency, nested search for shaving applies to only one variable, but consistency checking involves all constraints.

4 Implementation

The reduction operator described in Sect. 2.2 was implemented for the OpenSolver [10] platform. OpenSolver is a copying-based branch-and-prune tree search
engine that is configurable through a plug-in mechanism. Plug-ins come in a number of categories, corresponding to various aspects of constraint solving, notably:

- Variable domain types, such as finite domain and interval representations of integers, and a representation of real numbers using floating point intervals.
- Reduction operators. There are three kinds of reduction operators:
  - Propagation operators, or domain reduction functions (DRFs) that remove inconsistent values from the domains of variables. These are activated only during the pruning stage.
  - Branching operators that create new nodes of the search tree by splitting the domains of variables. These are activated on internal nodes of the search tree, after the pruning stage.
  - Optimization operators. These are propagation operators that may modify their state during the branching stage.
- Schedulers that apply reduction operators.
- Containers of nodes of the search tree, and selectors that examine the containers defining the state of the search algorithm and decide which nodes will be processed next.

The plug-in mechanism is similar to that used for DICE (DIstributed Constraint Environment), and is described in more detail in [9].

In addition to plug-ins that implement aspects of the constraint solving algorithm, every OpenSolver instance has a coordination layer plug-in that controls the solving process, and allows information to be shared with the environment. The solver executes a command loop, where it continually asks the coordination layer plug-in what to do next. Examples of commands that can be given are:

- Process a problem specification text. Such texts are formatted in a simple language for activating plug-ins. Figure 2 shows an example, which is discussed in Sect. 5.2.
- Run the scheduler of propagation operators. This executes the pruning phase in the nodes of the search tree that have been selected for exploration.
- Apply the selector that determines where to expand the search tree, and apply the branching operators in the selected nodes.
- Export the domain of a certain variable, in a given node of the search tree. When receiving this command, the solver passes the domain information to the coordination layer plug-in. This plug-in can share this information with the environment, for example to implement a distributed constraint propagation algorithm.

The operator for nested search is implemented by having an almost autonomous OpenSolver instance act as a propagation operator. A special coordination layer plug-in forms the interface between the solver that uses the nested search, and the solver that executes the nested search. A benefit of this approach is that all facilities of the framework are immediately available for nested search, but for operators that are activated often, the overhead is significant.
In the language for activating plug-ins, reduction operators are introduced by the following statement:

\[ \text{DRF identifier \{specifier\}}; \]

For example, the following text installs an instance of a plug-in called \text{RIARule} (real interval arithmetic rule), and provides this plug-in with the specifier string \( x^1 \cdot (1) \leq y \) for initialization:

\[ \text{DRF RIARule \{ x^1 \cdot (1) \leq y \}}; \]

This plug-in enforces the constraint \( x \leq y \). In combination with the operator

\[ \text{DRF RIARule \{ y^1 \cdot (-1) \leq -1 \cdot x \}}; \]

it will enforce the constraint \( x \leq y \). Problem specifications in this language are typically generated by a solver front-end, that performs symbolic manipulation of constraints.

Figures 1 and 2 show example operators for nested search. The specifier string consists of

- the name of a boolean variable, whose only purpose is that its domain will be voided if the solved form (5) does not exist;
- the variables of set \( X_I \). The current implementation assumes \( X_O = X_I \). To prevent that non-output variables are modified, local versions of these variables must be introduced, and domains have to be copied by means of constraint propagation;
- between curly brackets, an OpenSolver problem specification that configures a solver for the CSP (4). The variables of \( X_I \) have to be introduced again, but the domains will be provided by the coordination layer plug-in each time the operator is activated.

When it is activated during constraint propagation, the \text{NestedSearch} operator will pass the domains of the variables in \( X_I \) to the interface coordination layer, and then it will run the local OpenSolver instance. The interface coordination layer will issue commands for importing the new domains, and for a regular first solution search. When a solution is found, a command is issued to export the variable domains that we are interested in. The call-back functions provided for this purpose by the interface coordination layer will update the domains of the arguments to the \text{NestedSearch} operator. If no solution is found, the domain of the boolean variable is voided.

5 Experiments

In this section we detail the experiments that we performed on the optimization and box consistency applications discussed in Sect. 3. No experiments have yet been performed for shaving. Reported run times are the minimum elapsed time observed for five repeated runs on a 1200 MHz Athlon PC.
5.1 Optimization: Job-Shop Scheduling

We tested the optimization technique described in Sect. 3.1 on the job-shop scheduling problem. A number of special-purpose plug-ins have been developed for solving JSSP instances in OpenSolver. These implement the following facilities:

- a variable domain type \texttt{Activity} for representing the earliest starting time and latest completion time of an activity of a given duration;
- a variable domain type whose values represent different permutations of sequences of a given length. These represent the ordering of the activities that require the same machine;
- a reduction operator that links these two variable domain types, by enforcing an ordering on a set of activities;
- reduction operators for the \textit{disjunctive} constraint, stating that two activities that require the same machine cannot overlap in time, and the \textit{edge finding} algorithm. Both are described in [4].

The approach described in Sect. 3.1 is considered as an alternative for regular branch-and-bound, where each time a feasible schedule is found, further solutions are constrained to improve on the makespan of the current schedule. The last solution found during a complete search is then the optimal schedule.

Figure 1 shows the OpenSolver input for the approach of Sect. 3.1. The code inside the \texttt{NestedSearch} operator consists largely of the JSSP problem, where the length of the schedule must fit the domain of the integer variable \texttt{bound}.

The \texttt{BoundActivity} operator is a special-purpose plug-in that converts an \texttt{Activity} to an \texttt{IntegerInterval}: the earliest starting time and the latest completion time are constrained to be equal to the lower and upper bound, respectively, of the domain of the integer variable. Here this is applied to a virtual activity \texttt{makespan}, that is scheduled to start after the last activity of every job, and to an integer representation \texttt{imakespan} that is linked to the variable \texttt{bound} on which the outer bisection search is applied. The \texttt{IIARule} operators enforce the constraints \texttt{imakespan} := \texttt{bound} and \texttt{bound} \leq \texttt{imakespan}. The parameter 0 in the specifier for the \texttt{FailFirst} plug-in specifies a bisection where the left branch is generated last. By default, the search frontier is maintained as a stack, so this results in a depth-first, leftmost-first exploration.

Bisection search on the length of the schedule does not give the best performance on every benchmark, but the following result indicates that it is a useful tool. With a chronological backtracking strategy, it finds the optimal solution of the \texttt{ft20} benchmark in 3 minutes, but using branch-and-bound, the optimum has not yet been found after 1 hour. For scheduling problems, there are much better alternatives to chronological backtracking, and these can also be used in combination with the nested search, but a discussion of these is beyond the scope of this paper.
VARIABLE bound IS IntegerInterval {0..5110};
AUX b IS Bool {0,1};
DRF NestedSearch { b, bound, {
    AUX bound IS IntegerInterval {};
    AUX imakespan IS IntegerInterval {};
    VARIABLE makespan Is Activity {0,0,5110};
    ...
    code for the JSSP, where makespan executes after the last activity of every job.
    ...
    DRF BoundActivity { makespan, imakespan };
    DRF IIARule { imakespan - 1 * (1) = bound };
    DRF IIARule { bound - 1 * (1) <= imakespan };
} }
DRF FailFirst { 0, bound };

Fig. 1. OpenSolver code for bisection search for the minimal makespan of a JSSP

5.2 Box Consistency for Arithmetic Constraints on the Reals

The implementation of box consistency for constraints on the reals was tested on the Broyden banded function, a benchmark that is often used to demonstrate the advantage of box consistency over hull consistency, for example in [5]. The problem is to find the zeros of the functions

\[ f_i(x_1, ..., x_n) = x_i(2 + 5x_i^2) + 1 - \sum_{j \in J_i} x_j(1 + x_j) \quad (1 \leq i \leq n), \]

where \( J_i = \{ j \mid j \neq i, \max(1, i - 5) \leq j \leq \min(n, i + 1) \} \). All variables have initial ranges \([-1, 1]\).

Every function \( f_i \) depends on the 2 to 7 variables in the set \( J_i \cup \{ x_i \} \), and for every variable that a function depends on, two reduction operators are generated: one for the left narrowing, and one for the right narrowing. Figure 2 shows the operator that implements the left-narrowing for argument \( x_1 \) of the function \( f_3 \). Variable \( lx1 \) corresponds to \( x'_j \) used in Sect. 3.2. It is linked to \( x_1 \) by the first two RIARule operators. Auxiliary variables \( x1, ..., x4 \) are given their domain upon activation of the reduction rule. The other RIARule operators evaluate \( f_3 \), using \( fx1, ..., fx4 \) to store intermediate results. The resulting interval for \( f_3 \) is used to update the variable \( zero \), whose domain contains only the value 0, so this effectively implements a generate-and-test search for the leftmost zero of the unary constraint on \( x_1 \).

The following results demonstrate the (well known) effect of computing box consistency for this benchmark: computation time increases linearly with the problem size. The target precision is 1.0e-8, but inside the nested search we split down to machine precision. This way, the Broyden banded function is solved
Fig. 2. Left-narrowing for argument $x_1$ of $f_3(x_1, x_2, x_3, x_4)$ of the Broyden banded function by propagation alone. The reported numbers are elapsed times in seconds, for problem instances specified by $n$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
<th>320</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>2.514</td>
<td>6.137</td>
<td>13.410</td>
<td>28.118</td>
<td>58.017</td>
<td>118.807</td>
</tr>
</tbody>
</table>

This is of course only a very basic implementation of box consistency, and there is much room for improvement. Because the consistency check is implemented by constraint propagation, it should be easy to add propagators for the Newton reduction step described in [5]. Also using a nearly full OpenSolver instance for each reduction operator brings about considerable overhead. We plan to extend the coordination layer mechanism with a command for first solution search, to bypass the command loop for a long period, but a dedicated operator will likely always be faster.

5.3 Box Consistency for Arithmetic Constraints on the Integers

The box consistency technique can also be used for arithmetic constraints on integer variables. To illustrate that this is a useful application, consider the following constraint.

$$x^2y^2 - 4x^2y + 4x^2 - 4xy^2 + 16xy - 16x + 4y^2 - 16y + 16 = 4$$
and ranges \( x, y \in [0..10^5] \). When we solve this equation by means of decomposition into atomic constraints [3], the 8 solutions are found in 15.394 sec. in a search tree of 40255 nodes. When the code for the decomposed constraint is packed in four different \texttt{NestedSearch} operators, for left- and right narrowing of \( x \) and \( y \), the search tree is reduced to 39 nodes, and exploration takes less than 4 seconds.

6 Discussion

We have shown that a number of powerful constraint solving techniques can be expressed as applications of an operator for nested search. We implemented this operator for the OpenSolver framework. In this implementation, each instance of the reduction operator is an almost autonomous OpenSolver instance. An advantage of this approach is that all facilities of the framework are available for specifying the nested search. A disadvantage is the overhead of using a general-purpose solver for very specific search problems. Experiments show that our approach yields basic but viable implementations for two applications, that employ the operator at different levels of granularity: optimization by bisection search on the range of the objective function, and enforcing box consistency for arithmetic constraints on the reals and integers.

Another advantage of our approach is that it allows for composition of constraint solvers from basic facilities. From this point of view it is desirable to have a small set of basic operators and combinators, that can be used to realize a wide range of constraint solving techniques. It seems reasonable that compositionality comes at the cost of some computational overhead for the framework.

There is an analogy between our operator for nested search and the procedure mechanism in procedural programming languages. The input and output variables can be seen as by-value and by-reference parameters, respectively, and the OpenSolver input for the local CSP can be seen as a procedure body code. It would be interesting to investigate this analogy further, perhaps to come to some notion of a parameterized reduction operator. This way we may be able to avoid duplicate code for activating plug-ins. To illustrate that this is a useful facility, for the Broyden banded function, the code for each function (as shown in Fig. 2) is repeated up to 14 times, with minor differences for left or right narrowing, and for the particular variable that we want to update. The OpenSolver input for \( n = 320 \) contains 128979 lines.

Our immediate plans for future work on this topic are:

- Gain experience with the proposed implementation of the shaving operator on the job-shop scheduling problem. For this we have to extend the facilities described in Sect. 5.1 with a special-purpose operator for the branching scheme outlined at the end of Sect. 3.3.
- Extend the OpenSolver preprocessor for arithmetic constraints with an option to generate code for enforcing box consistency. This option should also generate the reduction operators corresponding to the Newton reduction
step [5]. The code of Fig. 2 is generated by a program written specifically for this benchmark.

The operator defined in Sect. 2 performs a single solution search, and in our applications we use this for updating the bounds of interval variables. For finite domain variables, it is attractive to have a variant that searches for all solutions. We can then use enumeration instead of bisection, and use the union of the domains for each variable as the result of the reduction operator. A potential application is to enforce singleton arc consistency [6]. It is also possible to define box-consistency (and the other applications discussed here) in terms of this all-solution variant, but all internal zeros would be lost because of the interval representation. Therefore the first solution search is a more realistic implementation of these techniques.

References