Course Notes
Constraint Programming (ID2204)
VT 2012

Christian Schulte
ICT-KTH

Contents

Chapter 2

Constraint Propagation

This chapter introduces a model for constraint propagation. Here, we distinguish constraint satisfaction problems as specifications from constraint models as implementations. Constraint propagation for constraint models is introduced first in a naive version which allows us to concentrate on some essential properties. This is followed by two more realistic versions of constraint propagation.

The mathematical concepts used in this chapter are summarized in Appendix A.

2.1 Constraint Satisfaction Problems

Constraint satisfaction problems (CSPs) are problem specifications: a CSP defines the variables, the values, and the constraints of a problem. We consider a CSP as a *declarative* specification: it defines the set of solutions (the *what*) but it does not define *how* to compute this set of solutions. Section 2.2 will explore how to solve constraint satisfaction problems.

A CSP is defined by its variables, values, and its constraints. A constraint is defined *extensionally* by giving all possible value combinations for its variables.

Definition 2.1 (CSP) *A constraint satisfaction problem (CSP) is a triple $\langle V, U, C \rangle$:*

- *V is a finite set of variables,*

- *U is a finite set of values,*

- *C is a finite set of constraints. A constraint c is a pair $\langle v, s \rangle$ where $v \in V^n$ are the variables of c and $s \subseteq U^n$ are the solutions of c for some $n \in \mathbb{N}$ which is called the arity of c.*

var (c) *refers to the variables of a constraint c; similarly,* sol (c) *refers to the solutions and* arity (c) *to the arity of c.*

The following example CSP E_1 is used as a running example as the presentation proceeds.

Example 2.1 (The CSP E_1) Given is the CSP $E_1 = \langle V, U, C \rangle$ where $V = \{x, y, z\}$, $U = \{1, 2, 3, 4\}$, and $C = \{c_1, c_2\}$. The constraints are defined by:

$$\begin{aligned} \text{var}\left(c_1\right) &= \langle x, y \rangle & \text{sol}\left(c_1\right) &= \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle\} \\ \text{var}\left(c_2\right) &= \langle y, z \rangle & \text{sol}\left(c_2\right) &= \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle\} \end{aligned}$$

The definition of a constraint in a CSP typically involves only a subset of all variables V. However, in order to define when values for all variables are solutions to a constraint (and a CSP), we introduce *assignments*: functions mapping all variables to values as follows.

Definition 2.2 (Assignment) *For a given set of variables V and a given universe U an assignment a is a function from V to U: $a \in V \rightarrow U$.*

As the set of variables is finite we write an assignment $a \in V \rightarrow U$ with $a(x_i) = n_i$ for $V = \{x_1, \ldots, x_k\}$ as follows: $\{x_1 \mapsto n_1, \ldots, x_k \mapsto n_k\}$.

Example 2.2 (Assignment) Suppose the set of variables $V = \{x, y, z\}$ and the set of values $U = \{1, 2, 3\}$. Then $a \in V \rightarrow U$ defined by $a(x) = 2$, $a(y) = 3$, and $a(z) = 1$ is an assignment.

The assignment can be written more conveniently as

$$a = \{x \mapsto 2, y \mapsto 3, z \mapsto 1\}$$

The following definition establishes the connection between assignments and solutions of a constraint:

Definition 2.3 (Assignments as Constraint Solution) *An assignment $a \in V \rightarrow U$ is a solution of a constraint c (written $a \in c$), if*

$$\text{var}(c) = \langle x_1, \ldots, x_n \rangle \quad \textit{and} \quad \langle a(x_1), \ldots, a(x_n) \rangle \in \text{sol}(c)$$

That is, for the variables var (c) of a constraint c the values defined by the assignment a must be solutions of c.

Example 2.3 (Assignments for E_1) An assignment $a \in V \rightarrow U$ with $a \in c_1$ and $a \in c_2$ is:

$$a = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$$

The solutions of a CSP are assignments that are solutions of *all* constraints of the CSP. This is defined as follows:

Definition 2.4 (Solutions of a CSP) *The set of* solutions *of a CSP* $\mathscr{C} = \langle V, U, C \rangle$ *is defined as*

$$\mathrm{sol}(\mathscr{C}) = \{a \in V \to U \mid a \in c \text{ for all } c \in C\}$$

Example 2.4 (Solutions of E_1) The set of solutions $\mathrm{sol}(E_1)$ of E_1 is as follows:

$$\{\{x \mapsto 1, y \mapsto 2, z \mapsto 3\}, \{x \mapsto 2, y \mapsto 3, z \mapsto 4\}\}$$

As mentioned at the beginning of this section, constraint satisfaction problems are used as specifications only. We will consider a different approach for solving, based on functions computing with sets of possible values. While the extensional representation of constraints serves as a simple way of specifying problems, it has some drawbacks as it comes to solving:

Space Requirement. A constraint is defined by all possible solutions, requiring considerable memory during solving.

Loss of Structure. We discussed that propagation is typically achieved by efficient algorithms: for a particular constraint (such as `distinct`) we want to use a particular algorithm. Finding out which algorithm to use from an extensional representation is difficult.

Therefore, today's constraint programming systems perform constraint propagation by using propagators (also known as filtering algorithms) as independent components to achieve constraint propagation for a particular constraint. This is detailed in the following sections.

Exercise 2.1 Given is the CSP $\langle V, U, C \rangle$ where $V = \{x, y, z\}$, $U = \{1, 2, 3\}$, and $C = \{c_1, c_2, c_3\}$. The constraints are defined by:

$$
\begin{aligned}
\mathrm{var}(c_1) &= \langle x, y \rangle & \mathrm{sol}(c_1) &= \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 3 \rangle\} \\
\mathrm{var}(c_2) &= \langle z, x \rangle & \mathrm{sol}(c_2) &= \{\langle 2, 1 \rangle, \langle 3, 2 \rangle, \langle 3, 1 \rangle\} \\
\mathrm{var}(c_3) &= \langle y, z \rangle & \mathrm{sol}(c_3) &= \{\langle 3, 2 \rangle, \langle 3, 1 \rangle, \langle 2, 3 \rangle, \langle 2, 1 \rangle, \langle 1, 3 \rangle\}
\end{aligned}
$$

1. Give an assignment $a \in V \to U$ with $a \in c_1$ and $a \notin c_3$.

2. Give an assignment $a \in V \to U$ with $a \in c_3$ and $a \notin c_2$.

3. Give the set of solutions of the CSP.

2.2 Constraint Models

A constraint model implements a constraint satisfaction problem: while it also defines variables and values, it defines *propagators* instead of constraints. A propagator is a function which performs constraint propagation. This section clarifies the essential properties of a propagator and when a constraint models can be considered an implementation of a CSP.

Constraint Stores.   Propagators compute with sets of possible values for variables, this is captured in the following definition:

Definition 2.5 (Constraint Store) *A constraint store $s \in V \rightarrow 2^U$ is a function mapping variables to sets of values.*

The set of all constraint stores is referred to by $S = V \rightarrow 2^U$.

We will often briefly refer to a constraint store as store.

Essential to how propagators compute with constraint stores is that they make a constraint store "stronger" by removing impossible values for variables. The intuition of "strength" of a constraint store is made clear by the following definition.

Definition 2.6 (Stronger Stores) *A constraint store $s_1 \in V \rightarrow 2^U$ is stronger than a constraint store $s_2 \in V \rightarrow 2^U$ (written $s_1 \leq s_2$), if $s_1(x) \subseteq s_2(x)$ for all variables $x \in V$.*

A constraint store $s_1 \in V \rightarrow 2^U$ is strictly stronger than a constraint store $s_2 \in V \rightarrow 2^U$ (written $s_1 < s_2$), if $s_1(x) \subseteq s_2(x)$ for all variables $x \in V$ and there exists a variable $x \in V$ with $s_1(x) \subset s_2(x)$.

If s_1 is (strictly) stronger than s_2, we also say that s_2 is (strictly) weaker than s_1.

Note that if a store s_1 is strictly stronger than a store s_2, then $s_1 \leq s_2$ and $s_1 \neq s_2$.

It is important to note that the set of stores together with the stronger-relation $(S, \leq)$ is a partial order. This can be seen easily by checking the properties required for a partial order (see A.2).

The weakest possible store $s_w \in V \rightarrow 2^U$ maps each variable to the entire universe:

$$s_w(x) = U \qquad \text{for all } x \in V$$

whereas the strongest possible store $s_s \in V \rightarrow 2^U$ is

$$s_s(x) = \emptyset \qquad \text{for all } x \in V$$

Example 2.5 (Constraint Stores) Suppose that $V = \{x, y\}$ is the set of variables and $U = \{1, 2, 3\}$ is the set of values.

Consider the following stores (using for stores the same notation as for assignments):

$$
\begin{aligned}
s_1 &= \{x \mapsto \{1, 2\}, y \mapsto \{2, 3\}\} \\
s_2 &= \{x \mapsto \{2\}, y \mapsto \{2, 3\}\} \\
s_3 &= \{x \mapsto \{2, 3\}, y \mapsto \{1, 2, 3\}\}
\end{aligned}
$$

Then $s_2 < s_1$ and $s_2 < s_3$ (and of course also $s_2 \leq s_1$ and $s_2 \leq s_3$). However, it also holds that $s_1 \not\leq s_3$ and $s_3 \not\leq s_1$: the stores s_1 and s_3 are incomparable. This is due to the fact that store-strength is a partial but not a total order.

Suppose two stores $s_1, s_2 \in V \to 2^U$ with $s_1 < s_2$. An equivalent statement to this is that $s_1 \leq s_2$ and there exists an $x \in V$ such that $s_1(x) \subset s_2(x)$.

The strict stronger-relation on stores $\langle S, < \rangle$ is also well-founded. This is a direct consequence from the fact that both the set of variables and the set of values is finite for constraint stores.

In the following we will need to make connections between assignments and stores as follows:

Definition 2.7 (Assignment and Stores) *An assignment $a \in V \to U$ is contained in a store $s \in V \to 2^U$ (written $a \in s$), if:*

$$
a(x) \in s(x) \qquad \text{for all } x \in V
$$

A store $s \in V \to 2^U$ is an assignment-store, *if:*

$$
|s(x)| = 1 \qquad \text{for all } x \in V
$$

For an assignment $a \in V \to U$, the store $\text{store}(a) \in V \to 2^U$ is defined as follows:

$$
\text{store}(a)(x) = \{a(x)\} \qquad \text{for all } x \in V
$$

By construction, $\text{store}(a)$ is an assignment-store.

Example 2.6 (Assignments and Stores) Suppose the set of variables $V = \{x, y, z\}$ and the set of values $U = \{1, 2, 3\}$ and the assignment $a = \{x \mapsto 2, y \mapsto 3, z \mapsto 1\}$. Then

$$
\text{store}(a) = \{x \mapsto \{2\}, y \mapsto \{3\}, z \mapsto \{1\}\}
$$

In the following discussion on propagator properties we will use a simple fact about the connection between assignments and assignment stores:

Proposition 2.1 *For an assignment $a \in V \rightarrow U$ and a store $s \in V \rightarrow 2^U$, the following holds:*

$$a \in s \qquad \Longleftrightarrow \qquad \text{store}(a) \leq s$$

    Proof: The proof is straightforward by using the appropriate definitions.

$$
\begin{aligned}
a \in s \iff{}& a(x) \in s(x) \quad \text{for all } x \in V \\
\iff{}& \{a(x)\} \subseteq s(x) \quad \text{for all } x \in V \\
\iff{}& \text{store}(a)(x) \subseteq s(x) \quad \text{for all } x \in V \\
\iff{}& \text{store}(a) \leq s
\end{aligned}
$$

$\square$

Exercise 2.2 Consider the CSP from Exercise 2.1.

1. Give the strongest store $s \in V \rightarrow 2^U$ containing all solutions to the CSP:

$$\text{for all } a \in \text{sol}(\langle V, U, C \rangle) : \qquad a \in s$$

2. Assume an arbitrary CSP $\langle V, U, C \rangle$ and $s \in V \rightarrow 2^U$ to be the strongest store containing all solutions of the CSP. Does the following hold:

$$\text{sol}(\langle V, U, C \rangle) = \{a \in V \rightarrow U \mid a \in s\}$$

Implementing Constraints. The following discussion develops when a propagator implements a constraint and what are the properties a propagator must satisfy. For the time being, we just assume that a propagator p is a function from constraint stores to constraint stores:

$$p \quad \in \quad S \rightarrow S$$

A straightforward property of a propagator p is that it is only allowed to remove values from constraint stores but never add values. A propagator p must be *contracting*:

$$p(s) \leq s \qquad \text{for all stores } s \in S$$

Our intuition is that a propagator p implementing a constraint c is never allowed to remove solutions of c. So a basic requirement is that if an assignment a is a solution of a constraint c (that is, $a \in c$), then

$$p(\text{store}(a)) = \text{store}(a)$$

8

What we insist on, is that a similar property holds for arbitrary and not only for assignment stores: if a store s contains a solution a of a constraint c, then a propagator p implementing c is not allowed to remove a by propagation. Hence, we are interested in the following property to hold:

$$a \in s \quad \Longrightarrow \quad a \in p(s)$$

From $a \in s$ it follows that $\text{store}(a) \leq s$ according to Proposition 2.1. Similarly, from $a \in p(s)$ it follows that $\text{store}(a) \leq p(s)$. The missing link is: if $\text{store}(a) \leq s$, then also $\text{store}(a) \leq p(s)$. The property of a propagator that will guarantee this is monotonicity. Now we are in the position to define what a propagator is.

Definition 2.8 (Propagator) *A* propagator *is a function $p \in S \rightarrow S$ from stores to stores which is:*

1. *Contracting: for all $s \in S$: $p(s) \leq s$.*

2. *Monotonic: for all $s_1, s_2 \in S$: $s_1 \leq s_2 \Longrightarrow p(s_1) \leq p(s_2)$.*

After this definition we can also state when a propagator is considered to be an implementation of a constraint.

Definition 2.9 (Implementation) *A propagator $p \in S \rightarrow S$ with $S = V \rightarrow 2^U$ is an* implementation *of a constraint c, iff for all $a \in V \rightarrow U$:*

$$a \in c \quad \Longleftrightarrow \quad p(\text{store}(a)) = \text{store}(a)$$

Assume that p is a propagator implementing the constraint c and that a is an assignment which is not a solution of c, that is $a \notin c$. Then we know from the above definition that $p(\text{store}(a)) \neq \text{store}(a)$. As p is contracting, we have that $p(\text{store}(a)) < \text{store}(a)$. This means that there exists a variable x for which $p(\text{store}(a))(x) \subset \text{store}(a)(x)$. As $\text{store}(a)(x)$ is a singleton (contains a single element) we know that $p(\text{store}(a))(x) = \emptyset$. Stores which map a variable to an empty set do not contain any solution at all. This is captured in the following definition.

Definition 2.10 (Failed Store) *A store $s \in S = V \rightarrow 2^U$ is* failed*, if there exists a variable $x \in V$ such that $s(x) = \emptyset$.*

A propagator $p \in S \rightarrow S$ fails *on a store $s \in S$, if $p(s)$ is failed.*

The fact that a propagator p implements a constraint c can also be characterized by the following slogan: propagators distinguish solution assignments from non-solution assignments. Solution assignment stores are fixpoints of p whereas p fails on non-solution assignment stores.

Example 2.7 (Propagator for $\leq$) Assume that $V = \{x, y, z\}$ and $U = \{0, \ldots, 5\}$. A propagator $p_\leq$ for $x \leq y$ can be defined as follows:

$$\begin{aligned}
p_\leq(s)(x) &= \{n \in s(x) \mid n \leq \max s(y)\} \\
p_\leq(s)(y) &= \{n \in s(y) \mid n \geq \min s(x)\} \\
p_\leq(s)(z) &= s(z)
\end{aligned}$$

For a store

$$s = \{x \mapsto \{3, 4, 5\}, y \mapsto \{0, 1, 2, 3\}, z \mapsto \{1, 5\}\}$$

the propagator $p_\leq$ computes

$$\begin{aligned}
p_\leq(s) = \{ \ &x \mapsto \{n \in s(x) \mid n \leq 3\}, \\
&y \mapsto \{n \in s(y) \mid n \geq 3\}, \\
&z \mapsto s(z)\} \\
= \{ \ &x \mapsto \{3\}, y \mapsto \{3\}, x \mapsto \{1, 5\}\}
\end{aligned}$$

Now we can summarize the above discussion which has motivated monotonicity for propagators in the following proposition:

Proposition 2.2 (Implementation) *Assume a propagator $p \in S \to S$ with $S = V \to 2^U$ implementing a constraint c and $a \in V \to U$ a solution of c, that is $a \in c$. Then for all stores $s \in S$:*

$$a \in s \quad \Longrightarrow \quad a \in p(s)$$

Proof: As discussed earlier, the proposition is a direct consequence of monotonicity:

$$\begin{aligned}
a \in s \ &\Longrightarrow \ \text{store}(a) \leq s && \text{Proposition 2.1} \\
&\Longrightarrow \ p(\text{store}(a)) \leq p(s) && p \text{ is monotonic} \\
&\Longrightarrow \ \text{store}(a) \leq p(s) && a \text{ is solution} \\
&\Longrightarrow \ a \in p(s) && \text{Proposition 2.1}
\end{aligned}$$

$\square$

Exercise 2.3 Assume that $V = \{x, y\}$ and $U = \{1, 2, 3, 4\}$. The set P is defined as the set of functions from stores to stores over V and U.

Answer whether the functions $p \in P$ are propagators. If the function is *not* a propagator, give a counter example showing that the function is not contracting or monotonic.

1. $p(s) = \{x \mapsto \emptyset, y \mapsto \emptyset\}$

2. $p(s) = \{x \mapsto \{n \in s(x) \mid n \leq 2\}, y \mapsto \{n \in s(y) \mid n \geq 3\}\}$

3. $p(s) = \{x \mapsto s(x) - s(y), y \mapsto s(y) - s(x)\}$

4. $p(s) = \left\{ \begin{array}{l} x \mapsto \text{if } |s(y)| = 1 \text{ then } s(x) - s(y) \text{ else } s(x) \\ y \mapsto \text{if } |s(x)| = 1 \text{ then } s(y) - s(x) \text{ else } s(y) \end{array} \right\}$

5. $p(s) = \left\{ \begin{array}{l} x \mapsto s(x) - \{1\} \\ y \mapsto \text{if } s(y) \subseteq s(x) \text{ then } s(y) \text{ else } s(y) - \{2, 3, 4\} \end{array} \right\}$

Constraint Models   The notion of a propagator implementing a constraint is mostly motivated by illustration purposes. In general, we will not require that there is a direct one-to-one correspondence between propagators and constraints. Instead, we will work out that constraint models featuring several propagators implement constraint satisfaction problems featuring several constraints.

Definition 2.11 (Constraint Model) *A constraint model $\mathcal{M} = \langle V, U, P \rangle$ is defined by a finite set of variables V, a finite set of values (the universe) U, and a finite set of propagators P over V and U.*

The solutions of a propagator and a constraint model are defined following the slogan that propagators can distinguish solutions from non-solutions.

Definition 2.12 (Propagator Solutions) *The set of* solutions $\text{sol}(p) \subseteq V \to U$ *of a propagator p is defined as:*

$$\text{sol}(p) = \{a \in V \to U \mid p(\text{store}(a)) = \text{store}(a)\}$$

Definition 2.13 (Constraint Model Solutions) *The set of* solutions $\text{sol}(\mathcal{M}) \subseteq V \to U$ *of a constraint model $\mathcal{M} = \langle V, U, P \rangle$ is defined as:*

$$\text{sol}(\mathcal{M}) = \bigcap_{p \in P} \text{sol}(p)$$

Finally, we are in the position to state when a constraint model implements a CSP.

Definition 2.14 (Implementation) *A constraint model $\mathcal{M} = \langle V, U, P \rangle$ implements a constraint satisfaction problem $\mathcal{C} = \langle V, U, C \rangle$, if*

$$\text{sol}(\mathcal{M}) = \text{sol}(\mathcal{C})$$

In the following we will be interested only in those solutions which can be obtained by starting from some initial store s. These are exactly those solutions which are stronger than the initial store s.

Definition 2.15 (Constraint Model Solutions) *The set of* solutions $\text{sol}(\mathcal{M}, s) \subseteq V \to U$ *of a constraint model* $\mathcal{M} = \langle V, U, P \rangle$ *for a store* $s \in V \to 2^U$ *is defined as:*

$$\text{sol}(\mathcal{M}, s) = \{a \in \text{sol}(\mathcal{M}) \mid \text{store}(a) \leq s\}$$

Given the above definition, the Proposition 2.1 can be reformulated for constraint models as follows:

Proposition 2.3 (Propagators Preserve Solutions) *Assume a constraint model* $\mathcal{M} = \langle V, U, P \rangle$ *and a store* $s \in V \to 2^U$. *Then for all propagators* $p \in P$:

$$\text{sol}(\mathcal{M}, s) = \text{sol}(\mathcal{M}, p(s))$$

Exercise 2.4 Given is the CSP $\langle V, U, C \rangle$ where $V = \{x, y, z\}$, $U = \{1, 2, 3\}$, and $C = \{c_1, c_2, c_3\}$. The constraints are defined by:

$$\text{var}(c_1) = \langle x, y \rangle \qquad \text{sol}(c_1) = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 1 \rangle, \langle 2, 3 \rangle, \langle 3, 1 \rangle, \langle 3, 2 \rangle\}$$
$$\text{var}(c_2) = \langle y, z \rangle \qquad \text{sol}(c_2) = \text{sol}(c_1)$$
$$\text{var}(c_3) = \langle x, z \rangle \qquad \text{sol}(c_3) = \text{sol}(c_1)$$

Give two different constraint models implementing the CSP $\langle V, U, C \rangle$.

2.3 Naive Constraint Propagation

Our first step in defining how to actually do constraint propagation for a constraint model is to start with a naive version. Naive constraint propagation allows us to focus on the main ideas and the most fundamental properties of constraint propagation while ignoring efficiency.

Constraint propagation will be provided as a function

$$\text{propagate} : Models \times S \to S$$

taking a constraint model M and a constraint store s as input and returning a new store on which constraint propagation has been performed. Clearly, it needs to be seen what is exactly computed by this propagation function.

Naive constraint propagation is performed by the following algorithm.

Algorithm 2.1 (Naive Propagation) *For a constraint model* $\langle V, U, P \rangle$ *naive constraint propagation is performed as follows:*

```
propagate(⟨V, U, P⟩,s)
   while exists p ∈ P with p(s) ≠ s do
      s := p(s);
   return s;
```

The essential questions we need to answer are: does the loop always terminate? What is computed by naive constraint propagation? This is formulated in the following theorem.

Theorem 2.4 (Naive Propagation) *Naive constraint propagation as defined by Algorithm 2.1 has the following properties:*

1. *Naive constraint propagation always terminates.*

2. *If propagate(⟨V, U, P⟩, s) = s′, then* sol(⟨V, U, P⟩, s) = sol(⟨V, U, P⟩, s′).

3. *If propagate(⟨V, U, P⟩, s) = s′, then s′ is the weakest (largest) simultaneous fixpoint of all propagators in P:*

    (a) *s′ is simultaneous fixpoint:*

    $$p(s′) = s′ \qquad for\ all\ p ∈ P$$

    (b) *s′ is the weakest (largest) simultaneous fixpoint stronger than s (s′ ≤ s): any other simultaneous fixpoint t of P is stronger than s′ (t ≤ s′).*

Proof:

1. Consider the sequence of stores s_i after the i-th iteration of the while-loop, defined as

    $$s_i := p_i(s_{i-1}) \qquad for\ i > 0$$

    and

    $$s_0 := s$$

    where p_i is the propagator selected at iteration $i > 0$.
    From the loop condition $p(s) ≠ s$ and the fact that propagators are contracting, it follows that

    $$s_{i+1} < s_i \qquad for\ i > 0$$

    That is, the sequence of stores s_i forms a strictly decreasing sequence. As $⟨S, <⟩$ is well-founded, the sequence must be finite. Hence, the loop terminates.

2. This is a direct consequence of Proposition 2.3 (by induction).

3. Only the last statement requires an explicit proof, the first one is a direct consequence of how the loop is constructed:

   (a) After termination of the loop, the negation of the loop condition holds: for all $p \in P$ we have that $p(s') = s'$.

   (b) Assume that t is a simultaneous fixpoint of P which is stronger than s. We will show that t is stronger than s' by an inductive proof of the following fact:

   $$t \leq s_i \qquad \text{for } i \geq 0$$

   where the s_i are as above. As the loop terminates, we know that there exists $n \in \mathbb{N}$ such that $s_n = s'$. From this the statement $t \leq s'$ follows.

   Base case ($i = 0$) This holds, as we assume that $t \leq s = s_0$.

   Induction step Under the induction assumption we have:

   $$
   \begin{aligned}
   t \leq s_i \implies\ & p_{i+1}(t) \leq p_{i+1}(s_i) \\
   & \qquad p_{i+1} \text{ is monotonic} \\
   \implies\ & t = p_{i+1}(t) \leq p_{i+1}(s_i) \\
   & \qquad t \text{ is fixpoint of } p_{i+1} \\
   \implies\ & t \leq p_{i+1}(s_i) = s_{i+1} \\
   & \qquad \text{definition of } s_{i+1} \\
   \implies\ & t \leq s_{i+1}
   \end{aligned}
   $$

   $\square$

The above theorem has a very important consequence: the order in which propagators are applied (that is, which propagator is chosen for application in the while loop) does not affect the result of constraint propagation. As an immediate consequence one can see that systems implementing constraint propagation can take advantage of this effect by choosing propagators in an order than can save runtime.

The two main properties expressed in the above theorem are direct consequences of properties of the individual propagators. Naive propagation terminates because propagators are contracting. They compute the weakest simultaneous fixpoint, because they are contracting and monotone.

2.4   Realistic Constraint Propagation

Naive propagation has the drawback that it insists on applying only propagators which are guaranteed to compute a strictly stronger constraint store. This is of course difficult, as the only way to find out whether a propagator will actually compute a strictly stronger store is by applying it.

A better idea is to maintain some information about which propagators are at fixpoint. Hence, it is needed that changes to stores that do not affect propagators can be identified. This will be done by identifying a set of variables of interest for a propagator as follows.

Definition 2.16 (Propagator Variables) *For a propagator $p \in S \to S$, the* variables var $(p) \subseteq V$ *of p are defined as the smallest set of variables such that:*

1. *For all $s \in S$ and all $x \in (V - \text{var}(p))$:*

$$p(s)(x) = s(x)$$

   This captures that p computes output only for variables in var (p).

2. *For all $s_1, s_2 \in S$: If for all $x \in \text{var}(p)$: $s_1(x) = s_2(x)$, then for all $x \in \text{var}(p)$: $p(s_1)(x) = p(s_2)(x)$.*

   This captures that the propagator p considers input only from the variables in var (p).

We say that a propagator p depends *on a variable x, if $x \in \text{var}(p)$.*

It is important to consider the smallest possible set as this will help avoiding useless propagator applications during constraint propagation.

Example 2.8 (Propagator Variables) For the propagator $p_\leq$ from Example 2.7 the set of variables var $(p_\leq)$ is $\{x, y\}$. Note that we insist on the smallest possible set, hence $z \notin \text{var}(p_\leq)$.

Improved constraint propagation has two main ideas:

- Maintain a set Q of "dirty" propagators: these are the propagators not known to be at fixpoint.

- Consider only those propagators DP as dirty, which have variables in common with those variables modified by propagation MV.

Algorithm 2.2 (Variable-centered Propagation) *Variable-centered constraint propagation for a constraint model $\langle V, U, P \rangle$ is performed as follows:*

```
propagate(⟨V,U,P⟩,s)
   N := P;
   while N ≠ ∅ do
     choose p ∈ N;
     s' := p(s); N := N − {p};
     MV := {x ∈ V | s(x) ≠ s'(x)};
     DP := {q ∈ P | var (q) ∩ MV ≠ ∅};
     N := N ∪ DP;
     s := s';
   return s;
```

Essential for understanding variable-centered propagation is the loop invariant for the while-loop in Algorithm 2.2. This is formulated in the following proposition:

Proposition 2.5 (Loop Invariant) *An invariant I for the while-loop in Algorithm 2.2 is*

$$I = \left(\text{for all } p \in P - N : \quad p(s) = s \right)$$

> **Proof:** We need to show that I holds initially and that if I holds before an iteration of the loop it also holds after that iteration.
>
> - I holds initially as N is initialized to be P.
> - Assume I holds before iteration of the loop. To hold after one iteration we have to show that
>
> $$\text{for all } p \in P - (N \cup DP) : \quad p(s') = s'$$
>
>   Let us assume that $p \in P - (N \cup DP)$, that is $p \in P$ and $p \notin N \cup DP$. From $p \notin N \cup DP$ it follows that there exists no $x \in MV$ such that $x \in \text{var}(p)$. That is, for all $x \in MV$ it holds that $x \notin \text{var}(p)$. From Definition 2.16 follows that $s'(x) = p(s')(x)$ for all $x \in MV$. By definition of MV this means that $s'(x) = p(s')(x)$ for all $x \in V$, hence $p(s') = s'$.

$\square$

The following theorem clarifies that besides avoiding some redundant applications of propagators, variable-centered propagation has the same properties as naive propagation.

Theorem 2.6 (Variable-centered Propagation) *Variable-centered constraint propagation has the following properties:*

1. *Variable-centered constraint propagation always terminates.*

2. *If* propagate($\langle V, U, P \rangle, s$) $= s'$, *then* sol($\langle V, U, P \rangle, s$) $=$ sol($\langle V, U, P \rangle, s'$).

3. *If* propagate($\langle V, U, P \rangle, s$) $= s'$, *then* s' *is the weakest (largest) simultaneous fixpoint of all propagators in* P.

Proof:

1. Consider the sequence of stores s_i after the i-th iteration of the while-loop, defined as

$$s_i := p_i(s_{i-1}) \qquad \text{for } i > 0$$

and

$$s_0 := s$$

where p_i is the propagator selected at iteration $i > 0$. Similarly, we will refer by N_i, MV_i, and DP_i to the values of N, MV, and DP after the i-th iteration.

It is helpful to make a case distinction as follows:

- $s_i = p_i(s_{i-1}) = s_{i-1}$. Then $MV_i = \emptyset$ and hence $N_i = N_{i-1} - \{p_i\}$ and $p_i \in N_{i-1}$. That is $N_i \subset N_{i-1}$.
- $s_i = p_i(s_{i-1}) \neq s_{i-1}$. Then $s_i < s_{i-1}$. The set N_i might contain more propagators than N_{i-1}.

Taking both cases together means that for pairs of stores and propagator sets

$$\langle s_i, N_i \rangle <_{lex} \langle s_{i-1}, N_{i-1} \rangle$$

with respect to the lexicographic order for $\langle S, < \rangle$ and $\langle 2^P, \subset \rangle$. The lexicographic order is well-founded as both orders are well-founded, the sequence must be finite. Hence, the loop terminates.

2. This is a direct consequence of Proposition 2.3 (by induction).

3. That s' is a simultaneous fixpoint follows directly from Proposition 2.5 considering that after termination of the loop $N = \emptyset$.

   The proof that s' is the *weakest* simultaneous fixpoint is exactly the same as for naive propagation: the only difference in the sequence of stores considered here is that the stores are not in strict order. However, this is not exploited by the proof for Theorem 2.4.

$\square$

Variable-centered propagation serves as starting point for the implementation of most constraint programming systems.

2.5 Exploiting Fixpoint Knowledge

Variable-centered propagation improves naive propagation by maintaining information on which propagators are at fixpoint. However, this information becomes only available by applying propagators. In the following we will present two important propagator properties that provide information about fixpoints. However, the optimization of the propagation loop relies on the fact that the propagators themselves tell the propagation loop about the fixpoint optimization.

Subsumtion. Consider the propagator p defined by

$$p(s)(x) = s(x) \cap \{1, 2, 3\}$$

implementing the domain constraint $x \in \{1, 2, 3\}$. After p has been executed once, there is no point to execute p again as for all stores s'

$$s' \leq p(s) \quad \Longrightarrow \quad p(s') = s'$$

Clearly, the propagator p can be deleted from the set of all propagators after being executed.

The propagator p is subsumed by the store $p(s)$. The general case is covered by the following definition.

Definition 2.17 (Subsumtion) *A propagator p is* subsumed by *a store s, iff*

$$\textit{for all } s' \leq s : p(s') = s'$$

A store s subsumes a propagator p, if all stronger stores are fixpoints of p. Subsumtion is also known as entailment.

Idempotence. Consider the propagator $p_\leq$ from Example 2.7. One can see easily that after applying $p_\leq$ on a store, there is no point to apply $p_\leq$ directly again (both min and max will not change). With other words, for all stores s, $p_\leq(s)$ is a fixpoint of $p_\leq$.

18

Now assume that for some store s we have that $p_\le(s) < s$ (as in the Example). This means that $MV = \{x, y\}$ in Algorithm 2.2 after application of $p_\le$ even though we know that $p_\le$ is at fixpoint for $p_\le(s)$. This means that $p_\le$ should not be added to the "dirty" set N.

This property of $p_\le$ is known as idempotence:

Definition 2.18 (Idempotence) *A function $f \in X \to X$ is idempotent, if $f(f(x)) = f(x)$ for all $x \in X$.*

Example 2.9 (Non-idempotent Propagators) Many propagators are idempotent. Some widely used ones, however, are *not* idempotent. Consider the constraint $3x = 2y$ and the propagator p_{32} (assume that $V = \{x, y\}$ and that U is some finite subset of $\mathbb{Z}$):

$$
\begin{aligned}
p_{32}(s)(x) &= s(x) \cap \{\lceil (2 \min s(y))/3 \rceil, \ldots, \lfloor (2 \max s(y))/3 \rfloor\} \\
p_{32}(s)(y) &= s(y) \cap \{\lceil (3 \min s(x))/2 \rceil, \ldots, \lfloor (3 \max s(x))/2 \rfloor\}
\end{aligned}
$$

The propagator p_{32} is not idempotent. Consider

$$
s = \{x \mapsto [0\mathinner{..}3], y \mapsto [0\mathinner{..}5]\}
$$

Then $s' = p_{32}(s)$ is

$$
\begin{aligned}
s'(x) &= [0\mathinner{..}3] \cap [0\mathinner{..}\lfloor 10/3 \rfloor] &&= [0\mathinner{..}3] \\
s'(y) &= [0\mathinner{..}5] \cap [0\mathinner{..}\lfloor 9/2 \rfloor] &&= [0\mathinner{..}4]
\end{aligned}
$$

Now $s'' = p_{32}(s')$ is

$$
\begin{aligned}
s''(x) &= [0\mathinner{..}3] \cap [0\mathinner{..}\lfloor 8/3 \rfloor] &&= [0\mathinner{..}2] \\
s''(y) &= [0\mathinner{..}4] \cap [0\mathinner{..}\lfloor 9/2 \rfloor] &&= [0\mathinner{..}4]
\end{aligned}
$$

Hence $p_{32}(p_{32}(s)) = s'' \neq s' = p_{32}(s)$.

Idempotence is a very strong property as it requires for a propagator p that $p(s)$ is a fixpoint of p for arbitrary stores s. For our purposes it is actually too strong: the propagator does not need to be idempotent, it is sufficient that the propagator signals by some means whether the newly computed store is a fixpoint.

Definition 2.19 (Weak Idempotence) *A function $f \in X \to X$ is idempotent on $x \in X$, if $f(f(x)) = f(x)$.*

Weak idempotence is truly weak: if a propagator p is idempotent on s, it does not mean that p is idempotent on s' with $s' \le s$. That it can be useful is made clear by the following example:

Example 2.10 (Using Weak Idempotence) Consider applying p_{32} from Example 2.9 to the store s'' from the same example.

Now $s''' = p_{32}(s'')$ is

$$
\begin{aligned}
s'''(x) &= [0..2] \cap [0..\lfloor 8/3 \rfloor] &= [0..2] \\
s'''(y) &= [0..4] \cap [0..\lfloor 6/2 \rfloor] &= [0..3]
\end{aligned}
$$

Notice that the new bound $y \le 3$ is obtained without rounding as $\lfloor 6/2 \rfloor = 6/2$. In this case we are guaranteed that the propagator is at a fixpoint ([2] Theorem 8). In other words, while p_{32} is not idempotent on s, s', and s'' we know that it is idempotent on s''' even though we have not applied p_{32} to s''' in order to find out.

Exercise 2.5 (Equality Propagator) Assume $V = \{x, y\}$, $U = [-100 .. 100]$ and the following function e from stores to stores over V and U:

$$
e(s) = \{x \mapsto s(x) \cap s(y), y \mapsto s(x) \cap s(y)\}
$$

1. Prove that e is contracting.

2. Prove that e is monotonic.

3. Prove that e is idempotent.

Exercise 2.6 (Addition Propagator) Assume $V = \{x, y, z\}$, $U = [-10 .. 10]$, and the following propagator p implementing addition $x + y = z$:

$$
p(s) = \left\{ \begin{array}{lll}
x & \mapsto & \{n \in s(x) \mid \min s(z) - \max s(y) \le n \le \max s(z) - \min s(y)\} \\
y & \mapsto & \{n \in s(y) \mid \min s(z) - \max s(x) \le n \le \max s(z) - \min s(x)\} \\
z & \mapsto & \{n \in s(z) \mid \min s(x) + \min s(y) \le n \le \max s(x) + \max s(y)\}
\end{array} \right\}
$$

1. Show that p is not idempotent.

2. Give a universe U' for which p would be idempotent.

3. Give a non-failed store s on which p fails.

4. Give two different, non-failed stores s_1 and s_2 which both subsume p.

While both subsumtion and idempotence appear to be properties which can help to avoid unnecessary propagator re-execution, the question remains how to compute whether a propagator is subsumed by or idempotent on a store. In order to compute whether s subsumes a propagator p, it is of course way too costly to check for all

$s' \leq s$ that $p(s') = s'$. Similarly, to find out whether $p(s') = s'$ after a propagator p has computed s' is too costly: the additional application of p is what we hope to save in the first place.

The problem is addressed by status messages which are also returned by a propagator. So for the remainder of this section we assume that an extended propagator is a function

$$ep \in S \rightarrow SM \times S$$

returning a pair of status message and store where

$$SM = \{\texttt{nofix}, \texttt{fix}, \texttt{subsumed}\}$$

Clearly we insist that the returned store satisfies the necessary properties to make the propagator contracting and monotone.

Now assume an extended propagator ep and a store s. Then

- If $ep(s) = \langle \texttt{fix}, s' \rangle$, then s' is a fixpoint of ep (actually, it is a fixpoint for the corresponding non-extended propagator p which just returns the store).

- If $ep(s) = \langle \texttt{subsumed}, s' \rangle$, then s' subsumes ep.

- If $ep(s) = \langle \texttt{nofix}, s' \rangle$, then no further knowledge is available. This case corresponds to the normal case for non-extended propagators.

Example 2.11 (Extended Propagator for $\leq$) An extended version of the propagator $ep_\leq$ from Example 2.11 can also take into account that $p_\leq$ is subsumed by stores s with $\max s(x) \leq \min s(y)$. Hence, an extended propagator is

$$
\begin{aligned}
ep_\leq(s) = \quad &\textbf{let } s' = p_\leq(s) \textbf{ in} \\
&\textbf{if } \max s'(x) \leq \min s'(y) \textbf{ then } \langle \texttt{subsumed}, s' \rangle \\
&\textbf{else } \langle \texttt{fix}, s' \rangle
\end{aligned}
$$

Propagation now also changes the set of propagators, as subsumed propagators are deleted. Therefore, the propagation function taking advantage of extended propagators returns a pair containing the set of extended propagators and the store computed by propagation.

Algorithm 2.3 (Improved Propagation) *For an extended constraint model $\langle V, U, EP \rangle$ (where we assume that EP are extended propagators), improved constraint propagation is performed as follows:*

```
propagate(⟨V,U,EP⟩,s)
  N := EP;
```

```
while N ≠ ∅ do
    choose ep ∈ N;
    ⟨ms,s′⟩ := ep(s); N := N − {ep};
    if ms = subsumed then EP := EP − {ep};
    MV := {x ∈ V | s(x) ≠ s′(x)};
    DP := {eq ∈ EP | var (eq) ∩ MV ≠ ∅};
    if ms = fix then DP := DP − {ep};
    N := N ∪ DP;
    s := s′;
return ⟨EP,s⟩;
```

There are of course many more ideas available for improving constraint propagation, such as rewriting and prioritizing of propagators. For some more information consider for example [4].

2.6 Brief Summary and Further Reading

This chapter has introduced constraint satisfaction problems as problem specifications and constraint models as their implementations. We know how to perform constraint propagation on stores as described by a constraint model implementing some specification. Constraint propagation is well-behaved in the sense that it always computes the weakest simultaneous fixpoint while not removing solutions. This good behavior is due to the fact that the sets of variables and values are finite and that propagators are contracting and monotonic.

This chapter gives a very brief and condensed account on principles behind constraint propagation — just enough to understand the most essential properties of constraint propagation and how today's constraint programming systems work. For an excellent and much more detailed account on the principles of constraint programming, the reader might want to consult [1].

Chapter 3

Search

This chapter presents how to search for solutions of a constraint model. One important ingredient is of course constraint propagation which is performed after each search step. The remaining ingredients are: branchings defining search trees; how search trees are constructed with respect to a constraint model and a branching; and last but not least, how to explore the search tree.

3.1 Branchings

The purpose of a branching is to suggest new constraints for the decomposition of a problem into smaller or simpler subproblems on which constraint propagation can be re-applied. However, we are considering branchings in the context of constraint models. This means that decomposition is defined by propagators instead of constraints.

As has been discussed, it is vital for a branching to make an informed decision, typically following some application-dependent heuristic. This requires that a branching must have access to at least the following information:

Store. This is necessary, for example, to find the variable with the currently smallest number of possible values.

Propagator Set. For example, this is required to find the *most-constrained* variable: the variable on which the largest number of propagators depend.

A branching must be well-behaved: it is absolutely necessary that the search tree remains finite (there is obviously no need to consider infinite search trees as there are only finitely many different stores). Considering the solutions to our constraint

model, we insist that no solutions are lost during search. Furthermore, it is desirable that also solutions are not duplicated (too much duplication could get us back again to an infinite search tree).

This is reflected in the following definition, where we assume that $\mathtt{propagate}(P,s)$ refers to naive (Algorithm 2.1) or variable-centered (Algorithm 2.2) propagation of the propagators P on a store s which just returns a store obtained by constraint propagation.

Definition 3.1 (Branching) *Assume a constraint model $\mathcal{M} = \langle V, U, P \rangle$ and $S = V \rightarrow 2^U$ the set of all stores for $\mathcal{M}$.*

A branching *for $\mathcal{M}$ is a function b taking a set of propagators $Q \subset S \rightarrow S$ and a store $s \in S$ as input and returns an n-tuple $\langle Q_1, \ldots, Q_n \rangle$ of sets of propagators $Q_i \subseteq S \rightarrow S$. The set of propagators Q_i is called the i-th* alternative.

Assume that

$$A = \mathrm{sol}\left(\langle V, U, Q \rangle, s\right)$$

and

$$A_i = \mathrm{sol}\left(\langle V, U, Q \cup Q_i \rangle, s\right) \qquad for\ 1 \leq i \leq n$$

Then a branching b satisfies the following properties:

1. *It must be* complete: $\bigcup_{i=1}^{n} A_i = A$

2. *It must be* non-overlapping: $A_i \cap A_j = \emptyset$ *for $1 \leq i, j \leq n$ with $i \neq j$.*

3. *It must be* decreasing: $\mathtt{propagate}(Q \cup Q_i, s) < s$.

For a branching b, completeness guarantees that no solutions are lost. No solution appears twice in the search tree constructed for a branching b, this is because b is non-overlapping. That b is decreasing guarantees that it only suggests propagators which will trigger further constraint propagation. Hence the search tree will be finite. The next section makes clear how to construct a search tree for a given constraint model and branching.

Example 3.1 (Naive Branching) Assume that $V = \{x_1, \ldots, x_n\}$ and U arbitrary. Then a simple branching can be defined as

$$
\begin{aligned}
b(P,s) \quad = \quad & \text{if there exists } x \text{ with } |s(x)| > 1 \text{ and } s(x) = \{n_1, \ldots, n_k\} \text{ then} \\
& \quad \langle \{\mathrm{assign}(x, n_1)\}, \ldots, \{\mathrm{assign}(x, n_k)\} \rangle \\
& \text{else} \\
& \quad \langle\rangle
\end{aligned}
$$

where the propagator assign(u, n) for a variable $u \in V$ and a value $n \in U$ is defined as

$$\text{assign}(u, n)(s)(v) = \text{if } u = v \text{ then } \{n\} \cap s(v) \text{ else } s(v)$$

The branching considers any variable which is not yet assigned by the store s and does not enforce a certain ordering on the variables.

3.2 Search Trees

The properties for branchings are designed to yield well-behaved search trees. A search tree is defined as follows:

Definition 3.2 (Search Tree) *A search tree for a model $\langle V, U, P \rangle$ and a branching b is a tree where the nodes are labeled with pairs $\langle Q, s \rangle$ where Q is a set of propagators and s is a store obtained by constraint propagation with respect to Q.*

1. *The root of the tree is labeled with $\langle P, s \rangle$ where*

$$\texttt{propagate}(P, s_{init}) = s$$

   with

$$s_{init} = \lambda x \in V.U$$

2. *For all leaves $\langle Q, s \rangle$, either*

   - *s is failed. The leaf is called* failed.
   - *$b(Q, s) = \langle \rangle$. The leaf is called* solved.

3. *For an inner node $\langle Q, s \rangle$, then s is not failed and $b(Q, s) = \langle Q_1, \ldots, Q_n \rangle$ with $n \geq 1$. Then the node has n children where the i-th node ($1 \leq i \leq n$) is labeled*

$$\langle Q \cup Q_i, \texttt{propagate}(Q \cup Q_i, s) \rangle$$

A search tree has by construction several interesting properties. For a node $\langle Q, s \rangle$ in the search tree, s is a simultaneous fixpoint for Q. As discussed above, the decreasing property of a branching guarantees that a search tree is finite. That branchings are decreasing has also the consequence that: if the node $\langle Q_1, s_1 \rangle$ is on the same path but below node $\langle Q_2, s_2 \rangle$, then $s_1 < s_2$.

Notice the difference between a solved node (which is defined by the branching as it returns the empty tuple $\langle \rangle$) and a solution of a problem. As a solution of a problem we consider a leaf node where the store is an assignment store. This means that for a

branching resulting in solved nodes which feature assignment stores, the branching needs to assign all variables (or there must be the guarantee that those variables not directly assigned by a branching are assigned by constraint propagation).

The interaction between search and constraint propagation can be optimized. Assume an inner node $\langle Q, s \rangle$ and $b(Q, s) = \langle Q_1, \ldots, Q_n \rangle$ with $n \geq 1$. Then the i-th child node ($1 \leq i \leq n$) is labeled

$$\langle Q \cup Q_i, \mathtt{propagate}(Q \cup Q_i, s) \rangle$$

In this situation, s is a simultaneous fixpoint for all propagators in Q but definitely not for at least one $q \in Q_i$ (this is due to the fact that a branching is decreasing). Hence, in the Algorithms 2.2 and 2.3 it is sufficient to initialize the "dirty set" to Q_i rather than to $Q \cup Q_i$.

Exercise 3.1 Assume a constraint model $\langle V, U, P \rangle$ and a branching b defined as follows:

$$
\begin{aligned}
V &= \{x, y, z\} \\
U &= \{1, 2, 3, 4\} \\
P &= \{x < y, y < z\} \\
b(P, s) &= \text{if } |s(x)| > 1 \text{ and } s(x) = \{n_1, \ldots, n_k\} \text{ then} \\
&\qquad \langle \{\mathrm{assign}(x, n_1)\}, \ldots, \{\mathrm{assign}(x, n_k)\} \rangle \\
&\quad \text{else if } |s(y)| > 1 \text{ and } s(y) = \{n_1, \ldots, n_k\} \text{ then} \\
&\qquad \langle \{\mathrm{assign}(y, n_1)\}, \ldots, \{\mathrm{assign}(y, n_k)\} \rangle \\
&\quad \text{else} \\
&\qquad \langle \rangle
\end{aligned}
$$

where the propagator $\mathrm{assign}(u, n)$ for a variable $u \in V$ and a value $n \in U$ is defined as

$$\mathrm{assign}(u, n)(s)(v) = \text{if } u = v \text{ then } \{n\} \cap s(v) \text{ else } s(v)$$

Draw the resulting search tree.

3.3 Exploration

Given a constraint model and a branching, we know how a search tree is defined. An orthogonal question is how to incrementally construct a search tree until a first solved node has been found. This process of constructing a search tree until some criterion is met (such as first solution found, a certain number of solutions found, or even all solutions found) is referred to as *exploration*.

We will just consider depth-first, left-most exploration for the first solution as an example here. Furthermore, we restrict our attention to the case where a branching either returns the empty tuple $\langle\rangle$ or a pair $\langle Q_1, Q_2\rangle$. This is defined by the following algorithm.

Algorithm 3.1 (Depth-first Exploration) *Depth-first exploration takes a set of propagators P and a store s as input and returns a possibly failed store.*

```
dfe(P,s)
  s' := propagate(P,s);
  if s' failed then
    return s';
  else
    case b(P,s')
    of ⟨⟩       then
      return s';
    [] ⟨P₁,P₂⟩ then
      s'' := dfe(P ∪ P₁,s');
      if s'' failed then
        return dfe(P ∪ P₂,s');
      else
        return s'';
```

3.4 Programming Exploration

The previous section just briefly presented how to explore a search tree. In case you are inclined to learn more about exploration for search engines and even program some exploration algorithms, the book [3] might be a good place to look at.

Appendix A

Mathematical Prerequisites

This appendix reminds you of a few mathematical concepts used throughout the book.

A.1 Sets

The empty set is denoted by $\emptyset$.

For sets of numbers we use the following symbols: $\mathbb{N}$ refers to the natural numbers $\{0, 1, 2, \ldots\}$, $\mathbb{Z}$ refers to the integers $\{0, -1, 1, -2, 2, \ldots\}$, and $\mathbb{R}$ refers to the real numbers.

By $[n \mathinner{..} m]$ we denote the set $\{i \in \mathbb{Z} \mid n \leq i \leq m\}$.

Subset Relation. Given are two sets X and Y. We write $X \subseteq Y$, if X is a subset of Y, that is for all $x \in X$ we have that $x \in Y$. We write $X \subset Y$, if X is a proper (strict) subset of Y, that is $X \subseteq Y$ and there exists $y \in Y$ such that $y \notin X$.

Power Sets. For a given set X, we write the power set (the set of all subsets of X) of X as 2^X. For example, if $X = \{1, 2\}$ then $2^X = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$.

Set Cardinality. The cardinality of a given set X is denoted by $|X|$.

Cartesian Product. Given are sets $X_1, \ldots, X_n$. Then the Cartesian product $X_1 \times \cdots \times X_n$ is the set containing n-ary tuples:

$$\{\langle x_1, \ldots, x_n \rangle \mid x_1 \in X_1, \ldots, x_n \in X_n\}$$

Given a set X, we write X^n for the Cartesian product $X \times \cdots \times X$ (n-times).

A.2   Relations and Orders

Assume a set X and a binary relation $R \subseteq X \times X$ over X. Then

- R is *reflexive*, if for all $x \in X$: $\langle x, x \rangle \in R$.

- R is *irreflexive*, if for all $x \in X$: $\langle x, x \rangle \notin R$.

- R is *antisymmetric*, if for all $x, y \in X$: if $\langle x, y \rangle \in R$ and $\langle y, x \rangle \in R$, then $x = y$.

- R is *transitive*, if for all $x, y, z \in X$: if $\langle x, y \rangle \in R$ and $\langle y, z \rangle \in R$, then $\langle x, z \rangle \in R$.

We will often write binary relations using *infix* notation, that is $x \, R \, y$ instead of $\langle x, y \rangle \in R$, or $x \leq y$ instead of $\langle x, y \rangle \in \leq$

Definition A.1 (Partial Order) *A pair $\langle X, \leq \rangle$ where $\leq \, \subseteq X \times X$ is a* partial order, *if $\leq$ is reflexive, antisymmetric, and transitive.*

Definition A.2 (Strict Partial Order) *A pair $\langle X, < \rangle$ where $< \, \subseteq X \times X$ is a* strict partial order, *if $<$ is irreflexive and transitive.*

Definition A.3 (Well-founded Order) *A strict partial order $\langle X, < \rangle$ is* well-founded, *if there exists no infinite sequence $x_0, x_1, x_2, \dots$ with $x_i \in X$ and $x_{i+1} < x_i$ for $i \geq 0$.*

Example A.1 (Example Orders)

- $\langle 2^X, \subseteq \rangle$ is a partial order for any set X.

- $\langle 2^X, \subset \rangle$ is a strict partial order for any set X.

- $\langle 2^X, \subset \rangle$ is a well-founded order for any finite set X.

Definition A.4 (Lexicographic Order) *Assume that $\langle X, \leq_X \rangle$ and $\langle Y, \leq_Y \rangle$ are partial orders. Then $\langle X \times Y, \leq_{lex} \rangle$ defined by*

$$\langle x_1, y_1 \rangle \leq_{lex} \langle x_2, y_2 \rangle \iff (x_1 \leq_X x_2 \text{ and } x_1 \neq x_2) \text{ or } (x_1 = x_2 \text{ and } y_1 \leq_Y y_2)$$

is called the lexicographic order *for $\langle X, \leq_X \rangle$ and $\langle Y, \leq_Y \rangle$.*

Definition A.5 (Strict Lexicographic Order) *Assume that $\langle X, <_X \rangle$ and $\langle Y, <_Y \rangle$ are strict partial orders. Then $\langle X \times Y, <_{lex} \rangle$ defined by*

$$\langle x_1, y_1 \rangle <_{lex} \langle x_2, y_2 \rangle \Longleftrightarrow (x_1 <_X x_2) \text{ or } (x_1 = x_2 \text{ and } y_1 <_Y y_2)$$

is called the strict lexicographic order *for $\langle X, <_X \rangle$ and $\langle Y, <_Y \rangle$.*

The following propositions clarify some properties that carry over to the lexicographic order from the orders on which the lexicographic order is defined.

Proposition A.1 (Lexicographic Order) *The lexicographic order $\langle X \times Y, \leq_{lex} \rangle$ as defined in Definition A.4 is a partial order.*

If both $\langle X, \leq_X \rangle$ and $\langle Y, \leq_Y \rangle$ are strict partial orders, then also $\langle X \times Y, \leq_{lex} \rangle$ is a strict partial order.

If both $\langle X, \leq_X \rangle$ and $\langle Y, \leq_Y \rangle$ are well-founded orders, then also $\langle X \times Y, \leq_{lex} \rangle$ is a well-founded order.

  Proof: Left as exercise to the reader.   □

It is obvious that the concept of lexicographic order can be generalized to an arbitrary number of orders. However, for our cases the more specialized definition from above is sufficient.

A.3 Functions

The set of (total) functions from X to Y is denoted $X \rightarrow Y$.

For a function $f \in X \rightarrow X$, $x \in X$ is called a *fixpoint* of f, iff $f(x) = x$.

To define functions, we use λ-notation in places. Suppose that $f \in X \rightarrow Y$ and $f(x)$ is defined by some expression e (typically involving x) for all $x \in X$. Then we write

$$f = \lambda x \in X. e$$

This is just convenience as it does not require to introduce a name for a function.

If the set X is clear from the context (such as variables for propagators), we will also write

$$f = \lambda x. e$$

Iterated application of a function $f \in X \rightarrow X$ is defined as:

$$f^i(x) = \begin{cases} x & \text{if } i = 0 \\ f^{i-1}(f(x)) & \text{otherwise} \end{cases}$$

A.4   Rounding

Propagators involving division or multiplication over the integers require rounding. We will use the following two operations:

Definition A.6 (Floor and Ceiling) *For a real number $x \in \mathbb{R}$, the* floor $\lfloor x \rfloor$ *is the greatest integer $k \in \mathbb{Z}$ with $k \leq x$.*

For a real number $x \in \mathbb{R}$, the ceiling $\lceil x \rceil$ *is the smallest integer $k \in \mathbb{Z}$ with $k \geq x$.*

For example, $\lfloor 3.5 \rfloor = 3$, $\lceil 3.5 \rceil = 4$, $\lfloor -3.5 \rfloor = -4$, and $\lceil -3.5 \rceil = -3$.

Appendix B

Solutions to Selected Exercises

B.1 Solutions for Chapter 2

Solution 2.1

1. $a = \{x \mapsto 1, y \mapsto 2, z \mapsto 2\}$

2. $a = \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}$

3. $\mathrm{sol}(\langle V, U, C \rangle) = \{\{x \mapsto 1, y \mapsto 2, z \mapsto 3\}, \{x \mapsto 1, y \mapsto 3, z \mapsto 2\}\}$

Solution 2.2

1. $s = \{x \mapsto \{1\}, y \mapsto \{2, 3\}, z \mapsto \{2, 3\}\}$

2. No, of course not!

Solution 2.3

1. p is a propagator.

2. p is a propagator.

3. p is not a propagator, as it is not monotonic:

$$
\begin{aligned}
s_1 &= \{x \mapsto \{1\}, y \mapsto \{2\}\} \\
p(s_1) &= \{x \mapsto \{1\}, y \mapsto \{2\}\} \\
s_2 &= \{x \mapsto \{1, 2\}, y \mapsto \{1, 2\}\} \\
p(s_2) &= \{x \mapsto \emptyset, y \mapsto \emptyset\}
\end{aligned}
$$

That means we have $s_1 \leq s_2$, but $p(s_2) \leq p(s_1)$.

4. p is not a propagator as it is not monotonic on failed stores. Consider

$$
\begin{aligned}
s_1 &= \{x \mapsto \{1,2\}, y \mapsto \emptyset\} \\
p(s_1) &= \{x \mapsto \{1,2\}, y \mapsto \emptyset\} \\
s_2 &= \{x \mapsto \{1,2\}, y \mapsto \{2\}\} \\
p(s_2) &= \{x \mapsto \{1\}, y \mapsto \{2\}\}
\end{aligned}
$$

That means we have $s_1 \leq s_2$, but $p(s_1) \not\leq p(s_2)$.

If p is applied to non-failed stores only, it is an idempotent propagator (implementing binary disequality).

5. p is not a propagator as it is not monotonic:

$$
\begin{aligned}
s_1 &= \{x \mapsto \{1,3\}, y \mapsto \{1,3\}\} \\
p(s_1) &= \{x \mapsto \{3\}, y \mapsto \{1,3\}\} \\
s_2 &= \{x \mapsto \{1,3\}, y \mapsto \{1,2,3\}\} \\
p(s_2) &= \{x \mapsto \{3\}, y \mapsto \{1\}\}
\end{aligned}
$$

That means we have $s_1 \leq s_2$, but $p(s_1) \not\leq p(s_2)$.

Solution 2.4

1. The first model uses disequality propagators with a direct mapping between each constraint and each propagator. The constraint model is $\langle V, U, P \rangle$ where P is

$$
\{x \neq y, x \neq z, y \neq z\}
$$

2. A different model is by using a single distinct propagator: P is defined as

$$
\{\text{distinct}(x, y, z)\}
$$

Solution 2.5

1. We show that $e(s) \leq s$ for arbitrary s: it holds that $e(s)(x) = s(x) \cap s(y) \subseteq s(x)$ and $e(s)(y) = s(x) \cap s(y) \subseteq s(y)$.

2. Assume that $s_1 \leq s_2$, that is $s_1(x) \subseteq s_2(x)$ and $s_1(y) \subseteq s_2(y)$.

   From $s_1(x) \subseteq s_2(x)$ and $s_1(y) \subseteq s_2(y)$ it follows that $s_1(x) \cap s_1(y) \subseteq s_2(x) \cap s_2(y)$:

   $$
   \begin{aligned}
   n \in s_1(x) \cap s_1(y) &\implies n \in s_1(x) \text{ and } n \in s_1(y) \\
   &\implies n \in s_2(x) \text{ and } n \in s_2(y) \\
   &\implies n \in s_2(x) \cap s_2(y)
   \end{aligned}
   $$

   The same holds true for y.

3. We need to show that $e(e(s)) = e(s)$ for arbitrary s. This is easy to see:

   $$
   \begin{aligned}
   e(e(s)) &= e\left(\{x \mapsto s(x) \cap s(y), y \mapsto s(x) \cap s(y)\}\right) \\
   &= \left\{ \begin{array}{l} x \mapsto (s(x) \cap s(y)) \cap (s(x) \cap s(y)) \\ y \mapsto (s(x) \cap s(y)) \cap (s(x) \cap s(y)) \end{array} \right\} \\
   &= \{x \mapsto s(x) \cap s(y), y \mapsto s(x) \cap s(y)\} \\
   &= e(s)
   \end{aligned}
   $$

Solution 2.6

1. Use the store $\{x \mapsto \{0,4\}, y \mapsto \{0,4\}, z \mapsto \{0,1,2\}\}$

2. For example, $U' = \{0\}$.

3. $\{x \mapsto \{0\}, y \mapsto \{0\}, z \mapsto \{2\}\}$

4.
$$
s_1 = \{x \mapsto \{0\}, y \mapsto \{0\}, z \mapsto \{0\}\}
$$
$$
s_2 = \{x \mapsto \{1\}, y \mapsto \{1\}, z \mapsto \{2\}\}
$$

B.2 Solutions for Chapter 3

Solution 3.1

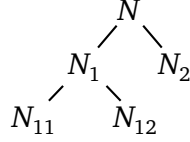The search tree consists of nodes obtained by constraint propagation starting from the weakest possible store.

The root node is

$$
\langle P, \{x \mapsto \{1,2\}, y \mapsto \{2,3\}, z \mapsto \{3,4\}\} \rangle
$$

and

$$b(P, \{x \mapsto \{1, 2\}, y \mapsto \{2, 3\}, z \mapsto \{3, 4\}\}) = \langle \{assign(x, 1)\}, \{assign(x, 2)\} \rangle$$

By iterating propagation and branching we get the following search tree:

where

$$
\begin{aligned}
N &= \langle P, \{x \mapsto \{1, 2\}, y \mapsto \{2, 3\}, z \mapsto \{3, 4\}\} \rangle \\
N_1 &= \langle P \cup \{assign(x, 1)\}, \{x \mapsto \{1\}, y \mapsto \{2, 3\}, z \mapsto \{3, 4\}\} \rangle \\
N_2 &= \langle P \cup \{assign(x, 2)\}, \{x \mapsto \{2\}, y \mapsto \{3\}, z \mapsto \{4\}\} \rangle \\
N_{11} &= \langle P \cup \{assign(x, 1), assign(y, 2)\}, \{x \mapsto \{1\}, y \mapsto \{2\}, z \mapsto \{3, 4\}\} \rangle \\
N_{12} &= \langle P \cup \{assign(x, 1), assign(y, 3)\}, \{x \mapsto \{1\}, y \mapsto \{3\}, z \mapsto \{4\}\} \rangle
\end{aligned}
$$

Bibliography

[1] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[2] Warwick Harvey and Peter J. Stuckey. Constraint representation for propagation. In *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming*, volume 1520 of *Lecture Notes in Computer Science*, pages 235–249. Springer-Verlag, October 1998.

[3] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Germany, 2002.

[4] Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In Mark Wallace, editor, *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633, Toronto, Canada, September 2004. Springer-Verlag.