Modern Constraint Programming (ESSAI'26) - DRAFT VERSION

Version of June 29, 2026

Emir Demirović

Contents

Preface

DISCLAIMER: This material is still being developed. Some statements made may not be precise or fully reflect the area. We are nevertheless releasing this material now so that summer school attendees can get a sense of the material and topics covered. By the start of the summer school, we will refine the material and address typographical issues, including references and section numbering.

Why another book on constraint programming? Constraint programming has undergone significant development over the past two decades. Techniques such as conflict analysis, nogood learning, lazy clause generation, certification, and proof systems have become fundamental components of many modern solvers. Beyond their practical impact, these developments have also provided new ways of understanding solver behaviour.

These advancements remain scattered across research papers and solver implementations, drawing on ideas from constraint programming, SAT, and mathematical programming. As a result, readers seeking to understand modern solver technology need to piece together a coherent picture from heterogeneous sources, most of which were written for specialists.

The goal of this book is to bring these developments together into a single, unified, and accessible presentation, effectively lowering the entry barrier to modern constraint solvers.

What is unique about this book? Beyond collecting recent developments in a single place, this book develops a formal view of constraint programming in which solver behaviour is interpreted as a sequence of mathematically justified derivations. Ultimately, a constraint solver is viewed not only as a search engine, but as a proof-producing machine. This perspective makes it easier to reason about the correctness of individual solver components and understand how local reasoning accumulates into proofs of infeasibility or optimality.

The book also aims to reduce the gap between introductory material and contemporary research. Readers should not only learn how solvers operate, but also acquire the conceptual foundation needed to understand recent research papers and ongoing developments.

Who is this book for? This book is intended for students, practitioners, and researchers interested in understanding how constraint programming solvers work. It assumes no prior

knowledge of constraint programming and introduces the fundamental concepts from first principles. At the same time, it covers modern solver techniques and recent developments from the research literature in an accessible form.

How is the material presented? The main philosophy of this book is to first introduce concepts at a high level and illustrate them through concrete examples. Visual explanations are extensively used to support intuition and highlight the key ideas. Once an intuitive understanding has been established, the concepts are formalised mathematically and developed in greater depth. This allows readers to first understand why a concept matters before focusing on its formal details and helps them develop their own algorithmic intuition within the framework of constraint programming.

Chapter 1

Foundations

After studying this chapter, you will be able to...

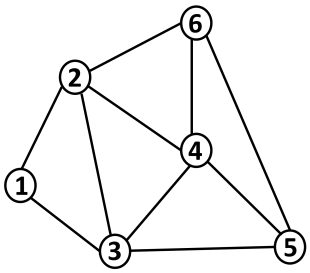
- describe variables, domains, partial assignments, and constraints, and explain how they jointly represent the state of a constraint solver;
- explain how search interacts with propagation, detects conflicts, and performs backtracking;

1 Illustrative Example

We begin by illustrating the main ideas of solving combinatorial optimisation problems using constraint programming on a concrete instance of *graph colouring*, a classical **NP**-complete problem. The key notions of search, propagation, and conflicts will be introduced informally through this running example. Formal definitions will follow afterwards.

The problem is as follows:

Is it possible to colour the vertices of the graph below using at most three colours (blue, orange, green) such that no two adjacent vertices share the same colour?

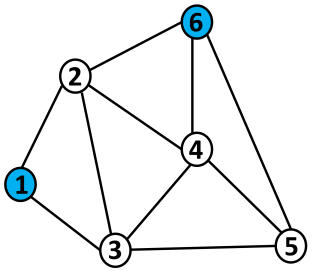


The most widely used approach in constraint programming to solve these types of problems is **(backtracking) search**. It performs a brute-force enumeration while using propagation to prune infeasible partial assignments. Although the worst-case running time is exponential, we will study techniques that make this general approach practical for many problems.

The general approach is as follows: we first make a *decision*, that is, we select an uncoloured vertex and assign it a colour consistent with those of its neighbours. We then use the constraints to restrict the remaining possibilities through *propagation*. Search interleaves decisions and propagation until we either find a feasible assignment or discover that the current partial assignment (implied by the decisions) is infeasible. When this happens, we undo the last decision and continue the search with a different one. We can apply this procedure to solve our graph colouring problem.

Initially, we start with a *partial assignment* where all vertices are uncoloured. We proceed with making a decision. Since this is the first decision, we say that we are at *decision level one*. We begin by assigning *blue* to vertex 1, extending our partial assignment. Because of the constraints, we know that our next decision cannot assign *blue* to adjacent vertices 2 and 3.

At *decision level two*, we then assign *blue* to vertex 6, and similarly note that vertex 4 and 5 cannot be assigned *blue*.

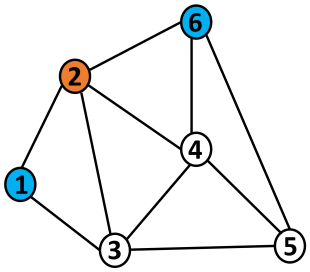


Decision Level	Assignment
1	$x_1 = \text{blue}$
	$x_2 \neq \text{blue}$
	$x_3 \neq \text{blue}$
2	$x_6 = \text{blue}$
	$x_4 \neq \text{blue}$
	$x_5 \neq \text{blue}$

All of our decisions and propagations are recorded on the *trail*, shown on the right side. The trail is a helper data structure that tracks every change to the domains, making backtracking easier when we later need to revert some of our decisions. In addition to the trail, we also maintain the remaining allowed values for each vertex, called *domains*, but for simplicity, we do not display domains. The trail, together with the initial problem, describes the current partial assignment.

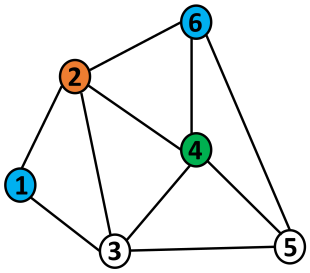
The type of deduction using constraints is called *propagation* (also known as inference or reasoning), and it is an essential component of backtracking search. Propagation differs from decisions in that it does not involve choice: these assignments are forced by the constraints and must occur for a feasible completion of the partial assignment to exist. For this reason, we conclude by propagation that vertices 2, 3, and 4 cannot be assigned *blue*.

Continuing with the search, at decision level three, we decide to colour vertex 2 *orange*, which removes *orange* as an option for neighbouring vertices 3 and 4 by propagation.



Decision Level	Assignment
1	$x_1 = \text{blue}$
	$x_2 \neq \text{blue}$
	$x_3 \neq \text{blue}$
2	$x_6 = \text{blue}$
	$x_4 \neq \text{blue}$
	$x_5 \neq \text{blue}$
3	$x_2 = \text{orange}$
	$x_3 \neq \text{orange}$
	$x_4 \neq \text{orange}$

We previously concluded that *blue* cannot be assigned to vertex 4, and we can now also conclude that *orange* is not a possible assignment for vertex 4. Given our earlier decisions, vertex 4 has only one option: *green*. This is not recorded on the trail since it is fully implied by the decisions and propagations. By propagation, we remove *green* as an option for vertices 3 and 5.



Decision Level	Assignment
1	$x_1 = \text{blue}$
	$x_2 \neq \text{blue}$
	$x_3 \neq \text{blue}$
2	$x_6 = \text{blue}$
	$x_4 \neq \text{blue}$
	$x_5 \neq \text{blue}$
3	$x_2 = \text{orange}$
	$x_3 \neq \text{orange}$
	$x_4 \neq \text{orange}$
	$x_5 \neq \text{green}$
	$x_3 \neq \text{green}$

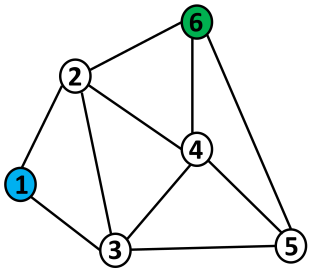
However, we now reach a situation where vertex 3 cannot be coloured, because all three colours appear among its neighbours. This is a *conflict*: the current partial assignment cannot be extended into a feasible one, regardless of any future decisions we may make. At this point, we know that we must *backtrack* on our previous decision.

In other words, our last decision (colouring vertex 2 with *orange*) was incompatible with earlier decisions. We therefore *backtrack* to the point when we made the decision $x_2 = \text{orange}$, record that the previous decision is not feasible, and continue from that point onwards as usual.

However, before we could make a new decision after backtracking, we already concluded that *orange* is not feasible for vertex 2 and that *blue* is also infeasible because vertices 1 and 6 are already coloured *blue*. The only remaining option for vertex 2 is *green*, which is assigned by propagation. Further inference concludes that vertex 4 must be coloured *orange*, leading to a conflict similar to the one encountered earlier.

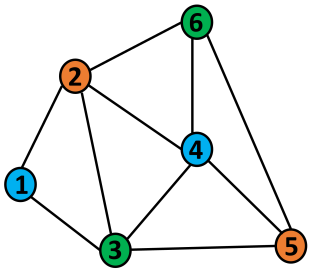
We backtrack once again to the last decision, this time to decision level 2. Since colouring vertex 6 with *blue* causes conflicts given that we previously decided vertex 1 is *blue*, we

make a decision to colour vertex 6 to *green*. Note that, after backtracking, we record at decision level 1 that *blue* is not feasible for vertex 6.



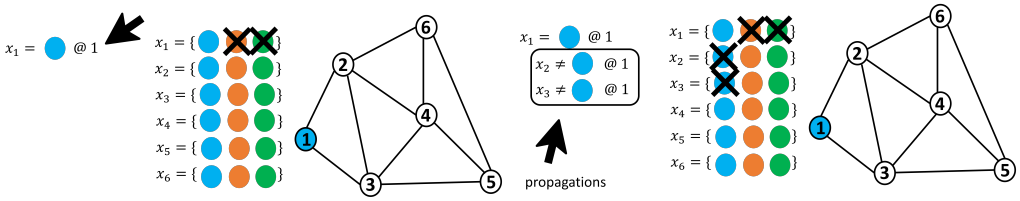
Decision Level	Assignment
1	$x_1 = \text{blue}$
	$x_2 \neq \text{blue}$
	$x_3 \neq \text{blue}$
	$x_6 \neq \text{blue}$
2	$x_6 = \text{green}$

The current partial assignment triggers a cascade of propagations: vertex 2 becomes *orange*, vertex 4 becomes *blue*, vertex 3 becomes *green*, and vertex 5 becomes *orange*. All vertices are now coloured, and we have found a feasible solution after making a total of four decisions, two of which are on the trail.



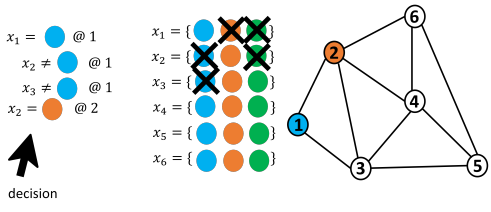
Decision Level	Assignment
1	$x_1 = \text{blue}$
	$x_2 \neq \text{blue}$
	$x_3 \neq \text{blue}$
	$x_6 \neq \text{blue}$
2	$x_6 = \text{green}$
	$x_2 \neq \text{green}$
	$x_4 \neq \text{green}$
	$x_5 \neq \text{green}$
	$x_4 \neq \text{orange}$
	$x_3 \neq \text{orange}$
	$x_5 \neq \text{blue}$

A more informed strategy for deciding the next vertex could have reduced the search effort. For example, colouring vertex 1 and then vertex 2 would already force the colours of all remaining vertices by propagation, effectively solving the problem using only two decisions instead of the four we required above. This is shown below.

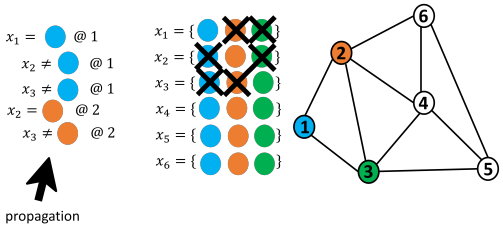


Decide on vertex 1.

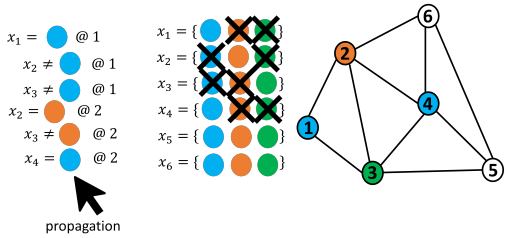
Propagation removes blue from neighbours.



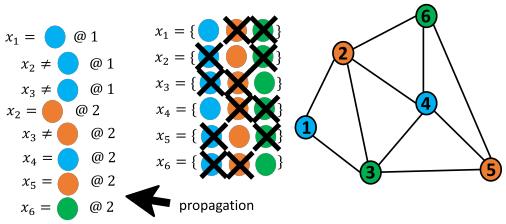
Decide on vertex 2.



Propagation assigns green to vertex 3.



Propagation assigns blue to vertex 4.



Propagation assigns the remaining vertices.

The *branching strategy* determines the decisions made during search. There are many different strategies, and they largely depend on the problem.

For this graph, we found a feasible assignment, so we say the instance is *feasible*. However, had there been an edge between vertices 1 and 4, our search procedure would correctly report that no feasible assignment exists; in other words, the instance would be *infeasible*.

2 Preliminaries

We now turn to formalising the constraint programming concepts we have seen informally in the example.

2.1 Variables and Domains

A variable represents an unknown quantity whose value is to be determined. Since we are *deciding* the values of the variables, we speak of *decision-making* problems.

Each variable x is associated with a domain $D(x)$, which specifies the set of values that x may assume. Depending on the nature of its domain, a variable may be classified as:

- *Binary*: $D(x) = \{0, 1\}$,
- *Integer* (or discrete): $D(x) \subseteq \mathbb{N}$,
- *Continuous* (or real-valued): $D(x) \subseteq \mathbb{R}$.

We restrict our attention to *finite* domains; however, many of the techniques presented in this book are also applicable to *infinite* domains.

Variables and domains have an intuitive meaning depending on the problem. For example, in graph colouring with three colours, each node in the graph is associated with an integer

variable with the domain $\{blue, orange, green\}$, where colours may be represented with integers. The domain thus represents the possible colours that may be assigned to each node.

Similarly, in scheduling, a task may be represented by a variable, and its domain specifies the allowable starting times for that task. For instance, the domain $\{5,6,7\}$ may represent the three possible starting times for a particular task.

The **lower bound** $\mathrm{LB}(x)$ of a variable x is the smallest value in its domain $D(x)$:

$$\mathrm{LB}(x) = \min\{v : v \in D(x)\}.$$

Conversely, the **upper bound** $\mathrm{UB}(x)$ of a variable x is the largest value in its domain:

$$\mathrm{UB}(x) = \max\{v : v \in D(x)\}.$$

The lower and upper bounds are only meaningful when the domain is numerical. When the domain represents categorical values, such as colours in graph colouring, the lower and upper bounds are still formally defined through the integer encoding of the categories, but they do not carry any meaningful interpretation.

The lower and upper bounds are key concepts, and many of the algorithms presented in this book will refer to them.

2.1.1 Domain Representation

In practice, the way variable domains are represented has a significant impact on both memory usage and computational efficiency. The chosen representation directly constrains how algorithms can operate over domains. We consider two prototypical representations: the *set representation* and the *interval representation*.

Set representation Each value in the domain is stored explicitly in a set. This representation captures domains precisely without loss of information. The drawback is that operations that modify domains may require inspecting all values, and memory usage grows proportionally with the domain size, both of which can be problematic for large domains.

For example, a variable with domain $x \in [0,99]$ can be represented using 100 bits, where the i-th bit corresponds to the value i. Applying the constraint $x \neq 10$ corresponds to setting the 10th bit to `false`. In contrast, enforcing $x = 10$ requires setting all other bits to `false`, resulting in a linear-time operation.

Interval representation The domain is represented as an interval by storing only its lower and upper bounds. This allows constant-time operations over the domain, since the domain is described by only two values (the bounds) regardless of its size. The downside is that non-contiguous domains (i.e., domains with "holes") cannot be represented. Although this compact representation may lead to a loss of information, it does not affect the completeness of the search procedure. Variables using this representation are referred to as *interval variables* and are commonly used in scheduling applications.

For example, a variable with domain $x \in [0,99]$ is represented by its lower and upper bounds. Applying the constraint $x \geq 50$ corresponds to updating the lower bound in constant

time. However, constraints such as $x \neq 10$ cannot be represented, resulting in a loss of information.

Other representations that mix these two ideas also exist. For example, augmenting an interval representation with a separate set capturing the holes, or storing a set of disjoint intervals. It is important to understand the underlying domain representation because, as we will see in later sections, certain propagation algorithms may only make sense with specific variable representations.

2.2 Solutions and Constraints

During search, algorithms manipulate the domains of variables. When each domain has been reduced to a single value, we obtain a *solution* (or *assignment*).

Formally, given a set of variables $\mathcal{X}$ with domains D, a **solution** is a mapping ϕ that assigns to each variable a value from its domain, i.e.,

$$\forall x \in \mathcal{X}, \qquad \phi(x) \in D(x).$$

We may represent a solution ϕ as a tuple $(\phi(x_1), \phi(x_2), \ldots, \phi(x_n))$, where the i-th entry corresponds to the value assigned to the i-th variable. Then the set of all possible solutions, denoted by $Sols(D)$, is the Cartesian product of the domains of the variables $X = \{x_1, \ldots, x_n\}$:

$$\text{Sols}(D) = \prod_{x \in X} D(x) = D(x_1) \times D(x_2) \times \cdots \times D(x_n) = \{(v_1, \ldots, v_n) \mid v_i \in D(x_i)\}.$$

We use *constraints* to restrict the set of all possible feasible solutions to only those solutions that are of interest. For example, the requirement "no two neighbouring nodes can share the same colour" is represented by a constraint $x_i \neq x_j$ for every edge (x_i, x_j).

Formally, a **constraint** $c : \Phi \to \{\top, \bot\}$ is a predicate that maps each solution $\phi \in \Phi$ to either *feasible* or *infeasible*, denoted by $\top$ and $\bot$, respectively. We write

$$\phi \models c$$

to denote that the solution ϕ satisfies the constraint c, i.e., $c(\phi) = \top$. Similarly, we write

$$\phi \not\models c$$

to denote that solution ϕ does not satisfy c, i.e., $c(\phi) = \bot$. This is also read as "solution ϕ violates constraint c".

We extend this notation to sets of constraints in a natural way. For a set of constraints C, we write

$$\phi \models C$$

to denote that the solution ϕ satisfies every constraint in the set, i.e.,

$$\forall c \in C : \phi \models c.$$

Similarly,

$$\phi \not\models C$$

denotes that solution ϕ does not satisfy at least one constraint in the set, i.e.,

$$\exists c \in C : \phi \not\models c.$$

The **scope** of a constraint is the set of variables relevant for the constraint. For example, in graph colouring, each constraint involves two variables corresponding to adjacent nodes. When referring to solutions of a particular constraint, we implicitly consider only the variables in its scope, even if there may be other variables and constraints in the problem. For example, for a constraint $c : x \neq y$, we have $\text{scope}(c) = \{x, y\}$.

We may also treat a constraint as a subset of all possible solutions. Given domains D, we use the notation $\text{c}(D)$ to denote the set of all feasible solutions of constraint c with respect to D. By definition,

$$\text{c}(D) \subseteq \text{Sols}(D).$$

For example, consider the constraint $c : x \neq y$ with domains

$$D(x) = \{0, 1\}, \qquad D(y) = \{0, 1, 2\}.$$

The set of feasible solutions is

$$\text{c}(D) = \{(0,1), (0,2), (1,0), (1,2)\}.$$

For illustrative purposes, we have explicitly enumerated the feasible solutions. In practice, they would typically be described implicitly, for example:

$$\text{c}(D) = \{(i, j) \mid i \in D(x),\ j \in D(y),\ i \neq j\}.$$

The definition of a constraint is intentionally abstract: it does not specify how feasibility is evaluated. The general definition merely characterises the constraint as a predicate or, equivalently, as a subset of solutions. The concrete feasibility conditions depend on the particular constraint under consideration and significantly vary from one constraint to another. When we introduce specific constraints later, we will make explicit the criteria used to determine their feasibility.

The term **global constraint** is used in the literature to distinguish between *primitive constraints*, which operate on a fixed number of variables such as $x = y$ (only two variables), and more *complex constraints*, whose scope may involve a variable number of variables, such as $\sum_{i \in I} w_i x_i \leq k$ where I is a set without predetermined size. In this book, however, we do not adopt this distinction and instead use the term *constraint* uniformly.

2.3 Constraint Satisfaction and Optimisation Problems

When solving a problem, we consider a set of constraints and seek a solution that satisfies all constraints, or show that no such solution exists. We may also introduce an objective function to discriminate between feasible solutions, and then aim to find the best solution according to the given objective function. The collection of variables, domains, and constraints is referred to as a *constraint satisfaction problem*, or a *constraint optimisation problem* when we add an objective function.

Formally, a **constraint satisfaction problem** (abbreviated as "CSP") is a tuple (X, D, C), where $X = (x_1, x_2, \ldots, x_n)$ is the set of variables with the domains D, and C is the set of constraints.

A CSP is *feasible* (or *satisfiable*) iff it admits at least one feasible solution:

$$\exists \phi : \ \phi \models C.$$

Otherwise, the CSP is *infeasible* (or *unsatisfiable*) as there does not exist a solution that satisfies all constraints:

$$\forall \phi : \phi \not\models C.$$

Deciding the feasibility of a CSP is an NP-complete problem, directly connecting it to the central open question in theoretical computer science: "Is $P = NP$?". While no polynomial-time algorithm is known for solving CSPs in general, many practical instances can nevertheless be handled efficiently. This is because real-world problems exhibit additional structure that can be exploited to reduce the search space to a manageable size. Understanding and leveraging such structure lies at the core of constraint programming.

2.3.1 Model vs. Instance

It is important to distinguish between the *model* of a problem and an *instance* of that problem.

The **model** provides the abstract formulation: it defines the variables, the domains, the constraints, and provides the semantics for the problem. In other words, the model describes the problem *in full generality*, without referring to any concrete case.

For example, graph colouring may be modelled by introducing n variables representing the vertices of the graph, each with a domain of k colours, and by adding a constraint for every edge stating that the two vertices associated with that edge cannot simultaneously be assigned the same colour. Notice that we did not specify the exact number of vertices or which edges exist; instead, we provided a general formulation based on an abstract graph.

A single problem may have different models. For example, an alternative model for graph colouring is to introduce binary variables $x_{i,j}$ with the meaning "vertex i is assigned colour j", together with constraints ensuring that each vertex receives exactly one colour and that adjacent vertices do not share a colour.

In both cases, the models act as an abstract template for the same problem. For a given problem, there may be many different models, some of which are better suited than others for solving it.

An **instance** is a concrete realisation of this model. It supplies all specific details required to solve a particular case: the actual variables that appear in the instance, their precise domains, and the explicit list of constraints that must be satisfied. An instance, therefore, fills in all parameters and data that the model leaves unspecified.

In our graph colouring example, the model describes the general problem of assigning colours to vertices so that adjacent nodes receive different colours, whereas an instance corresponds to a specific graph: it specifies the exact set of vertices, the concrete edges between them, and the set of available colours.

The model tells us *how* graph colouring is represented; the instance tells us *which* concrete graph to colour.

2.3.2 Optimisation

In some applications, we are not merely interested in any feasible solution but in determining the best feasible solution according to a given criterion. Problems of this type are known as *constraint optimisation problems*.

Formally, a **constraint optimisation problem** (COP) extends a CSP by introducing an objective function f. The function $f : \phi \to \mathbb{Z}$ assigns a numerical value to each solution, representing its quality. Feasible solutions are then compared according to this objective: in maximisation problems, higher values are preferred, whereas in minimisation problems, lower values are preferred.

For a maximisation COP (and analogously for minimisation COPs), a solution ϕ^* is considered *optimal* if it is feasible and no other feasible solutions achieve a strictly higher objective value:

$$\forall \phi : \qquad \phi \models C \implies f(\phi*) \geq f(\phi).$$

In general, the optimal assignment need not be unique; multiple distinct solutions may achieve the best objective value.

Note that the expression *"more optimal"* is not well defined: optimality is a binary property: an assignment is either optimal or not.

We may still compare solutions by saying that one solution is *better* than another. In this case, we mean that the objective value of one solution is strictly better than the objective value of the other. This notion is particularly important in settings where finding an optimal solution is impractical, and we settle for a *good-enough* solution instead.

Although optimisation problems are important, we will primarily focus on feasibility problems. The reason is that many optimisation techniques rely on repeatedly solving constraint satisfaction problems. For example, after obtaining a feasible solution, one may add a constraint requiring an improved solution, resolve the problem, and iterate this process until no better solution exists. Consequently, much of our attention will be devoted to feasibility.

3 Backtracking Search

Solving a CSP means either finding a feasible assignment or concluding that no solution satisfying all constraints exists.

We can solve any given CSP $= (X, D, C)$ by exhaustively exploring the search space, i.e., the set of all possible solutions. Using reasoning mechanisms such as propagation and conflict analysis allows us to avoid explicitly enumerating each solution. Instead, we can implicitly discard parts of the search space that cannot appear in any feasible solution, which is what makes search practical.

Our search procedure operates on the domains, starting with the initial domains given in the CSP. During search, the algorithm modifies the domains through a combination of *decisions* (predicates representing assumptions made by the algorithm), *propagation* (the consequences implied by those choices), and backtracking (reverting decisions that have been discovered to be infeasible).

For simplicity, in the remainder of this book, we refer to the search procedure, including all its components, as the *solver*.

In practice, the solver maintains a stack, known as the **trail**, that records all explicit modifications to the domains. It maintains a distinction between *decisions*, *propagations*, and their associated *decision levels*, as well as the *current domains* induced by these effects. As we will discuss in Chapter 4, the solver also stores derived constraints and additional data structures that improve the efficiency of propagation and decision making. These components form the *state* of the solver.

A **conflict** occurs whenever the current domains violate a constraint or, equivalently, when the domain of at least one variable becomes empty as a result of propagation. Conflicts trigger backtracking, allowing the search to reconsider earlier decisions.

Generalising the illustrative example from the previous section, we may summarise the structure of backtracking **search** as follows.

1. Apply **propagation** until no further inferences can be made. This is the main topic of Chapter 2.
2. If a **conflict** arises (i.e., a domain becomes empty as a result of propagation):
   - at decision level 0 (no decisions have been made), declare the CSP *infeasible*. Chapter 5 discusses how to provide a certificate verifying the infeasibility of the instance.
   - otherwise, **backtrack** by undoing the most recent decision and all its corresponding propagations, record the negation of that decision as a new propagation, and continue the search. In Chapter 4, we will study **conflict analysis** techniques that may allow backtracking more than one decision level.
3. If all domains are singleton, a *feasible* solution has been found.
4. Otherwise, make a new **decision**, which is discussed in Chapter 3.

The above is a depth-first search procedure, which is the standard approach in constraint programming. Because it exhaustively explores the search space, it is a **complete** search method: it is *guaranteed* to either find a feasible assignment or conclude that none exists.

In the worst case, backtracking search requires exponential time. Nevertheless, with the techniques developed in this book, many practically significant problems can be solved efficiently.

Other search paradigms exist, such as A* and best-first search, although these are less common in constraint programming than the depth-first search approach considered in this book. Nevertheless, many of the concepts developed here apply directly to those settings.

We also omit various forms of *incomplete* search, including local search and metaheuristics. In practice, however, complete solvers are used as subroutines within such frameworks, meaning that many of the techniques studied in this book remain relevant there as well.

In the next chapters, we will study each component in more detail.

Chapter 2

Constraints and Propagation

After studying this chapter, you will be able to...

- describe how propagators operate, with a focus on several widely used propagation algorithms;
- construct explanations and checkers to justify propagations and verify their correctness, providing a principled safety net against implementation errors;
- analyse and compare the effectiveness of propagators against their decompositions.

Constraints and propagators are central concepts in constraint programming. A constraint specifies which solutions are feasible, whereas propagators reduce the search space by removing values that cannot participate in any feasible solution according to constraints. The practical effectiveness of constraint programming relies heavily on propagators.

Constraints capture a specific problem structure. In this sense, each constraint can be viewed as a subproblem, and solving a CSP corresponds to finding a solution that simultaneously satisfies all of these subproblems. There are essentially no restrictions on the form a constraint may take, provided that its structure can be exploited by propagators. This flexibility is one of the key strengths of constraint programming.

For example, the *AllDifferent* constraint enforces that no two integer variables may be assigned the same value. The *Circuit* constraint specifies that variables representing a path in a graph must form a Hamiltonian cycle. In both cases, the constraints simplify modelling the larger problem. Crucially, by explicitly representing the structure through high-level constraints, we may employ specialised reasoning to solve the problem more efficiently. The combination of high-level constraints and dedicated algorithms that exploit their structure is one of the defining characteristics of constraint programming.

The Global Constraint Catalogue lists over 400 constraints [**?**][1], many of which have multiple propagation algorithms that trade off pruning strength against computational cost. In this chapter, we study several of the most widely used constraints and their associated propagation algorithms.

[1] `https://sofdem.github.io/gccat/`

Beyond classical propagation algorithms, we also discuss two concepts that have become increasingly important in modern solver design.

First, we associate each propagation with an *explanation*: a propositional statement describing the conditions that caused the propagation. Explanations play a central role in conflict analysis, which is the subject of Chapter 4, where we will also see how recent developments extend explanations beyond propositional reasoning.

Second, we adopt a proof-theoretic perspective on explanations, viewing them as certificates that explicitly state the reasoning used by the propagator. This viewpoint naturally leads to the notion of *checkers*: small, independent pieces of code that verify the correctness of explanations. Because checkers are intentionally simple, they can often be trusted more than the complex propagators whose behaviour they certify, offering a principled mechanism for detecting incorrect reasoning.

Checkers are used extensively in the constraint solver Pumpkin to increase confidence in the correctness of propagators. Related ideas form the foundation of Chapter 5, where we develop certificates for constraint satisfaction and optimisation problems. These certificates provide an independent means of verifying solver conclusions, namely infeasibility and optimality claims.

We also discuss an alternative to dedicated propagators: *decomposing* high-level constraints into simpler ones such as linear inequalities. While decomposition may not capture the full reasoning power of a specialised propagator, conflict analysis techniques (Chapter 4) can sometimes compensate for this loss and, in certain cases, even provide advantages. The extent to which decomposition can rival specialised propagators remains an active topic of investigation. We will see examples illustrating both the strengths and limitations of this approach.

To ground these ideas, we begin with a concrete example of a single linear inequality. Through this example, we illustrate how propagation is performed, how explanations justify individual propagations, and how a checker certifies those explanations. We then turn to the formal foundations, defining propagators, explanations, and checkers, before studying in detail a variety of constraints and their associated propagation algorithms.

4 Illustrative Example

We illustrate the main concepts of propagation, explanations (including lifted explanations), and verification using the running example:

$$x + 2y + 4z \geq 10,$$

with initial domains

$$x, y, z \in [0, 10].$$

We introduce the concepts informally in the example. After building the intuition, we present the complete formalisation in the next section.

Propagation The goal is to restrict the domains as much as possible, based on the constraint, without eliminating any feasible solutions. This reduces the search space and may

enable further propagation by other constraints. Removing values from domains is called *propagation*, and it is performed by algorithms we refer to as *propagators*.

In the initial state, every value in each domain participates in at least one feasible solution of the linear inequality. Consequently, no domain can be reduced. This situation, where no further propagation is possible, is referred to as a *fixpoint*.

Suppose the solver later updates the domains of variables x and z, possibly as a result of other propagators:

$$x \in [0,2], \quad y \in [0,10], \quad z \in [0,1].$$

Given these changes, the constraint now allows further propagation. In particular, the domain of y can be tightened: no value smaller than 2 can appear in any feasible solution. In other words, the value of y must be at least 2 in every feasible solution. We therefore *propagate* the atomic constraint $y \geq 2$ and update the domains accordingly:

$$x \in [0,2], \quad y \in [2,10], \quad z \in [0,1].$$

We again arrive at a fixpoint, meaning that no further propagation is possible.

Explanations We may be interested in understanding *why* a particular propagation took place. Providing an explanation helps us verify the correctness of the propagation and, as we will see in later chapters, allows us to derive additional propagation rules by analysing how multiple explanations interact in Chapter 4.

Let us examine several candidate explanations for the propagation $y \geq 2$:

$$\langle x \leq 6 \rangle \wedge \langle z \leq 0 \rangle \implies \langle y \geq 2 \rangle,$$

$$\langle x \leq 2 \rangle \wedge \langle z \leq 1 \rangle \implies \langle y \geq 3 \rangle,$$

$$\langle x \leq 2 \rangle \wedge \langle z \leq 1 \rangle \implies \langle y \geq 2 \rangle.$$

The first explanation is entailed by the linear inequality, so we say it is a *valid* explanation, but it is not *applicable* at the time of propagation. This is because the left-hand side contains the predicate $\langle z \leq 0 \rangle$, yet the current domain is $z \in [0,1]$, which does not satisfy this predicate. Consequently, this explanation cannot justify the propagation. If the solver had produced it for our particular propagation, this would indicate a bug.

The second explanation is applicable, since its left-hand side is satisfied by the current domains. However, it is not a logical consequence of the constraint; that is, it is not a *valid* explanation. Although the stronger propagation $y \geq 3$ would imply $y \geq 2$, the inequality admits feasible solutions with $y = 2$, showing that the explanation is not valid. A verification procedure (described below) can automatically detect this error.

The third explanation accurately reflects the reasoning used by the solver. We say it is a *correct* explanation since it is both *valid* and *applicable*. After verifying this explanation, we have much higher confidence in the correctness of the propagation.

Verification of Explanations Given an explanation, the task of verification is to assert that the explanation is *correct*, that is, logically implied by the constraint given the current domains. Verification is crucial for establishing the correctness of the solver.

Since it is straightforward to verify that an explanation is applicable, i.e., that its left-hand side holds for the current domains, we focus our attention on verifying the validity of the explanation, i.e., verifying that the explanation is logically implied by our particular linear inequality.

For convenience, we reproduce both the constraint and the explanation:

$$x + 2y + 4z \geq 10, \qquad \langle x \leq 2 \rangle \wedge \langle z \leq 1 \rangle \implies \langle y \geq 2 \rangle .$$

It is generally easier to verify a conflict explanation than a propagation explanation. To simplify the verification procedure, we therefore convert the propagation explanation into an equivalent conflict explanation by moving the right-hand side to the left-hand side:

$$\langle x \leq 2 \rangle \wedge \langle y \leq 1 \rangle \wedge \langle z \leq 1 \rangle \implies \bot.$$

The rewritten explanation states that, when the variables are restricted to these bounds, the linear inequality becomes infeasible. This form is equivalent to the original explanation.

To verify this claim, we evaluate the inequality under the bounds specified by the explanation. Observe that assigning larger values to x, y, and z increases the left-hand side of the inequality, intuitively, leading the constraint closer to feasibility. Therefore, the "best case" for satisfying the inequality is to assign each variable the largest value permitted by the explanation. If this optimistic assignment still violates the constraint, then every other assignment consistent with the explanation must also be infeasible, since all of them produce a strictly smaller left-hand side. This establishes that the explanation correctly describes an infeasible state.

Conversely, if the optimistic assignment satisfies the inequality, we obtain a concrete feasible solution under the bounds of the explanation. This contradicts the claim of the explanation that no feasible solution exists, allowing us to conclude that the explanation is incorrect.

Applying the procedure to our linear inequality

$$x + 2y + 4z \geq 10$$

and the explanation

$$\langle x \leq 2 \rangle \wedge \langle y \leq 1 \rangle \wedge \langle z \leq 1 \rangle \implies \bot,$$

the optimistic solution dictated by the explanation gives

$$1 \cdot 2 + 2 \cdot 1 + 4 \cdot 1 \geq 10,$$

which simplifies to

$$8 \geq 10.$$

The inequality is violated, confirming that the explanation correctly describes an infeasible state. This completes the verification.

We now illustrate the same procedure using an incorrect explanation. Recall the faulty explanation:

$$\langle x \leq 2 \rangle \wedge \langle z \leq 1 \rangle \implies \langle y \geq 3 \rangle .$$

Rewriting it results in the following explanation

$$\langle x \leq 2 \rangle \wedge \langle y \leq 2 \rangle \wedge \langle z \leq 1 \rangle \implies \bot.$$

Evaluating the inequality under these bounds gives

$$1 \cdot 2 + 2 \cdot 2 + 4 \cdot 1 \geq 10,$$

which simplifies to

$$10 \geq 10.$$

Since the constraint can be satisfied, it is clear that the explanation incorrectly claims infeasibility. A verification procedure would therefore identify this explanation as invalid.

Now that we have introduced the ideas behind propagation, explanations, and verification, we can turn to a more subtle aspect of explanations.

Direct Redundancies in Explanations In addition to requiring that explanations are correct, we aim to avoid immediate redundancies. Such redundancies are undesirable: they make explanations less faithful to the actual propagation algorithm, complicate verification, and reduce their usefulness in later tasks such as conflict analysis (Chapter 4).

Consider the following explanations:

$$\langle x \neq -1 \rangle \wedge \langle x \leq 2 \rangle \wedge \langle z \geq 0 \rangle \wedge \langle z \leq 1 \rangle \implies \langle y \geq 2 \rangle ,$$

$$\langle x \leq 2 \rangle \wedge \langle y \neq 2 \rangle \wedge \langle z \leq 1 \rangle \implies \langle y \geq 3 \rangle .$$

Both explanations are formally valid, but they contain redundancies. In the first, the predicates involving holes and lower bounds (highlighted in blue) do not contribute to the reasoning. In the second, the explanation reasons explicitly about how holes in the domain of y influence its lower bound—logic that is handled by the solver, not by this propagator.

We now focus on refining explanations, which may be seen as removing implicit redundancies.

Lifted explanations Ideally, explanations should capture reasoning that extends beyond the specific situation in which they are generated. In other words, we prefer explanations that are as general as possible. Such generality improves our understanding of the propagation algorithm, although the main purpose of lifting will become clear in later chapters when we discuss conflict analysis, which combines multiple explanations to derive new, stronger propagation rules. It is therefore useful to become familiar with the notion of *lifted explanations*.

Recall our constraint and current domains:

$$x + 2y + 4z \geq 10, \qquad x \in [0,2],\ y \in [0,10],\ z \in [0,1].$$

Consider the following two explanations for the propagation $y \geq 2$:

$$\langle x \leq 2 \rangle \wedge \langle z \leq 1 \rangle \implies \langle y \geq 2 \rangle,$$

$$\langle x \leq 3 \rangle \wedge \langle z \leq 1 \rangle \implies \langle y \geq 2 \rangle.$$

Both explanations are applicable and valid. They differ only in the bound placed on x: the second explanation uses a weaker bound. Since it covers strictly more situations while remaining correct, the second explanation is *more general*. We therefore refer to it as a *lifted* version of the first explanation.

Lifting is not always unique. For example, suppose the domains are

$$x \in [0,1], \quad y \in [0,1], \quad z \in [0,1],$$

and consider again the inequality $x + 2y + 4z \geq 10$. The constraint is infeasible, and a straightforward conflict explanation is

$$\langle x \leq 1 \rangle \wedge \langle y \leq 1 \rangle \wedge \langle z \leq 1 \rangle \implies \bot.$$

We may attempt to lift this explanation by increasing the bounds while preserving infeasibility. Increasing the bound on z immediately breaks validity, since larger values of z may satisfy the constraint. However, lifting the bounds on the variables x or on y remains possible, resulting in the following explanations:

$$\langle x \leq 2 \rangle \wedge \langle y \leq 1 \rangle \wedge \langle z \leq 1 \rangle \implies \bot,$$

$$\langle x \leq 3 \rangle \wedge \langle y \leq 1 \rangle \wedge \langle z \leq 1 \rangle \implies \bot,$$

$$\langle x \leq 1 \rangle \wedge \langle y \leq 2 \rangle \wedge \langle z \leq 1 \rangle \implies \bot.$$

All three explanations are more general than the original explanation above. Moreover, the second explanation is a lifted version of the first. In contrast, the second and third explanations are *incomparable*: neither is a lifted version of the other. In this case, we may choose either explanation, as determining principled criteria for selecting among incomparable explanations remains an open research question. The motivation for such criteria will become clearer once we discuss conflict analysis in Chapter 4. For the purposes of this chapter, any explanation that can be verified by a checker may be selected.

5 Main Concepts

Now that we have, in the illustrative example, introduced the core concepts of propagation, fixpoints, explanations (including lifted explanations), and verification, we turn to formalising these concepts in a more general manner, forming the foundations for all propagators.

5.1 Propagators

Recall that a constraint merely classifies full assignments as either `feasible` or `infeasible`. During search, however, it is essential to remove (prune) values from the domains of variables without compromising the overall feasibility of the problem. This reduction of domains is carried out by *propagators*.

A **propagator** is a function *associated* with a constraint that *contracts* the domains of variables by pruning values that cannot appear in any feasible solution.

Formally, a propagator p associated with constraint c is a mapping from domains to domains

$$p : D \to D$$

such that the domains are nonincreasing

$$\forall x_i \in \text{scope}(c) : p(D)(x_i) \subseteq D(x_i).$$

whilst preserving the set of feasible solutions before and after propagation

$$c(p(D)) = c(D).$$

Values removed from domains are said to be *pruned by propagation*. Propagators represent pruning by enforcing atomic constraints. For example, removing value 3 from the domain of variable x corresponds to the atomic constraint $\langle x \neq 3 \rangle$, whereas removing all values greater than 5 corresponds to the atomic constraint $\langle x \leq 4 \rangle$. In the literature, propagation algorithms are also referred to as *propagation rules* or *filtering rules*.

It is crucial to distinguish between a *constraint*, which defines feasibility, and a *propagator*, which performs pruning based on a constraint.

There is no uniform pattern for how propagators are constructed; instead, this depends heavily on the constraint. We illustrate this with two simple examples.

For the constraint $x \neq y$, there is only one straightforward propagator: as soon as either variable x or y becomes assigned to a value v, the value v is pruned from the domain of the other variable. We write this rule as $x = v \implies y \neq v$ and $y = v \implies x \neq v$.

For the constraint $2x + y \leq 10$, a natural approach is to reason over bounds:

$$x \leq \left\lfloor \frac{10 - \text{LB}(y)}{2} \right\rfloor \qquad \text{and} \qquad y \leq 10 - 2\text{LB}(x).$$

In both cases, understanding the possible propagation rules required reasoning about the specific structure of the constraint. These particular constraints leave little room for design choices and rely primarily on arithmetic reasoning. Later in this chapter, however, we will study constraints over a much wider range of structures, namely scheduling tasks and graphs. We will also see that different propagation algorithms involve different trade-offs between pruning strength and computational cost for the same constraint. In some cases, one propagation algorithm may discover inferences that another does not, and vice versa.

5.2 Propagation properties

Since propagators can offer different trade-offs between inference strength and computational cost, we require notions that characterise their behaviour and allow us to compare them. We begin with the notion of a fixpoint and then introduce three key propagation properties: idempotence, domain consistency, and bound consistency.

Fixpoint We say that a propagator p is at a **fixpoint** for a domain D if applying the propagator does not further reduce the domain. Formally, this may be represented as

$$p(D) = D.$$

When the propagator is clear from the context, we may simply refer to the domain D as a fixpoint.

The notions discussed below relate propagator behaviour with respect to the fixpoint.

Idempotence A propagator is said to be **idempotent** if it is guaranteed to reach a fixpoint after a single application.

Most propagators used in practice, though not all, eventually reach a *unique* fixpoint after repeated applications. In principle, we could construct idempotent propagators by wrapping a given propagator and repeatedly applying it until a fixpoint is reached, effectively transforming a non-idempotent propagator into an idempotent one.

However, enforcing propagation to a fixpoint is not always desirable in practice. Since reaching a fixpoint can be computationally expensive, it may be preferable to apply simpler propagators first before reapplying more expensive propagators.

Nevertheless, at each decision level, solvers repeatedly invoke propagators until a global fixpoint is reached. Although this final fixpoint is unique, the order in which propagators are applied can have a significant impact on the computational effort required to reach it [**?**].

Having considered fixpoints and idempotence, we now turn to propagation strength. Two fundamental notions are **domain consistency** and **bound consistency**.

Domain consistency Intuitively, a propagator is *domain consistent* if, at every fixpoint, each value in the domain participates in at least one feasible solution. In other words, there is no possibility of further reducing the domain without removing feasible solutions. Domain consistency is the strongest notion of propagation strength.

For example, consider the equality constraint $x = y$ with the propagation rules

$$x = v \implies y = v, \qquad y = v \implies x = v.$$

These rules are sound, in the sense that they preserve the feasible solutions of the equality constraint: whenever either variable is assigned a value, the other variable is forced to take the same value; otherwise a conflict occurs. However, the rules do not enforce domain consistency. Indeed, if $x \in [0,5]$ and $y \in [3,10]$, then the values 0, 1, and 2 in the domain of x cannot participate in any feasible solution, yet they are not removed.

To obtain a domain-consistent propagator, we must reason directly about the intersection of the domains:

$$D(x) \leftarrow D(x) \cap D(y), \qquad D(y) \leftarrow D(x) \cap D(y).$$

Applying these rules to the example above results in the domains

$$D(x) = D(y) = [3,5].$$

These rules are also idempotent, as a single application is sufficient to reach a fixpoint. At this point, every remaining value participates in a feasible solution.

Formally, let c be a constraint with $\text{scope}(c) = \{x_1, \ldots, x_n\}$ and let p be a propagator associated with c. Recall that $p(D)$ denotes the domain obtained after application of propagator p, $c(D)$ denotes the set of solutions of constraint c with the domain D, and $p(D) = D$ means domain D is at a fixpoint for propagator p. We say that propagator p enforces **domain consistency** (also called *arc consistency*) if

$$\forall D_{fp}\big(p(D_{fp}) = D_{fp}\big), \forall x_i \in \text{scope}(c), \forall v \in D_{fp}(x_i) :$$

$$\exists \theta \in c(D_{fp}) \quad \text{s.t.} \quad \theta(x_i) = v.$$

Bound consistency The other notion of consistency considered here is *bound consistency*. This is a weaker notion that focuses only on the lower and upper bounds of the domains, rather than all individual values.

Bound consistency requires that, at every fixpoint, if each domain is viewed as an interval defined by its lower and upper bounds (i.e., ignoring any holes), then each bound participates in at least one feasible solution.

To define bound consistency formally, we first introduce the interval overapproximation operator

$$\square : D \to D,$$

which replaces each domain with the interval defined by its lower and upper bounds, effectively filling any holes in the domain. Formally,

$$\square(D)(x) = [\text{LB}(x), \text{UB}(x)].$$

We may now define bound consistency for a propagator p with respect to a constraint c with $\text{scope}(c) = \{x_1, \ldots, x_n\}$. We say that propagator p enforces **bound consistency** if

$$\forall D_{fp}\big(p(D_{fp}) = D_{fp}\big), \forall x_i \in \text{scope}(c), \forall b \in \{\text{LB}(x_i), \text{UB}(x_i)\} :$$

$$\exists \theta \in c(\square(D_{fp})) \quad \text{s.t.} \quad \theta(x_i) = b.$$

Bound and domain consistency, as presented here, are the most widely used notions. Although many other consistency notions have been proposed in the literature, they are not required for the techniques studied in this book, and we do not discuss them further.

We will see examples of both notions, as well as propagators that are neither bound nor domain-consistent, because achieving such consistency is **NP**-complete under their constraints.

Propagation Trade-offs Idempotence, domain consistency, and bound consistency describe the propagation strength and behaviour of propagators. Another important consideration is the computational effort required to enforce them.

In practice, the strongest propagator is not necessarily the most effective. The ultimate objective is not to maximise propagation strength, but to minimise the overall time required to solve the problem. Much of propagator design can therefore be viewed as balancing the computational cost of propagation against its pruning strength. For example, an expensive propagator may provide only marginally more pruning than a simpler alternative, making its additional cost difficult to justify. Conversely, a very inexpensive propagator may perform so little pruning that it has little impact on the search.

In Chapter 4, we will also see that the generality of the explanations provided by a propagator plays an important role when analysing conflicts.

Propagator design is therefore a multi-objective problem involving propagation strength, computational effort, and explanation generality. The relative importance of these factors is not yet fully understood and may depend on the problem being solved. A propagator may be highly beneficial when its associated constraint is central to the structure of the problem, yet offer little advantage when that constraint is only weakly constraining relative to the other constraints in the model.

5.3 Explanations

Propagators are required to provide *correct explanations* for the propagations they perform and for any conflicts they detect.

Explanations serve two main purposes. First, they allow us to verify the correctness of the propagator. Second, as we will see in a later chapter, explanations form the basis of conflict analysis, where they are combined to derive stronger propagation rules.

An **explanation** is a propositional statement that specifies the logical reason why a propagation or conflict occurred.

Explanations are constructed using **atomic constraints**, which are elementary predicates involving a single variable. Each atomic constraint has the form

$$\langle x \bowtie c \rangle,$$

where x is an integer variable, $c \in \mathbb{N}$ is an integer constant, and $\bowtie \in \{=, \neq, \geq, \leq\}$ is a comparison operator.

Explanations are built by combining atomic constraints a_i in one of two principal ways:

1. A **propagation explanation** has the form

   $$a_1 \wedge a_2 \wedge \cdots \wedge a_{n-1} \implies a_n,$$

   where a_n is the *propagated atomic constraint*.

2. A **conflict explanation** has the form

   $$a_1 \wedge a_2 \wedge \cdots \wedge a_n \implies \bot,$$

   where $\bot$ denotes contradiction.

We refer to the left-hand side of an explanation as the **reason** (or *premise*). The reason specifies why the right-hand side (the *consequent*) must hold.

Although propagation and conflict explanations have different syntactic forms, they are interchangeable: one can always be transformed into the other. A conflict explanation can be viewed as a special case of a propagation explanation in which the propagated atomic constraint is the contradiction $\bot$. However, it is convenient to keep the two forms distinct because they play different roles during search.

Correctness of explanations

Explanations must satisfy two key properties to be considered *correct*.

Validity. An explanation is **valid** if it is logically entailed by the constraint. That is, every feasible assignment of the constraint must satisfy the explanation. Formally, for a constraint C, an explanation E is correct if

$$\forall \theta \in Sol(C): \quad \theta \models E.$$

Note that even though propagators provide explanations, correctness is defined with respect to the constraint.

Applicability. An explanation is applicable if it can be used as a justification *given the state of the search*. Note that even though an explanation may be valid, it might not be relevant for the current trail. To make this precise, we introduce an ordering $\preceq_T$ over atomic constraints with respect to the *trail* T, representing the order in which the atomic constraints became true during search.

If an atomic constraint a was assigned before an atomic constraint b, we write $a \preceq_T b$. Any assigned atomic constraint precedes any unassigned atomic constraint. During a conflict, every assigned atomic constraint is considered to precede the contradiction $\bot$. When an atomic constraint a is placed on the trail, any atomic constraints it implies are considered to occur *after* a, ordered according to their natural relationship—for example, tighter bounds are ordered before weaker bounds. Implied disequality atomic constraints, however, do not have a defined order among themselves.

Example 1. *If $\langle x_9 \leq 0\rangle$ is assigned at decision level 1 and $\langle x_2 \leq 3\rangle$ at level 3, then $\langle x_9 \leq 0\rangle \preceq_T \langle x_2 \leq 3\rangle$.*

Example 2. *Consider $x \in [0, 10]$. Placing $\langle x \geq 5\rangle$ on the trail implicitly satisfies $\langle x \geq 3\rangle$. We therefore have $\langle x \geq 5\rangle \preceq_T \langle x \geq 3\rangle$.*

We may now formally introduce applicability. Let $D(T)$ denote the domains induced by the current trail T. An explanation

$$a_1 \wedge \cdots \wedge a_{n-1} \implies a_n$$

is **applicable** with respect to T if:

$$D(T) \models a_1 \wedge \cdots \wedge a_{n-1},$$

and every a_i in the reason occurs before the right-hand side:

$$\forall a_i \in \mathit{reason}(E): \quad a_i \preceq_T a_n.$$

The definition naturally applies to conflict explanations as well, by treating the propagated atomic constraint as the contradiction $\bot$. However, when determining the applicability of a propagation explanation, we must check applicability on the propagation explanation itself and not convert it into a conflict explanation.

An explanation is **correct** if it is both valid and applicable; otherwise it is **incorrect**. Throughout this text, explanations are assumed to be correct unless stated otherwise.

Example 3. *Given a trail T containing $\langle x_1 \geq 2\rangle$ and $\langle x_2 \leq 0\rangle$, the constraint $-x_1 + x_2 + 2x_3 \geq 0$ propagates $\langle x_3 \geq 1\rangle$, with explanation*

$$\langle x_1 \geq 2\rangle \wedge \langle x_2 \leq 0\rangle \implies \langle x_3 \geq 1\rangle.$$

This explanation is valid and applicable, and therefore correct.

Example 4. *Consider the constraint $x_1 + x_2 + x_3 \neq 7$ with domains $x_1, x_2, x_3 \in \{1,2,3,4\}$. After domain reductions $x_1 = 1$ and $x_2 = 2$, we obtain the propagation $x_3 \neq 4$ with explanation*

$$[x_1 = 1] \wedge [x_2 = 2] \implies [x_3 \neq 4].$$

This explanation is correct. However, the explanation

$$[x_1 = 2] \wedge [x_2 = 1] \implies [x_3 \neq 4]$$

is not correct, because its left-hand side is not applicable.

As with conflict explanations, multiple explanations may exist for a single propagation. In such cases, any explanation satisfying the validity and applicability conditions is acceptable.

We also prefer explanations without redundancies; however, we do not formally define this concept and instead rely on intuition.

5.4 Checkers

Checkers increase our confidence in the correctness of a solver by independently verifying the explanations it produces. They are implemented separately from the algorithms that generate these explanations and are intentionally kept much simpler. This separation and simplicity are deliberate design choices: the simpler the checker, the easier it is to trust. In some cases, this simplicity even enables the use of *formal software verification* techniques to further strengthen assurance, although such techniques lie beyond the scope of this material.

Checkers are tailored to specific propagators and, ideally, determine definitively whether an explanation is correct or incorrect. In practice, however, achieving full completeness is difficult. Consequently, checkers often adopt an incomplete but practical approach: when a checker verifies an explanation, we can be reasonably confident in its correctness; when it fails to confirm it, the explanation may still be correct, but the verification procedure cannot

certify it. This limitation is acceptable because the checker does not need to validate all possible explanations—only those produced by a specific propagation method. This focus ensures that the verification procedure remains both simple and efficient.

Formally, let $\mathcal{E}$ be the set of all explanations. A **checker** V, which is tied to a specific propagation algorithm, is a mapping

$$V : \mathcal{E} \longrightarrow \{\text{valid}, \text{unknown}\}$$

that, for an explanation E, provides a one-sided guarantee of validity:

$$\forall E \in \mathcal{E}: \quad V(E) = \text{valid} \implies E \text{ is a valid explanation.}$$

5.5 Decomposition

An alternative to using propagators is to **decompose** a constraint into other, typically simpler, constraints—for example, a decomposition into linear inequalities. This is always possible and becomes necessary when the underlying solver does not support a dedicated propagator. Modelling tools such as MiniZinc automatically apply such decompositions when required.

The rule of thumb is that using a dedicated propagator is preferable to using a decomposition. A propagator may run faster than its decomposition or exploit additional problem structure to achieve stronger propagation. However, as more advanced techniques—particularly *conflict analysis*—are introduced into constraint programming, this conventional wisdom can be challenged. We postpone a detailed discussion of these trade-offs to a later chapter.

A decomposition may or may not replicate the reasoning performed by a propagator. When working with decompositions, the same notions of consistency defined for propagators can be applied to the resulting set of constraints.

One of the strengths of constraint programming is its wide range of constraints and propagation algorithms, allowing approaches to be tailored to specific problems. For example, the global-constraint catalogue includes more than 400 constraints, many with multiple propagation algorithms. In the following sections, we study some of the most common constraints and their associated propagation algorithms.

6 Linear Integer Inequalities

The running example in the previous section illustrated the essential ideas of propagation, explanation, and verification on a single concrete inequality.

We now extend these ideas to the general case of *linear integer inequalities*, one of the most commonly used constraints. Such a constraint requires that a weighted sum of variables must meet or exceed a given threshold. Throughout this section, we consider inequalities of the form

$$\sum_i a_i x_i \geq b,$$

where $a_i, b \in \mathbb{Z}$ are integer constants and x_i are integer variables. The case of $\leq$ will be analogous.

We begin with *conflict detection*, which determines whether the constraint is already violated under the current domains. We then derive *bound propagation* rules that tighten domains while preserving all feasible assignments. For each of these inferences, we also introduce the corresponding explanations and describe the verification procedures used to ensure their correctness. We conclude the section by illustrating the complete reasoning process through several worked examples.

6.1 Conflict detection

Intuition

To determine whether the inequality is already violated under the current variable domains, we consider the *most optimistic* assignment for satisfying the constraint: every variable is set to the value that maximises its contribution to the left-hand side. Consequently, a variable with a positive coefficient $a_i > 0$ contributes more as its value increases, so we use its upper bound; a variable with a negative coefficient $a_i < 0$ contributes more as its value decreases, so we use its lower bound.

If the left-hand side remains smaller than the right-hand side, even under this best-case scenario, then no feasible assignment exists, and the propagator declares a conflict.

Algorithm

To formalise the intuition above, we introduce the concept of the *optimistic contribution of a variable* x_k for a linear inequality. This represents the maximum contribution the variable can make to the left-hand side of the linear inequality to support feasibility in the best case. We compute it as

$$\text{optimistic\_contribution}(x_k) = \begin{cases} a_k UB(x_k), & a_k \geq 0, \\ a_k LB(x_k), & a_k < 0. \end{cases}$$

We may then introduce the **slack** of a linear inequality, computed as

$$slack = \sum_i optimistic\_contribution(x_i) - b.$$

Informally, the slack represents the amount by which the variables can deviate from their optimistic values before the constraint becomes infeasible.

We may now state the conflict detection procedure: *the linear inequality is infeasible if its slack is negative*, since even the most optimistic assignment fails to satisfy it. Formally, given the current domains $\mathcal{D}$ and the linear inequality C, there are no feasible assignments:

$$slack < 0 \iff \forall A \in \mathcal{A}(\mathcal{D}) : A \not\models C$$

A straightforward implementation recomputes the slack from scratch by scanning all variables whenever the propagator is invoked. Alternatively, the slack may be maintained incrementally: it is computed once at the root and updated whenever a relevant bound changes. While incrementality avoids redundant calculation, it requires careful bookkeeping to ensure correctness under backtracking.

Explanation

For a linear inequality evaluated under optimistic assignments, infeasibility arises because each variable is restricted to a bound that prevents the left-hand side from reaching the right-hand side threshold value. These bounds are the source of the conflict.

For variables with positive coefficients, the optimistic value is the upper bound; for variables with negative coefficients, it is the lower bound. The conflict explanation takes the form

$$\bigwedge_{a_i>0} \langle x_i \leq UB(x_i)\rangle \wedge \bigwedge_{a_i<0} \langle x_i \geq LB(x_i)\rangle \implies \bot.$$

Explanation lifting can be performed if weakening an individual bound still preserves the conflict. Multiple incomparable lifted explanations may exist.

Although lifting is beneficial, there is no universally accepted procedure for lifting explanations for linear inequalities.

Verification

To verify a conflict explanation, we treat the bounds appearing in the left-hand side of the explanation as the only restrictions on the variable domains and recompute the optimistic left-hand side of the linear inequality. As in the conflict detection algorithm, we assign each variable the value that maximises its contribution: upper bounds for positive coefficients and lower bounds for negative ones. If the resulting sum is strictly smaller than the threshold b, the explanation correctly describes an infeasible state.

Formally, let E be the explanation, and let $LB_E(x_i)$ and $UB_E(x_i)$ denote the bounds stipulated by E. We check whether

$$\sum_{a_i>0} a_i \cdot UB^E(x_i) + \sum_{a_i<0} a_i \cdot LB^E(x_i) < b$$

If so, the explanation is valid, meaning the explanation correctly describes a scenario in which the constraint is infeasible.

Although this procedure is similar to conflict detection, it remains a separate component of the solver: verification inspects only the bounds given in the explanation, whereas the propagator may rely on more intricate incremental logic. Its simplicity increases our confidence that incorrect explanations will be detected.

6.2 Bound Propagation

Intuition

Once we have established that the inequality is not conflicting under the current domains, we can reason about the bounds of each variable to ensure a feasible assignment. A variable with a positive coefficient $a_k > 0$ increases the left-hand side as its value increases, so feasibility may require raising its lower bound. Conversely, a variable with a negative coefficient $a_k < 0$ increases the left-hand side as its value decreases, so feasibility may require tightening its upper bound.

For each variable x_k, we assume that all other variables take their *optimistic* values—those that maximise their contribution to the left-hand side. This gives us enough information to then determine the lower bound (for variables with positive coefficients $a_k > 0$) or upper bound (for variables with negative coefficients $a_k < 0$) for x_k that still allows the inequality to be satisfied.

Algorithm

We now formalise bound propagation by quantifying how much each variable must contribute in order to compensate for any deficit in the remaining part of the constraint to maintain the prospect of feasibility.

We introduce a refinement of the slack that excludes the contribution of a specific variable:

$$slack\_without(x_k) = slack - optimistic\_contribution(x_k).$$

Since a conflict arises when the slack becomes negative ($slack < 0$), to ensure feasibility, we require

$$slack\_without(x_k) + a_k x_k \geq 0.$$

Rewriting the above, we obtain the expression that provides the corresponding inferences:

$$a_k x_k \geq -slack\_without(x_k).$$

We distinguish two cases depending on the sign of the coefficient a_k.

Positive coefficient ($a_k > 0$). A positive coefficient increases the left-hand side when x_k *increases*. To maintain feasibility, the contribution of the variable x_k must be at least $-slack\_without(a_k)$; any smaller value would lead to a violation of the constraint. This propagates the lower bound

$$x_k \geq \left\lceil \frac{-slack\_without(x_k)}{a_k} \right\rceil,$$

where rounding up accounts for the discreteness of the domain.

Negative coefficient ($a_k < 0$). A negative coefficient increases the left-hand side when x_k *decreases*. To maintain feasibility, the negative contribution of the variable x_k cannot exceed $slack\_without(a_k)$; any larger value results in infeasibility. The upper bound becomes

$$x_k \leq \left\lfloor \frac{-slack\_without(x_k)}{a_k} \right\rfloor,$$

where rounding down again reflects the discrete nature of the domains.

This reasoning is applied to each variable in the linear inequality. Since propagation leaves the slack unchanged, tightening the bound of one has no effect on the bounds of the other variables. Consequently, a single pass over all variables suffices to reach a fixed point.

Explanation

The new bound for x_k is derived by assuming that all other variables take their optimistic values—upper bounds for positive coefficients and lower bounds for negative coefficients. These optimistic bounds form the reason supporting the propagation on x_k.

For a lower bound propagation $x_k \geq m$ (with $a_k > 0$), the explanation is

$$\left(\bigwedge_{\substack{a_i>0 \\ i\neq k}} \langle x_i \leq UB(x_i) \rangle \wedge \bigwedge_{\substack{a_i<0 \\ i\neq k}} \langle x_i \geq LB(x_i) \rangle \right) \implies \langle x_k \geq m \rangle .$$

The upper-bound case $x_k \leq m$ for $a_k < 0$ is analogous.

As with conflict explanations, lifting is possible: weakening one of the bounds while preserving the propagation still yields a correct explanation. No universally accepted lifting procedure exists, and a widely accepted procedure has yet to be established.

Verification

We verify a propagation explanation by reducing it to an equivalent conflict explanation. If the current bounds justify $x_k \geq m$, then any assignment with $x_k \leq m-1$ must violate the inequality. This can be checked exactly as in the conflict case.

The benefit of this reduction is that it reduces verifying propagation explanations to the simpler task of verifying conflict explanations.

Formally, given a propagation explanation

$$\bigwedge_{\substack{a_i>0 \\ i\neq k}} \langle x_i \leq UB(x_i) \rangle \wedge \bigwedge_{\substack{a_i<0 \\ i\neq k}} \langle x_i \geq LB(x_i) \rangle \implies \langle x_k \geq m \rangle ,$$

we convert it into a conflict explanation by negating the right-hand side and moving it to the left-hand side

$$\bigwedge_{\substack{a_i>0 \\ i\neq k}} \langle x_i \leq UB(x_i) \rangle \wedge \bigwedge_{\substack{a_i<0 \\ i\neq k}} \langle x_i \geq LB(x_i) \rangle \wedge \langle x_k \leq m-1 \rangle \implies \bot .$$

We use an analogous procedure for the propagation $x_k \leq m$ and its negation $x_k \geq m+1$.

The two explanations above are equivalent, but the second is easier to verify. Consequently, we consider verification only for conflict explanations and use this reduction for propagation explanations.

6.3 Examples

The previous sections introduced all components of the linear inequality propagator: conflict detection, bound propagation, explanations, and verification. We now put these components together through a series of worked examples. Each example highlights a different aspect of the propagation process and illustrates how the general reasoning applies in concrete situations.

Example #1: Simple Propagation

Consider the inequality

$$x_1 + 2x_2 \geq 4,$$

with initial domains

$$x_1 \in \{0,1,2,3,4\}, \qquad x_2 \in \{0,2\}.$$

Propagation Inspecting x_1 gives no tightening: as long as $x_2 = 2$, the inequality can be satisfied for any value of x_1 in its domain. Similarly, inspecting x_2 yields no propagation because $x_1 = 4$ remains possible. The constraint is initially at a fixed point.

Assume now that another propagator removes the value 4 from the domain of x_1, giving

$$x_1 \in \{0,1,2,3\}, \qquad x_2 \in \{0,2\}.$$

This change has no impact on the propagation of the lower bound of variable x_1, since the value 2 still remains in the domain of variable x_2. However, this change does affect the optimistic assignment for variable x_1, which triggers propagation for variable x_2:

$$\begin{aligned} & x_1 + 2x_2 \geq 4 \\ & 3 + 2x_2 \geq 4 \quad \text{(applying the optimistic } \langle x_1 = 3 \rangle) \end{aligned}$$

Intuitively, we observe that variable x_2 must contribute at least 1 unit to the left-hand side, otherwise the constraint is surely violated. If the domain were continuous, setting $x_2 \geq 0.5$ would be sufficient; however, because the domain is discrete, this yields the propagation

$$x_2 \geq 1.$$

The above is precisely what the propagation algorithm would conclude. The slack without the variable x_2 is -1, indicating that the contribution of x_2 must be at least 1 to cover the deficit of -1:

$$2x_2 - 1 \geq 0.$$

This leads us to conclude that

$$x_2 \geq \left\lceil \frac{1}{2} \right\rceil = 1.$$

Explanation Propagation of $x_2 \geq 1$ is justified by the optimistic bound on x_1:

$$\langle x_1 \leq 3 \rangle \;\Rightarrow\; \langle x_2 \geq 1 \rangle.$$

Verification To verify the explanation, we first convert the propagation explanation into a conflict explanation

$$\langle x_1 \leq 3 \rangle \wedge \langle x_2 \leq 0 \rangle \implies \bot,$$

and then check whether the bounds in the explanation result in an infeasible constraint. Substituting the values into the equation confirms the correctness of the explanation:

$$\begin{aligned} x_1 + 2x_2 &\geq 4 \\ 3 + 2 \cdot 0 &\geq 4 \\ 3 &\geq 4 \\ \bot. &\quad \text{(explanation verified)} \end{aligned}$$

Had our explanation been incorrect, for example

$$\langle x_1 \leq 3 \rangle \wedge \langle x_2 \leq 1 \rangle \implies \bot,$$

our verification procedure would have computed

$$\begin{aligned} x_1 + 2x_2 &\geq 4 \\ 3 + 2 \cdot 1 &\geq 4 \\ 5 &\geq 4 \\ \top, &\quad \text{(verification not successful)} \end{aligned}$$

and the explanation would have been rejected.

Example #2: Multiple Propagations And Holes

Let us now consider a more involved example with the constraint

$$x_1 + 2x_2 + 3x_3 \geq 17,$$

with the initial domains

$$x_1 \in \{1,4\},\ x_2 \in \{0,2\},\ x_3 \in \{0,1,2,3,4\}.$$

We consider the variables in order. No propagation is done on the lower bound of the variable x_1, since regardless of its value, the other variables can be set in a way to satisfy the constraint: $x_2 = 2$ and $x_3 = 4$.

When considering variable x_2, the remaining variables may contribute at most 16 to the left-hand side, implying that the contribution of the variable x_2 must account for at least 1 unit to meet the right-hand side of 17. This leads to the propagation $x_2 \geq 1$ with the explanation

$$\langle x_1 \leq 4 \rangle \wedge \langle x_3 \leq 4 \rangle \implies \langle x_2 \geq 1 \rangle .$$

Note that the propagation sets $x_2 \geq 1$ rather than $x_2 \geq 2$. Even though the domain was $x_2 \in \{0,2\}$, the propagator can only justify $x_2 \geq 1$.

Lastly, using similar reasoning, we discover that setting the variable x_3 to any value smaller than 3 results in infeasibility, enforcing the propagation $x_3 \geq 3$ with the explanation

$$\langle x_1 \leq 4 \rangle \wedge \langle x_2 \leq 2 \rangle \implies \langle x_3 \geq 3 \rangle .$$

This concludes the propagation, which has now reached a fixed point with the domains

$$x_1 \in \{1,4\},\ x_2 \in \{2\},\ x_3 \in \{3,4\}.$$

We now verify the last explanation by converting it into a conflict explanation

$$\langle x_1 \leq 4\rangle \wedge \langle x_2 \leq 2\rangle \wedge \langle x_3 \leq 2\rangle \implies \bot,$$

and then executing our verification procedure by substituting in the bound values:

$$\begin{aligned} x_1 + 2x_2 + 3x3 &\geq 17 \\ 4 + 2\cdot 2 + 3\cdot 2 &\geq 17 \\ 16 &\geq 17 \\ \bot. &\quad \text{(explanation verified)} \end{aligned}$$

Example #3: Lifting

We now consider an example that includes lifting. Consider the linear inequality

$$10x_1 + x_2 \geq 15,$$

with domains that allow for no propagation

$$x_1 \in [1,2],\ x_2 \in [0,15].$$

Now consider that another propagator sets $x_2 \leq 0$.

With x_2 not contributing at all to the left-hand side, the propagation algorithm forces $x_1 \geq 2$. A straightforward explanation based on the current domains is

$$\langle x_2 \leq 0\rangle \implies \langle x_1 \geq 2\rangle.$$

We may notice that, given $x_1 \geq 2$, the constraint is "over-satisfied" since the left-hand side of the constraint exceeds the right-hand side. This suggests that lifting may be possible. We may consider the only variable that is responsible for the propagation and weaken its bound in the explanation, obtaining

$$\langle x_2 \leq 4\rangle \implies \langle x_1 \geq 2\rangle.$$

Lifting is an example where verification is particularly useful. Had we increased the bound for the variable x_2 excessively, the verification procedure would detect the resulting inconsistency.

7 Cumulative

The CUMULATIVE constraint applies to activities characterised by start times, durations, and resource requirements, and enforces that, at any time, the total resource consumption of all executing activities does not exceed a given capacity.

It plays a key role in scheduling problems, which historically constitute one of the main application areas of constraint programming.

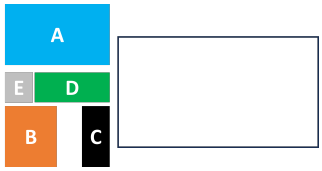
Determining whether a feasible assignment exists for the CUMULATIVE constraint is NP-complete. As a result, a wide range of propagation algorithms with varying strengths and computational costs have been proposed in the literature.

In this chapter, we study *timetabling*, one of the most widely used propagation algorithms, as well as its stronger but more computationally expensive extension, known as *energetic reasoning*.

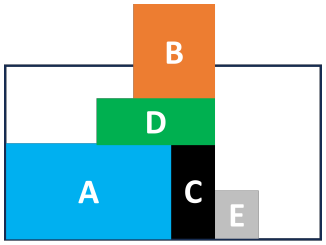
7.1 Illustrative Examples

Consider a scheduling problem involving a set of tasks to be executed over a finite time horizon. Each task is characterised by a fixed duration and a fixed resource requirement. A limited amount of a shared resource is available, and at no point in time may the total resource consumption of the executing tasks exceed this capacity.

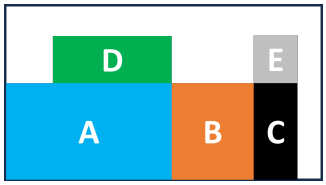
The figures below illustrate an instance of this problem together with two candidate schedules. In the middle schedule, the resource consumption exceeds the available capacity, rendering the schedule infeasible. In contrast, the schedule shown on the right respects the resource constraint and is therefore feasible.



Initial problem.

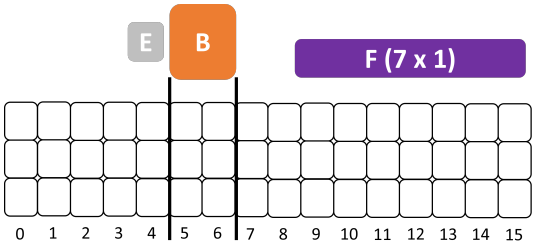


Infeasible assignment.



Feasible assignment.

We may also allow tasks to have different ranges of possible start times. For example, in the instance shown below, task F may be scheduled anywhere within the time horizon, whereas tasks B and E must start and finish within the time interval between 5 and 6.



Propagators for the CUMULATIVE constraint infer lower and upper bounds on the start times of tasks by accounting for resource availability and the possible start times of other

tasks. In the example above, this reasoning allows us to conclude that task F cannot start before time 6.

7.2 Formal Definition

We first introduce a few notions before formally defining the constraint.

A task is represented by a triplet (s_i, d_i, r_i), where s_i is an integer variable denoting the start time of task i, $d_i \in \mathbf{N}$ is its duration, and $r_i \in \mathbf{N}$ is its resource requirement.

Given a task i and a time point t, let $a_{i,t}$ be a binary variable indicating whether task i is active at time t, that is, whether it consumes resources at that time. Formally,

$$a_{i,t} = 1 \iff s_i \leq t < s_i + d_i.$$

Let T denote the scheduling horizon, that is, the set of time points under consideration. It is commonly assumed that the horizon is defined by the domain of the variables

$$T = \{0, \ldots, m\},$$

where $m = \max_i \big(UB(s_i) + d_i - 1\big)$.

We may now formally define the constraint. Given a set of tasks

$$X = \{(s_i, d_i, r_i)\}_i$$

and a total resource capacity $R \in \mathbf{N}$, the constraint

$$\text{CUMULATIVE}(X, R)$$

enforces that, at every time point, the total resource consumption of the executing tasks does not exceed the capacity R:

$$\forall t \in T : \quad \sum_{(s_i, d_i, r_i) \in X} a_{i,t}\, r_i \leq R.$$

When multiple resource types are involved (e.g., personnel, machines, or equipment), a separate CUMULATIVE constraint can be posted for each resource.

Determining the feasibility of the CUMULATIVE constraint is an NP-complete problem. We therefore focus on polynomial-time relaxations that enable effective propagation in practice.

A wide range of propagation algorithms have been proposed for the CUMULATIVE constraint. In the following, we study two prominent variants.

Many generalisations of the CUMULATIVE constraint have been studied, including extensions in which task durations or resource requirements are variables, or where tasks can be executed in different modes with distinct characteristics. In this chapter, we restrict our attention to the standard form defined above.

7.3 Variant 1: Timetabling-Based Propagation

The most common approach to propagating the CUMULATIVE constraint is to identify time points at which resources are guaranteed to be consumed and use this information to restrict the start times of tasks. This idea is based on the activity variables $a_{i,t}$ from the definition of the CUMULATIVE.

We first introduce the notion of a *resource profile* as the central concept underlying timetabling propagation. It aggregates information from the activity variables $a_{i,t}$. We then discuss how the resource profile can be used for conflict detection and domain propagation.

Resource profile

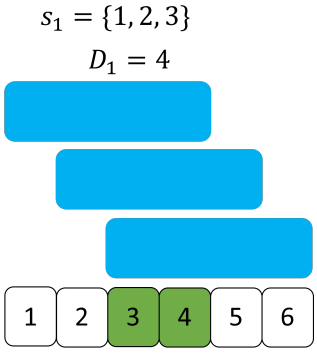
Consider the task illustrated below, represented by

$$(s,d,r) \quad \text{with} \quad s \in \{1,2,3\},\ d = 4,\ r = 1.$$

Given this narrow start-time domain, the task will consume resources at time points 3 and 4 regardless of whether it starts at time 1, 2, or 3. In terms of the activity variables, this corresponds to

$$a_{\text{task},3} = 1 \quad \text{and} \quad a_{\text{task},4} = 1.$$

This mandatory resource consumption follows from the relatively long task duration compared to the width of the start-time domain.



Timetabling identifies compulsory parts (shown in green).

The key observation is that, even though the exact start time of the task is unknown, we can still derive partial information about its resource consumption.

Time points at which a task is guaranteed to consume resources are called *compulsory* (or *mandatory*) *parts*. These are exactly the time points for which the corresponding activity variable $a_{i,j}$ from the definition must be true.

Computing the compulsory parts of all tasks yields a *resource profile*, which provides an incomplete but useful view of the aggregate resource consumption over time.

We may view the resource profile as a function $P(t)$ that captures, at each time point t, the total mandatory resource consumption at that time:

$$P(t) = \sum_i a_{i,t} r_i$$

A simple implementation of the resource profile tracks, for each time unit, the set of active tasks and the total resource consumption. More efficient implementations avoid storing information for every time slot; instead, they operate on contiguous time ranges, which is essential for large problems with long time horizons. This is a key advantage over the decomposition of the cumulative.

Conflict Detection

Algorithm. This resource profile can be used to detect infeasibility.

For example, consider a similar setting as before, but now two tasks, each having duration $d = 4$ and start-time domains $s_1, s_2 \in \{1,2,3\}$. Suppose that each task consumes one unit of resource and that the total resource capacity is one. Reasoning over the compulsory parts reveals that both tasks must consume resources at time points 3 and 4, thereby exceeding the available capacity. Hence, the instance is infeasible.

Detecting infeasibility based on a subset of tasks can significantly reduce the search space, since assignments for other tasks need not be considered.

The conflict-detection algorithm essentially reduces to computing the resource profile and checking whether there exists a time point at which the resource capacity is exceeded:

$$\exists t \in T: \quad P(t) > R.$$

If such a time point t exists, the propagation declares a conflict. Otherwise, no conflict is detected, and the algorithm proceeds with propagation.

The above procedure could be extended to check whether there exists a task that cannot be scheduled anywhere without exceeding the mandatory parts. In conflict detection, this check is usually omitted because it is implicitly handled by the propagation part. However, it is important to take this into account when implementing the checker (see further).

Explanations. Variables corresponding to tasks whose mandatory parts collectively exceed the resource capacity appear in the explanation. Since the bounds of these variables directly determine the mandatory parts, these bounds form the explanation.

Formally, let t be a time point at which the resource capacity R is exceeded by the mandatory parts of a set of tasks Ω, that is,

$$\sum_{i \in \Omega} r_i a_{i,t} > R.$$

Then the explanation for this conflict consists of the bounds of the starting times s_i of the violating tasks:

$$\bigwedge_{i \in \Omega} \big(\langle s_i \geq LB(s_i) \rangle \wedge \langle s_i \leq UB(s_i) \rangle \big) \implies \bot.$$

Lifting explanations can be performed either by removing tasks from the set Ω or by relaxing the bounds of individual tasks, provided that the resulting explanation still preserves the conflict.

Verification. The verification procedure for timetabling follows the same general principles as the conflict-detection algorithm: it reconstructs the resource profile and performs

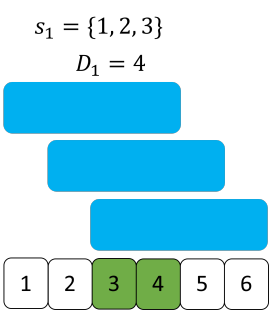
the conflict check. The main difference is that it executes both components of the conflict-detection logic (detecting excess resource consumption and identifying tasks that cannot be scheduled), and the checker uses a simpler and more trustworthy implementation—for example, evaluating the timetable per time point rather than using the interval-based approach, avoiding incremental computation, and without any lifting of explanations. Verification of conflicts is essential for asserting the correctness of propagation (see further).

Propagation

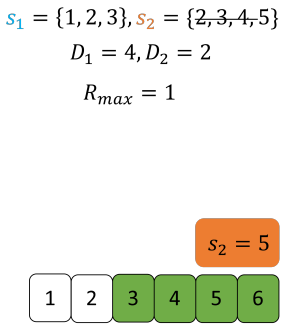
Algorithm. The resource profile can also be used to propagate bounds on the start times of tasks. The main idea is to perform propagation if assigning the task to its bound values would directly lead to a conflict based on the compulsory parts.

For example, consider extending our previous example with two tasks. The first task has the same start-time domain as before, $s_1 \in \{1,2,3\}$, with duration 4. The second task has start-time domain $s_2 \in \{2,3,4,5\}$ and duration 2. Each task requires one unit of resource, and only one unit is available.

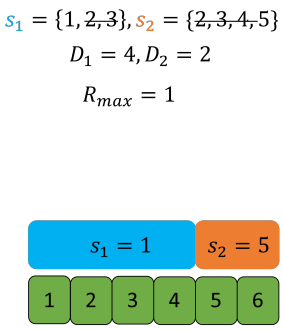
Computing the compulsory parts of the first task reveals compulsory resource consumption at time points 3 and 4 (as before), whereas the second task has no compulsory parts given its current domain. For convenience, we reproduce the compulsory part visualisation in the left-most figure below.



Timetabling identifies compulsory parts (shown in green).



Compulsory parts used to propagate the lower bound of s_2.



Compulsory-part reasoning reveals the only feasible solution.

We may now propagate the start times of the second task by taking into account the compulsory parts of the first task. The earliest start time for the second task is time point 2; however, assigning it to that start time would immediately lead to excess resource consumption on time point 3. Therefore, we can conclude that time point 2 is infeasible for the second task and remove it from its domain. By applying the same reasoning, we also determine that time points 3 and 4 are infeasible. As a result of this propagation, the second task is left with a single feasible value: $s_2 = \{5\}$.

Re-evaluating the resource profile, we observe that the second task now has compulsory parts at time points 5 and 6. This, in turn, forces the first task to start at time point 1, yielding $s_1 = \{1\}$ and completing the assignment through propagation alone (right-most figure). No further propagation is possible; we have reached a fixed point.

This fixed point was reached in three iterations: in the first, pruning the domain of s_2; in the second, pruning the domain of s_1; and, finally, an iteration in which no additional propagation was possible.

It is important to note that compulsory parts generated by a task i may be used only to propagate the start times of other tasks and *not* those of task i. Violating this principle is a common pitfall in timetabling-based propagation implementations.

In practice, timetabling is typically used to propagate lower and upper bounds on task start times. Although it is possible to remove interior values, creating holes can be computationally expensive while offering no benefit to timetabling reasoning. Moreover, scheduling applications commonly rely on interval variables because they often involve a large number of potential time points. Overall, these factors make it standard practice to propagate only bounds and never to introduce holes in the domains.

Explanation. As timetabling propagation reasons over the bounds of the tasks, this informs us that explanations will necessarily involve these bounds.

In our previous example, we were able to propagate the lower bound of the second task because the start times of the first and second tasks were constrained. This leads to the following explanation:

$$\langle s_1 \geq 1\rangle \wedge \langle s_1 \leq 3\rangle \wedge \langle s_2 \geq 2\rangle \implies \langle s_2 \geq 5\rangle.$$

Notice the bounds used in the explanation. The variable s_1 contributes both its lower and upper bounds on the left-hand side, as required to justify the mandatory part. The propagated variable s_2 appears on both the left- and right-hand sides. Crucially, the explanation must include the lower bound of s_2; without it, the propagation would not be triggered. In contrast, the upper bound of s_2 is not required on the left-hand side.

After this propagation, we then inferred $s_1 \leq 1$, which is captured by the following explanation:

$$\langle s_1 \leq 3\rangle \wedge \langle s_2 \geq 5\rangle \wedge \langle s_2 \leq 5\rangle \implies \langle s_1 \leq 1\rangle.$$

Once again, the propagated variable (s_1 in this case) appears on both the left- and right-hand sides of the explanation.

Lifting explanations can significantly improve the performance of timetabling reasoning. In some cases, it is possible to relax the bounds of the variables involved while still justifying the same propagation.

In the previous example, no lifting was possible, but consider a scenario in which two units of the resource are available at time point 4 (at other time points, only one unit of the resource is available), which could happen as a result of scheduling other tasks. In this case, we would propagate $s_2 \geq 4$, which could be explained by

$$\langle s_1 \geq 1\rangle \wedge \langle s_1 \leq 3\rangle \wedge \langle s_2 \geq 2\rangle \implies \langle s_2 \geq 4\rangle,$$

although a stronger explanation can be obtained by lifting the lower bound of s_1:

$$\langle s_1 \geq 0\rangle \wedge \langle s_1 \leq 3\rangle \wedge \langle s_2 \geq 2\rangle \implies \langle s_2 \geq 4\rangle.$$

Naturally, tasks that do not contribute to the compulsory parts responsible for the propagation can be omitted from the explanation. For example, if we were to add a new task

$s_3 \in \{0,1,2,\ldots,10\}$ with duration two, its bounds would be irrelevant for the previous explanations.

Verification. We convert propagation explanations into conflict explanations and then apply the standard procedure for verifying conflicts. The conversion is performed in the same way as for the linear inequality propagator: we remove the consequent and add its negation to the left-hand side.

Although the above procedure is sufficient, we can exploit information from the consequent to speed up verification. In particular, when verifying propagations, the resulting conflict explanation is expected to fail because, if the explanation is correct, starting the propagated task at any time point in its domain immediately leads to excess resource consumption. Instead of reconstructing the entire timetable and checking all tasks, we can restrict our attention to the propagated task and the time points in its domain that are specified by the explanation.

7.3.1 Comparison with the decomposition

Timetabling propagation is based on reasoning over the resource profile, which is derived directly from the decomposition and the activity variables. Consequently, timetabling reasoning offers *the same propagation strength* as the decomposition, but it avoids the overhead of maintaining a large number of per-time-slot variables.

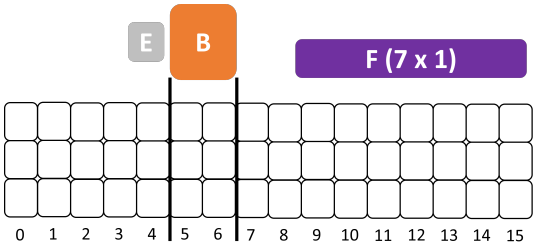
A key advantage of timetabling is that it does not require explicitly representing each time slot. Instead, it works naturally with interval variables that describe the domains of task start times using only their lower and upper bounds. This allows us to reason efficiently even when domains are large. As a result, the cost of propagation does not grow with the granularity of time representation. For example, if we multiply all task durations by 100, the decomposition-based approach would create one hundred times more resource constraints, whereas the timetabling algorithm would remain unchanged. This makes timetabling highly suitable for problems with fine-grained time scales or large domains.

Another benefit of this approach is that explanations derived from timetabling can often be made stronger through lifting. Because we keep information about the structure, we can apply lifting to relax some of the predicates appearing in explanations while still preserving correctness. This can produce more general explanations that prune larger portions of the search space. In practice, such lifted explanations can make a substantial difference in solver performance.

7.4 Energetic Reasoning

As timetabling reasoning only approximates the NP-complete problem represented by the CUMULATIVE constraint, it is natural that there are situations in which timetabling propagation falls short. We sketch the idea below.

Consider the following example, where tasks B and E must both be completed within the interval $[5,6]$, whereas task F is free to start at any time. Given these requirements, task B is fixed to start at time point 5, and task E may start at either 5 or 6.



Applying timetabling reasoning to this instance produces no propagation, even though it is, in some sense, clear that task F cannot start before time point 6.

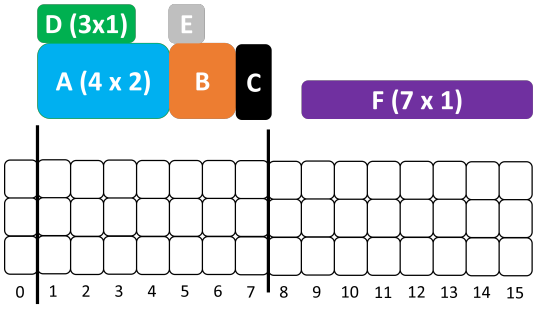
The intuition behind stronger propagation is as follows. Consider a relaxed setting in which tasks may be preempted arbitrarily often, allowing us to break them into unit-sized subtasks. This relaxation is weaker than the original CUMULATIVE constraint, which means that any inference we draw in the relaxed model is also valid for the true cumulative constraint, although the reverse does not necessarily hold.

Under this relaxation, we can reason as follows. In the time interval $[5,6]$, only one unit of the resource is available: task B requires four units, and task E requires one. Therefore, task F cannot start before time point 6, as starting earlier would require two units of resource within the interval $[5,6]$, but only one unit is available.

This example suggests a more powerful relaxation than timetabling: instead of reasoning over resource profiles, we approximate the two-dimensional structure (time and resource) using a single quantity called *energy*. Intuitively, energy corresponds to the area of the rectangle given by the time interval and resource consumption. For instance, an interval of length three with five units of capacity contains fifteen units of energy, and each task requires a fixed amount of energy depending on its duration and resource usage.

By comparing the available energy within a given interval to the energy required by the tasks whose execution overlaps that interval, we may be able to derive propagations that timetabling alone cannot infer, as demonstrated in the example above.

We now consider another example, adapted from Kameugne et al. (2013), in which tasks A–E must be completed within the interval $[1,7]$, while task F may start at any time. Timetabling reasoning does not yield any propagation in this case.



Let us apply energetic reasoning instead. The interval $[1,7]$ has length seven and capacity three, so its total available energy is $7 \times 3 = 21$. Using the same principle, we compute the energy requirement of each task i as $e(i) = d_i \times r_i$. For example, $e(A) = 4 \times 2 = 8$ and $e(B) = 2 \times 2 = 4$. Since the execution windows of tasks A–E lie entirely within $[1,7]$, these tasks require a total of 18 units of energy, leaving only 3 units of energy available within the

interval.

Task F requires a total of 7 units of energy. Because only 3 units can come from within $[1,7]$, at least 4 additional units must come from outside that interval. This leads us to the conclusion that task F cannot start before time point 5, otherwise there is not enough energy, so we infer the propagation $s_F \geq 5$.

This form of reasoning is known as *energetic reasoning*. It strictly subsumes timetabling: any propagation detected by timetabling will also be detected by energetic reasoning, but not necessarily the other way around.

However, energetic reasoning is computationally more expensive than timetabling. In the previous examples, we happened to select the right intervals to analyse, but in the worst case, the number of possible intervals grows quadratically with the time horizon.

Because of this, full energetic reasoning is not commonly used in practice. Instead, it has motivated the development of hybrid techniques that augment timetabling reasoning with limited forms of energetic reasoning.

8 All-Different

The ALL-DIFFERENT constraint enforces that no two integer variables in a given set may take the same value; that is, a violation occurs whenever two variables are assigned an identical value.

Classic applications of this constraint include assignment problems (e.g., assigning exactly one job to each person), permutation problems, and more complex constraints that rely on all-different reasoning as a subcomponent, such as the CIRCUIT constraint (Section 9).

In this chapter, we study three propagation algorithms for the ALL-DIFFERENT constraint, ranging from simple pairwise filtering to domain-consistent propagation, presented in order of increasing strength and computational cost.

8.1 Illustrative Examples

Consider the *assignment problem*: given a set of agents and a set of tasks, where each agent can perform only some of the tasks, the goal is to assign exactly one task to each agent.

We associate each agent i with an integer variable x_i, whose domain represents the tasks that the agent can perform. For example, the assignment $x_3 = 1$ means that agent 3 is assigned task 1.

The ALL-DIFFERENT constraint naturally arises in this problem, since each task can be assigned to at most one agent.

For a problem with four agents, the following is a feasible assignment, assuming that the only constraint is the ALL-DIFFERENT constraint and $x_i \in [1,4]$:

$$x_1 = 1, \quad x_2 = 3, \quad x_3 = 2, \quad x_4 = 4.$$

Each agent is assigned a distinct task, and the assignment therefore satisfies the constraint.

In contrast, the following assignment is infeasible, since agents 1 and 2 are both assigned task 1:

$$x_1 = 1, \quad x_2 = 1, \quad x_3 = 2, \quad x_4 = 4.$$

The above examples considered full assignments. However, partial assignments may already be infeasible. For example, consider the following domains:

$$x_1 = \{1,2,3\}, \quad x_2 = \{1,2,3\}, \quad x_3 = \{1,2,3\}, \quad x_4 = \{1,2,3\}.$$

All four agents can perform only the same three tasks. Consequently, it is impossible to assign a distinct task to each agent, and the problem is infeasible even though no variable has been assigned a single value.

Propagation can also be used to remove values that cannot participate in any feasible solution. Consider the following domains:

$$x_1 = \{1,3\}, \quad x_2 = \{1,3\}, \quad x_3 = \{1,2,3,4,5,6\}, \quad x_4 = \{1,2\}.$$

By inspection, we can conclude that task 2 must be assigned to agent 4 in any feasible assignment. Indeed, assigning the alternative task 1 to agent 4 would leave task 3 as the only possible option for both agents 1 and 2, which is infeasible.

By the same reasoning, tasks 1 and 3 must be assigned to agents 1 and 2 in some order. As a result, none of the tasks 1, 2, or 3 can be assigned to agent 3.

After propagation, the domains become:

$$x_1 = \{1,3\}, \quad x_2 = \{1,3\}, \quad x_3 = \{4,5,6\}, \quad x_4 = \{2\}.$$

The above reasoning was carried out by hand. We will now formally define the constraint and study how such propagation can be performed mechanically.

8.2 Formal Definition

Given a set of integer variables X, the constraint

$$\text{ALL-DIFFERENT}(\mathrm{X})$$

imposes that no two variables may be assigned the same value

$$\forall x_i, x_j \in X : \quad x_i \neq x_j.$$

We assume that all variables have finite domains. Variables with infinite domains could, in principle, be incorporated, but we ignore this nonstandard case for simplicity.

8.3 Variant 1: Propagation by Pairwise Decomposition

The simplest approach to propagating the ALL-DIFFERENT constraint is to decompose it into a set of pairwise disequality constraints, following its formal definition.

For example, an ALL-DIFFERENT constraint over the variables $\{x_1, x_2, x_3\}$ can be decomposed into the three constraints

$$x_1 \neq x_2, \quad x_1 \neq x_3, \quad x_2 \neq x_3.$$

This results in a quadratic number of constraints in the number of variables.

The same idea could also be implemented directly by a propagator, potentially avoiding the quadratic memory overhead of explicitly posting all pairwise constraints. In practice, however, the difference is often negligible.

The main advantage of this approach is its simplicity and low computational cost. However, such a decomposition provides only weak propagation and may fail to detect infeasibility. For example, consider an ALL-DIFFERENT constraint over variables

$$\text{ALL-DIFFERENT}(x_1, x_2, x_3) \wedge x_1, x_2, x_3 \in \{1, 2\}.$$

These constraints are infeasible, since three variables cannot take distinct values from a set of size two. However, the pairwise disequality decomposition does not detect this inconsistency. Instead, the solver must rely on search to conclude infeasibility.

We now turn to more sophisticated propagation algorithms.

8.4 The Key Idea Behind Stronger Propagation: Hall Sets

Stronger propagation for the ALL-DIFFERENT constraint relies on identifying subsets of variables, known as *Hall sets*, whose domains are so restricted that they must be matched among themselves in any feasible assignment.

We implicitly used Hall set reasoning in our previous examples. Recall the following domains:

$$x_1 = \{1, 3\}, \quad x_2 = \{1, 3\}, \quad x_3 = \{1, 2, 3, 4, 5, 6\}, \quad x_4 = \{1, 2\}.$$

Consider the variables $\{x_1, x_2, x_4\}$. The union of their domains is $\{1, 2, 3\}$. Since each variable must take a distinct value and the union of their domains contains exactly three values, it follows that these values must be assigned to these variables. In other words, assigning any of the values $\{1, 2, 3\}$ to another variable would leave at least one of x_1, x_2, or x_4 without a feasible value.

This observation justifies removing the values $\{1, 2, 3\}$ from the domain of variable x_3. We therefore say that the variables x_1, x_2, and x_4 form a Hall set of size three.

An even stronger observation is that the variables x_1 and x_2 already form a Hall set of size two, since their combined domains are $\{1, 3\}$. This implies that the values 1 and 3 must be assigned to x_1 and x_2 in some order, and hence can be removed from the domains of all other variables. In particular, the removal of value 3 from the variable x_4 forces $x_4 = 2$, which itself constitutes a Hall set of size one, allowing the value 2 to be removed from the domain of x_3.

This leads to the following general idea: given the domains D, identify a subset X of k variables such that

$$|X| = k$$

and the union of their domains,

$$D_{\text{union}} = \bigcup_{x_i \in X} D(x_i),$$

contains exactly k values:

$$|D_{\text{union}}| = k.$$

The set X that satisfies the above conditions is called a *Hall set*.

The values in D_{union} must be assigned exclusively to the variables in X and can therefore be removed from the domains of all other variables. Formally,

$$\forall x_j \notin X,\ \forall v \in D_{\text{union}} : \quad x_j \neq v.$$

Depending on how this idea is exploited algorithmically, one obtains either bounds consistency or domain consistency.

Finding a subset of variables X in which the union of domains D_{union} has strictly fewer than k elements,

$$|D_{\text{union}}| < k,$$

is in principle sufficient to declare a conflict. However, we will instead be computing Hall sets ($|D_{\text{union}}| = k$) and removing values from domains to obtain the same conflicts. This is a more convenient view when considering explanations and verifications, as will be discussed further.

8.5 Variant 2: Bound-Consistent Propagation

Rather than reasoning about the union of domains as an explicit set of values, we may instead over-approximate this union by an interval. For instance, the set $\{1,2,5\}$ can be over-approximated by the interval $[1,5]$. This approach is natural when integer variables use an *interval representation* of their domains, that is, when only lower and upper bounds are maintained. The underlying reasoning remains largely the same as before, but operates on intervals rather than on explicit sets of values.

As an example, consider the following domains:

$$x_1, x_2, x_3 \in [0,2], x_4 \in [1,2].$$

The union of these domains is the interval $[0,2]$, which contains at most three values. Since there are four variables, this immediately implies a conflict with the ALL-DIFFERENT constraint.

We now consider a more involved example:

$$x_1, x_2, x_3 \in [0,3], x_4 \in [1,2], x_5 \in [-2,6], x_6 \in [1,6].$$

No conflict is detected in this case, but further domain reduction is possible.

The values in the interval $[0,3]$ must be assigned to the variables x_1, x_2, x_3, and x_4. Consequently, variables x_5 and x_6 cannot take any values in $[0,3]$. However, no propagation can be performed for x_5, since its lower and upper bounds already extend beyond this interval and only bounds information is maintained. In contrast, we can propagate $x_6 \geq 4$, as its lower bound lies within $[0,3]$.

A straightforward implementation iterates over pairs of bounds to form candidate intervals, though more efficient $O(n \log n)$ algorithms exist (López-Ortiz et al., 2003), where n is the number of variables.

Bound propagation for the ALL-DIFFERENT constraint provides a compromise between propagation strength and computational efficiency: it prunes only the domain bounds while avoiding the higher cost of enforcing full domain consistency, which we discuss next.

8.6 Variant 3: Domain-Consistent Propagation

The strongest form of propagation for the ALL-DIFFERENT constraint is obtained by reasoning directly over Hall sets rather than over intervals or pairs of variables. This yields pruning that strictly subsumes the methods discussed previously, albeit at the expense of increased computational resources.

Let us revisit an earlier example:

$$x_1, x_2, x_3 \in [0,3], \quad x_4 \in [1,2], \quad x_5 \in [-2,6], \quad x_6 \in [1,6].$$

The domains of the variables x_1, x_2, x_3, and x_4 imply that these four variables are the only ones that may take values in the range $[0,3]$. Consequently, we can propagate

$$\forall i \in [0,3] : \; x_5 \neq i,$$

resulting in the pruned domain

$$x_5 \in \{-2,-1,4,5,6\}.$$

Recall that the bound-consistent propagator could not achieve this pruning, since reasoning over bounds only manipulates the bound values and does not introduce holes in the domain.

The main challenge in achieving domain-consistent propagation lies in identifying the relevant Hall sets in *polynomial time*.

We first discuss feasibility detection for the ALL-DIFFERENT constraint and then extend the reasoning to derive domain-consistent propagation. The presentation follows the survey of algorithms for the ALL-DIFFERENT constraint (Gent et al., 2008), though we note that other approaches exist that achieve the same level of consistency.

8.6.1 Conflict Detection

To reiterate, conflict detection determines whether it is possible to assign each variable a unique value. We raise a conflict when this is no longer possible even the current domains.

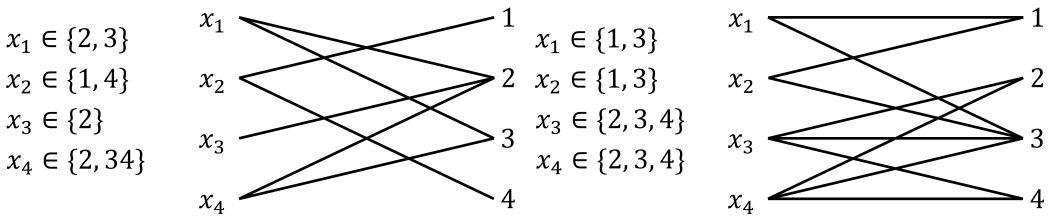
Algorithm based on maximum-flow. Conflict detection can be answered in polynomial time by reducing it to a *maximum bipartite matching* problem, which can be solved using standard *maximum-flow* algorithms.

The first step is to represent the ALL-DIFFERENT as a graph. Given an ALL-DIFFERENT constraint, we build an undirected bipartite graph with two sets of vertices: *variable vertices* and *value vertices*. Each variable becomes a variable vertex, and each value becomes a value vertex.

An edge (x, v) is added between a variable vertex x and a value vertex v whenever $v \in D(x)$. As usual in bipartite graphs, no edges connect two variable vertices or two value

vertices. For readability, we will sometimes refer to variable vertices simply as variables, and to value vertices simply as values.

Below, we illustrate the construction on two example instances. The domains shown on the left determine the edges in the corresponding bipartite graphs.

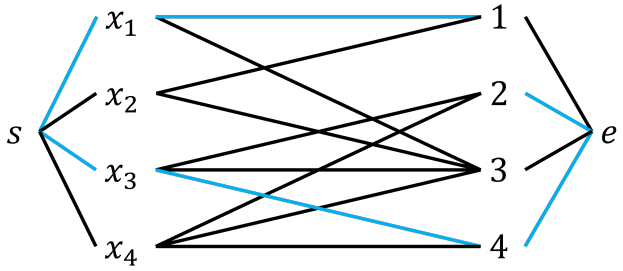


To formulate the corresponding flow problem, we introduce two additional vertices: a *start* vertex s and an *end* vertex e. Directed edges are added from s to each variable vertex, and from each value vertex to e.

All edges are given unit capacity. The crucial observation is that a unit of flow on an edge (x, v) corresponds to the assignment $x = v$. Hence every feasible matching corresponds to a valid flow, and every valid flow maps to a consistent assignment. The ALL-DIFFERENT constraint is therefore feasible if and only if the maximum flow saturates all variable vertices.

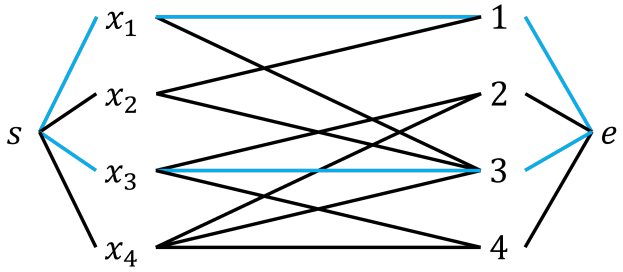
We now discuss these ideas using four examples.

A flow is valid only if each vertex with incoming flow also has outgoing flow, except for the start and end vertices. The example below violates this requirement.



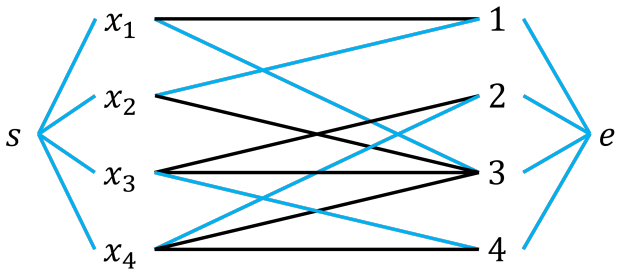
Invalid flow.
Vertex 1 has no proper outgoing flow, and vertex 2 has no proper incoming flow.

To demonstrate that the ALL-DIFFERENT is not conflicting, we must construct a maximum flow. The following flow therefore does not suffice:



Suboptimal flow of two units. A better flow exists.

Instead, the next example shows a maximum flow of four units, establishing that the constraint is feasible.

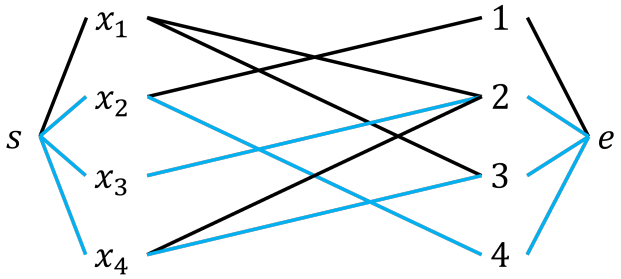


Maximum flow of four units.

From this flow, we can directly read the feasible assignment

$$x_1 = 3, \quad x_2 = 1, \quad x_3 = 4, \quad x_4 = 2.$$

Finally, the following graph illustrates an infeasible ALL-DIFFERENT instance:



At most three units of flow are possible.

The maximum flow is three units, yet there are four variables. The constraint is therefore infeasible. Observe that the variables x_1, x_3, and x_4 have only two distinct values in their domains, which certifies infeasibility. We will think about this conflict as computing a Hall set of size two containing x_1 and x_3, which justify the removal of their domains values from the variable x_4. In this example, we manually detected the Hall set, and later, we discuss how these conflict sets can always be extracted from a maximum-flow computation that demonstrates infeasibility.

To summarise, the value of the maximum flow corresponds to the maximum number of variables that can be assigned unique values in the ALL-DIFFERENT constraint. Achieving a flow of n units—where n is the number of variables—provides an explicit solution demonstrating feasibility. Conversely, if the maximum flow falls short of n, we may declare a conflict.

Before discussing explanations and verification, we first discuss how the maximum flow is computed, because the explanations will be closely tied to the maximum flow computation.

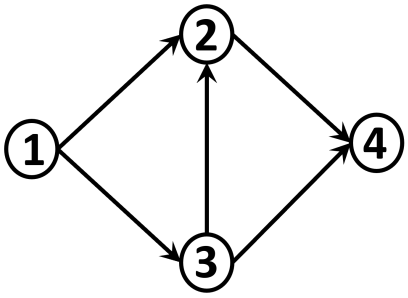
Flow computation. We now recap the Ford–Fulkerson method for computing the maximum flow, which has been shown to be the most effective flow algorithms for the ALL-DIFFERENT by the survey (Gent et al., 2008). This method is particularly relevant because it introduces the concept of a *residual graph*, which we will also use in propagation. Understanding how flow is computed will therefore help us understand how both conflict detection and propagation operates.

We restrict our discussion to the *special case* of directed graphs with *unit-capacity edges*, as these are the graphs that arise in ALL-DIFFERENT reasoning.

Recall that any feasible flow must satisfy the *conservation rule*: for every vertex except the start and end vertices, the amount of incoming flow must equal the amount of outgoing flow.

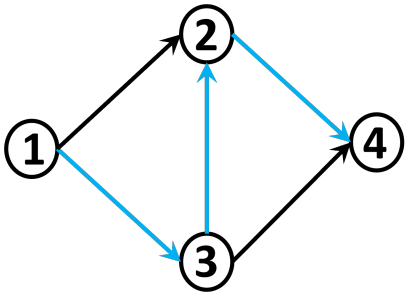
To compute a maximum flow, the idea is to iteratively improve the current flow by repeatedly finding *augmenting paths*. Using depth-first search, we search for a path along which we can push additional flow, increasing the total flow by 1. This greedy process continues until no such path exists and guarantees that we will get the optimal flow in the end.

Consider the example graph below, where we compute the maximum flow from vertex 1 to vertex 4. Initially, the flow is zero.



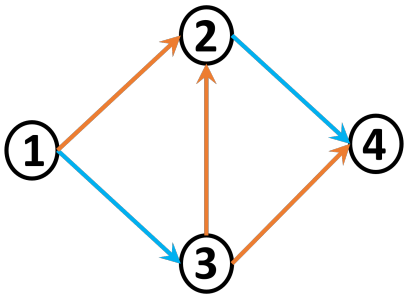
Initial example graph.

In the first iteration, the path $(1,3,2,4)$ allows us to increase the flow by 1, so we push one unit of flow along these edges. Because edges have unit capacity, we can no longer push flow along any of $(1,3)$, $(3,2)$, or $(2,4)$. The resulting flow is shown below.



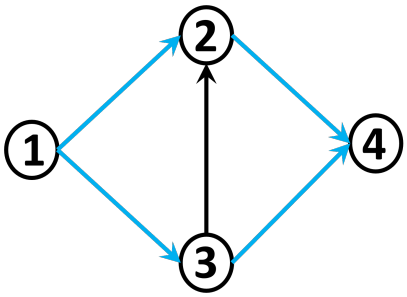
One unit of flow after the first iteration.

In the second iteration, we can push one more unit of flow by considering the path $(1,2,3,4)$.



Augmenting path goes against the existing flow on edge $(3,2)$.

The key detail is that pushing flow through $(2,3)$ *reverses* the previous flow on $(3,2)$. Because flow conservation is still maintained, this reversal is allowed and results in a total increase of 1 unit of flow. The updated flow is shown below.



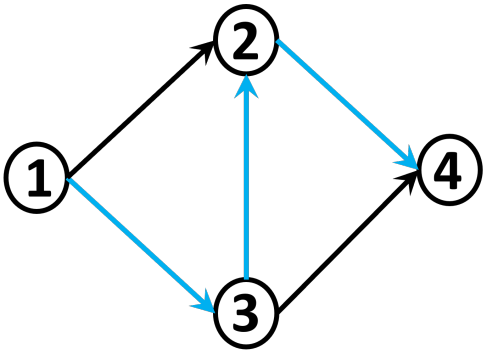
Maximum flow reached after two iterations.

In the third iteration, no further augmenting paths exist, so the algorithm terminates with an optimal flow of 2 units.

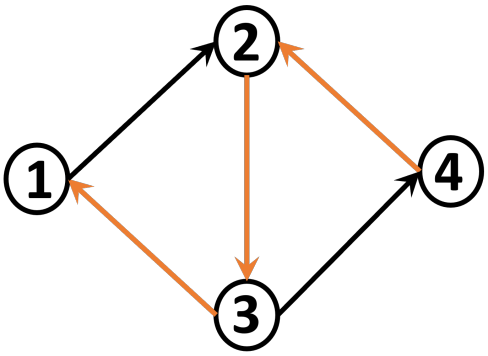
There are several important observations from this example. First, each iteration identifies an **augmenting path** that increases the flow by 1. Second, an augmenting path may push flow in the opposite direction of an existing flow, effectively cancelling it. This does not violate flow conservation: the flow is simply redistributed. Third, as long as an augmenting path exists, the current flow can be improved; and conversely, if no such path exists, the flow is already maximal. Fourth, multiple augmenting paths may be available at any step; any of them is sufficient.

In this discussion, we implicitly used the notion of a **residual graph**. For unit-capacity graphs, the residual graph is constructed from the original graph and the current flow. It contains all vertices of the original graph. An edge (x,y) appears in the residual graph if no flow has yet been pushed through it. If the original graph has flow along (x,y), then the residual graph instead contains the *reverse edge* (y,x).

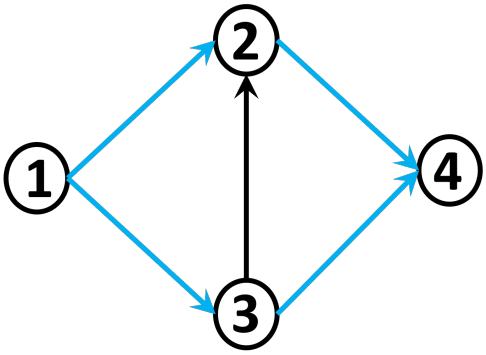
Finding an augmenting path is equivalent to finding a simple path from the start to the end vertex in the residual graph. Below, we show two flows and their corresponding residual graphs.



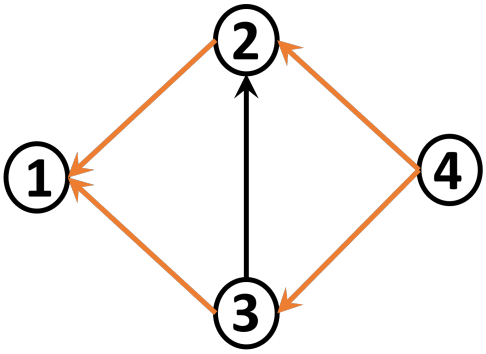
One unit of flow after the first iteration.



Corresponding residual graph with augmenting path $(1,2,3,4)$.



Final flow of two units.

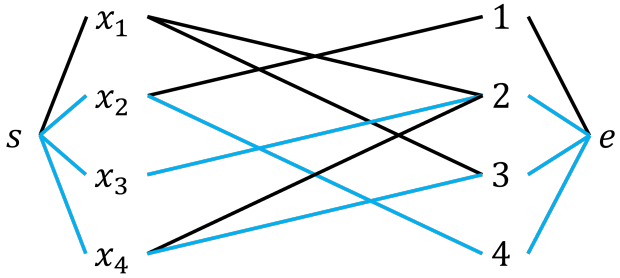


Residual graph with no augmenting path between vertices 1 *and* 4.

We rely heavily on residual graphs and maximum-flow computations in the propagation algorithms that follow.

Explanations. When the domain-consistent propagator detects a conflict, this occurs because a Hall set has removed all values from the domain of at least one variable.

Reproducing the previous infeasibility example for convenience:



At most three units of flow are possible.

In this situation, the conflict can be explained by detecting the Hall set consisting of

variables x_1 and x_3. These two variables together consume all values $\{2,3\}$ that are available to variable x_4, making the domain of variable x_4 empty. Hence, the three variables x_1, x_3, and x_4 suffice to justify infeasibility.

Note that x_3 alone already forms a Hall set of size one, but this is insufficient to explain the conflict: on its own, it does not force the domain of any other variable to become empty.

For this example, the conflict explanation corresponds to the domains of the variables involved:

$$\begin{aligned}&\langle x_1 \geq 2\rangle \wedge \langle x_1 \leq 3\rangle \wedge \\ &\langle x_3 \geq 2\rangle \wedge \langle x_3 \leq 2\rangle \wedge \\ &\langle x_4 \geq 2\rangle \wedge \langle x_4 \leq 3\rangle \implies \bot.\end{aligned}$$

One could also include x_2 in the explanation. However, since x_1, x_3, and x_4 already suffice to exhibit infeasibility, adding a variable whose domain does not intersect with theirs is redundant. Because we aim for non-redundant explanations, we omit variable x_2. For ALL-DIFFERENT, keeping explanations non-redundant also greatly simplifies the verifier (see below).

To compute a non-redundant set of variables involved in the conflict, we inspect the residual graph of the maximum flow. A strongly connected component containing more variables than values, corresponding to a set of vertices in which each vertex is reachable from every other, is a witness to the conflict. Such components can be identified in linear time, e.g., using Tarjan's algorithm.

Verification. A non-redundant explanation for an ALL-DIFFERENT conflict encodes precisely the domains of the variables in a Hall set together with the domain of an additional variable whose values lie entirely within the union of the Hall-set domains. This specific structure enables a verification procedure that is significantly simpler than the flow-based algorithm used to detect the conflict.

Because of this structure, the verifier only needs to:

1. count the number of variables appearing in the explanation,
2. determine the number of distinct values in their domains, and
3. check that the former exceeds the latter.

If there are strictly more variables than values, the explanation correctly establishes infeasibility.

This procedure is one-directional: whenever it concludes infeasibility, the explanation is indeed valid. If it does *not* conclude infeasibility, however, the explanation may still be correct. In such cases, the explanation contains redundancies, and verifying it would require essentially re-running a flow algorithm—a far more complex procedure than we would like for a verification.

Since non-redundant explanations both simplify the verifier and are advantageous for conflict analysis (Chapter 4), it is reasonable to expect that all generated explanations will be non-redundant. This justifies the simplicity of the chosen verification procedure.

8.6.2 Propagation

In the graph representation of the ALL-DIFFERENT constraint, propagation amounts to removing edges that can *never* belong to *any* maximum matching. Since edges correspond to admissible variable–value pairs, removing an edge results in a propagation of the form $x \neq v$.

Naive algorithms. Let us consider naive approaches to domain-consistent propagation. One idea is to iterate over every edge, temporarily delete it, and check whether the size of the maximum matching decreases. If removing the edge reduces the matching size, then the edge is essential for feasibility, and we can infer that the associated variable must take that value. Although this procedure runs in polynomial time, it is too expensive for practical propagation.

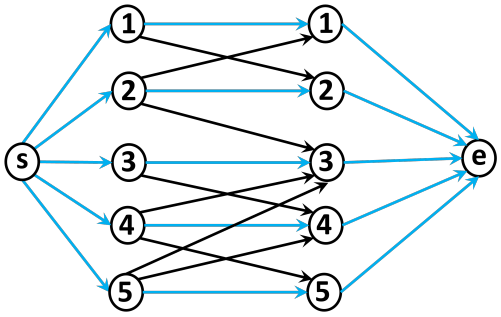
Removing edges is even more challenging when we attempt to detect edges that *never* appear in any maximum matching. To prove such an edge is removable, we must show that among *all* maximum matchings of size n, not a single one contains it. A naive approach would enumerate all maximum matchings and then check which variable–value pairs are absent from all of them. In the worst case, this becomes an exponential-time computation, making it unsuitable for propagation.

Intuition. To improve upon the naive propagation algorithms, we exploit the structure of the residual graph produced by a maximum flow. Graph theory allows us to identify, in polynomial time, exactly those variable–value pairs that can never appear in any maximum matching.

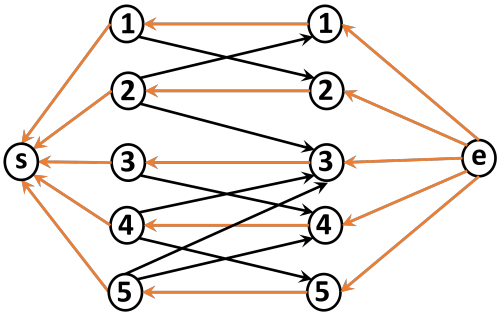
At a high level, the key idea is as follows: *an edge may be pruned if it does not belong to any cycle in the residual graph*. Equivalently, such edges are exactly the edges that connect two different strongly connected components.

Cycles in the residual graph. To develop the intuition, observe first that every directed cycle in the residual graph represents a way to transform the current feasible solution into a *different* feasible solution. Along a cycle, assignment edges (x,v) alternate with unassignment edges (v,x), so reversing all edges on the cycle results in a new matching. The only exceptions are edges involving the start or end nodes, which appear when the number of values exceeds the number of variables. These edges have no direct interpretation as assignments, but all other edges in the cycle follow the logic described above.

We illustrate this with a simple example. Below is a maximum flow and its corresponding residual graph:

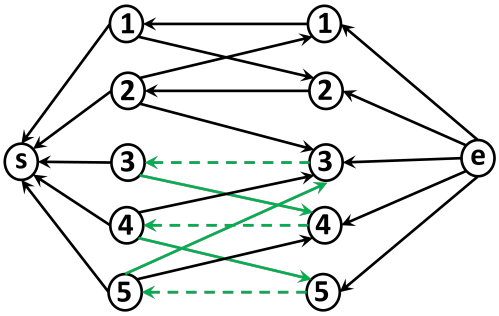


Flow representing the solution $x_i = i$.

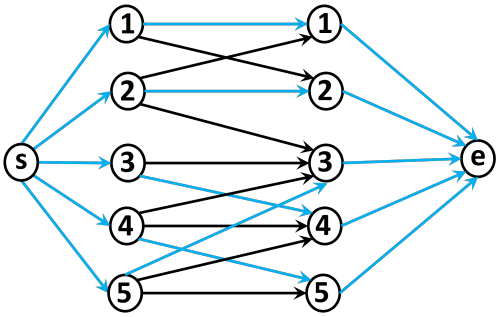


Corresponding residual graph.

Now consider the cycle involving variables 3, 4, and 5:



A cycle in the residual graph: $(v_3, x_3, v_4, x_4, v_5, x_5, v_3)$.



Flow after reversing the cycle: the updated assignment $x_3 = 4$, $x_4 = 5$, $x_5 = 3$.

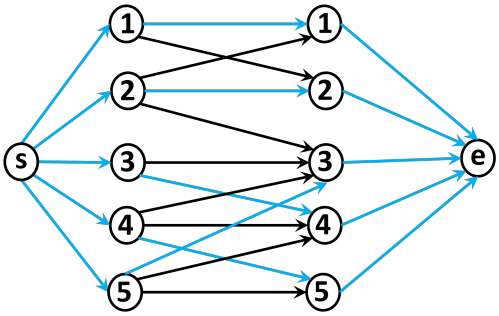
This cycle does not increase flow (so it is not an augmenting path), but reversing its edges produces a different feasible solution. Thus, *cycles serve as assignment generators*: every feasible assignment is reachable from the current one by following an appropriate cycle.

Propagation by removing edges connecting cycles. Propagation, therefore, seeks the edges that *cannot* be part of any cycle. Such edges will never appear in any maximum matching, so they may be pruned from the domains.

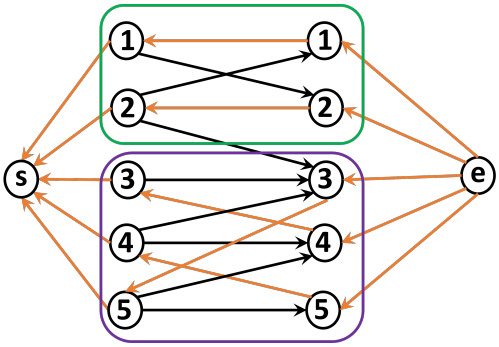
This is where *strongly connected components* (SCCs) of the residual graph become essential. By definition, a path cannot leave an SCC and later re-enter it; if such a cycle existed, the two SCCs would in fact form a single component.

Hence, if an edge (x, v) goes from one SCC to another, then it cannot be part of any cycle, and thus we may prune it.

This is illustrated in the following residual graph with two SCCs.



A maximum flow.

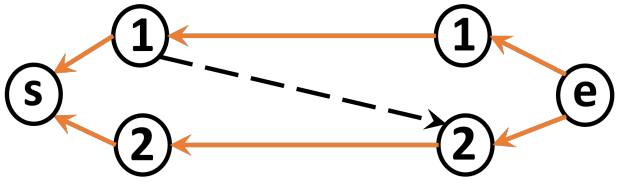


Two SCCs in the residual graph. The edge (x_2, v_3) *crosses between them and may be pruned:* $x_2 \neq 3$.

Using the same reasoning, we observe that any edge connecting two distinct strongly connected components in the residual graph of a feasible flow must be a variable–value edge rather than a value–variable edge. Once such an edge is identified, we prune it, ensuring that no edges remain between components at the fixed point.

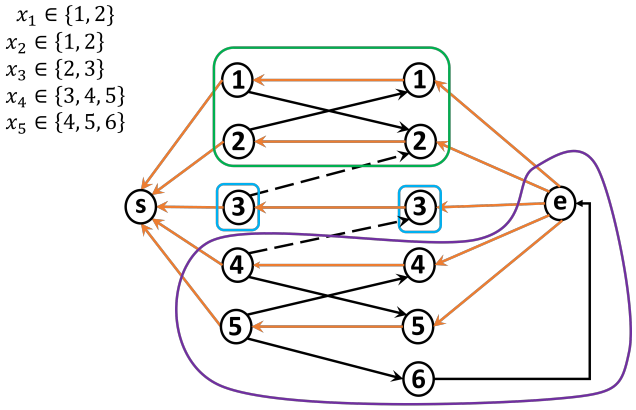
Special case: edges in the current matching. Edges belonging to the current flow represent assignments that are *already* part of a feasible solution; these edges must never be pruned, even if they lie between two SCCs.

For example:



Each node forms a trivial SCC. Edges (x_1, v_1) and (x_2, v_2) are flow edges and cannot be pruned. Only the dashed edge may be removed.

As another example:



The edge (x_3, v_3) is part of the flow and therefore cannot be pruned. Only dashed edges are candidates for removal.

This completes the intuition behind the domain-consistent propagation algorithm for the ALL-DIFFERENT constraint. Depending on the instance, this propagator may be essential for solving.

Explanations. To obtain small and non-redundant explanations, the order in which propagations are performed is crucial. Changing the propagation order incurs no computational overhead, yet it leads to smaller explanations that are easier for a verifier to check.

Propagation is based on the strongly connected components (SCCs) of the residual graph, while explanations rely on the related notion of reachability. Recall that the ALL-DIFFERENT propagator identifies a set X of k variables whose union of domains D_{union} has size k. In this case, every value $v \in D_{\text{union}}$ can be removed from every variable $x_j \notin X$, i.e.

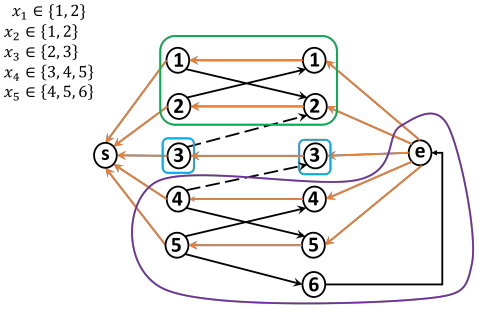
$$\forall v \in D_{\text{union}},\ x_j \notin X : x_j \neq v.$$

In the residual graph, the variables in X form an SCC. Pruning then corresponds to removing edges incoming to this component from other components. If the SCC containing X has no outgoing edges, its domain description alone suffices to explain the propagation.

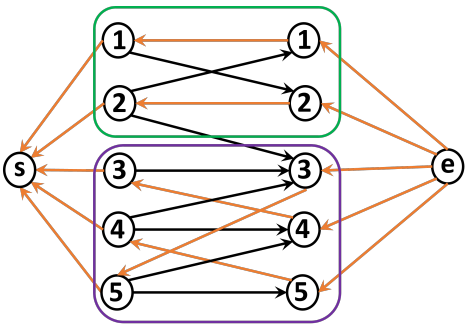
If, however, the SCC of X *has* outgoing edges, then every variable reachable from X in the residual graph must also be included in the explanation. This subsumes the previous case: an SCC with no outgoing edges can reach only itself.

The pruning order influences explanation size. Since all inter-component edges are pruned eventually, performing propagations in *reverse* topological order of the SCC graph results in the smallest explanations. Tarjan's algorithm naturally provides this order.

Consider the following examples.



Possible propagations indicated by dashed edges. The propagation $x_3 \neq 2$ is triggered by the top SCC; the ordering between $x_3 \neq 2$ and $x_4 \neq 3$ affects explanation size.



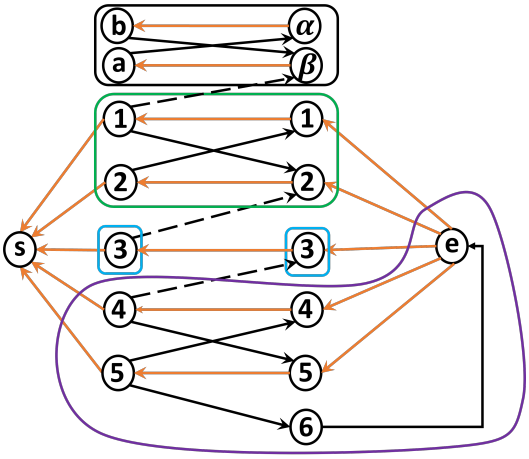
The propagation $x_2 \neq 3$ is explained using the lower SCC.

From the left figure, the propagation $x_3 \neq 2$ is explained by the top SCC:

$$\langle x_1 \geq 1 \rangle \wedge \langle x_1 \leq 2 \rangle \wedge \langle x_2 \geq 1 \rangle \wedge \langle x_2 \leq 2 \rangle \longrightarrow \langle x_3 \neq 2 \rangle .$$

In the right figure, the propagation $x_2 \neq 3$ is explained using the lower SCC. The upper SCC cannot be used: for example, $x_1 = 1, x_2 = 3$ is feasible when considering only the upper SCC, but becomes infeasible once the lower SCC is included.

Now consider the left graph again, but extended with an additional SCC on top.



Propagation order influences explanation size. Dashed edges indicate prunable assignments.

Given the additional SCC, then the explanation for $x_4 \neq 3$ must include the domains of x_1, x_2, x_3 and also x_α and x_β, since all are reachable from value 3 in the residual graph.

Propagating in reverse topological order results in the smallest explanations. In the figure above, the SCCs should be processed from top to bottom:

- $x_1 \neq \beta$ (reason: SCC $\{x_a, x_b\}$),
- $x_3 \neq 2$ (reason: SCC $\{x_1, x_2\}$, since $x_1 \neq \beta$ is already known),
- $x_4 \neq 3$ (reason: $[x_3 = 3]$, since $x_3 \neq 2$ was already propagated).

Lifting. Lifting relaxes explanations by omitting unnecessary predicates for establishing the propagation. A full domain description contains all predicates $\langle x \neq v \rangle$ that record removed values, but some may be irrelevant.

For instance, consider ten variables $x_1, \ldots, x_{10}$ whose union of domains is $\{1, \ldots, 10\}$. The ALL-DIFFERENT propagator removes these ten values from all other variables. The same propagation occurs if $D(x_5) = \{1, \ldots, 9\}$ while all other domains remain $\{1, \ldots, 10\}$. The predicate $\langle x_5 \neq 10 \rangle$ plays no role and may be omitted from the explanation.

Verification. We once again convert the propagation explanation into an equivalent conflict explanation and rely on the standard conflict–verification procedure.

9 Circuit

The **circuit** constraint enforces that variables representing edges in a directed graph form a *Hamiltonian cycle*: the selected edges must form a single cycle that visits each vertex *exactly once*.

The circuit constraint and its variants arise naturally in routing, sequencing, and scheduling applications. For example, in a scheduling problem where executing task j after task i incurs an additional setup time $c_{i,j}$, tasks correspond to vertices, and setup times to edge weights, the problem comes to find a Hamiltonian cycle of minimum weight.

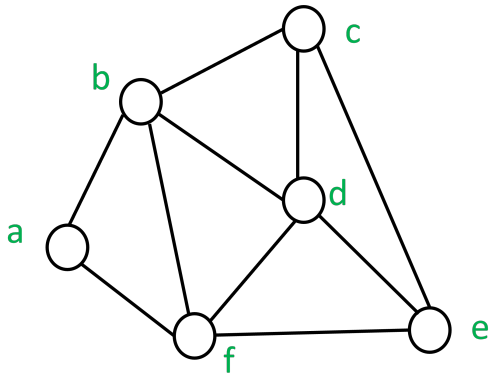
The circuit constraint is particularly challenging, since its decomposition is cumbersome, making propagators a natural alternative.

However, determining whether a graph admits a Hamiltonian cycle is NP-complete. Below, we discuss three propagators that offer different trade-offs between propagation strength and runtime complexity.

9.1 Illustrative Examples

Given a graph, we define integer variables that encode a simple path in the graph. Each vertex in the graph is associated with an integer variable, called a *successor variable*, where the variable x_i denotes the successor of vertex i in the path.

For example, consider the following undirected graph:



The graph is defined by the vertex set

$$V = \{a, b, c, d, e, f\}$$

and the edge set

$$E = \{(a,b),(a,f),(b,f),(b,c),(b,d),(c,d),(c,e),(d,e),(d,f),(e,f)\}.$$

The graph is undirected, so each edge $(i, j) \in E$ implicitly represents both (i, j) and (j, i).

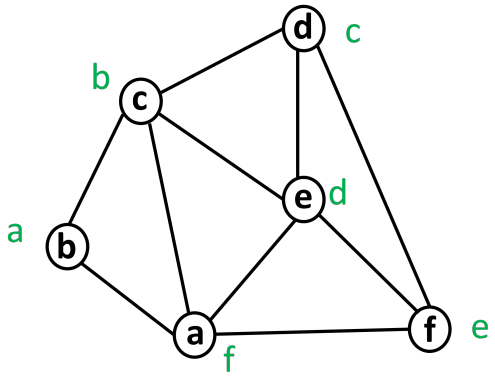
For each vertex $i \in V$, we introduce a successor variable x_i whose domain consists of the vertices adjacent to i. Specifically,

$$\begin{aligned} x_a &= \{b, f\}, \\ x_b &= \{a, f, c, d\}, \\ x_c &= \{b, d, e\}, \\ x_d &= \{b, c, e, f\}, \\ x_e &= \{c, d, f\}, \\ x_f &= \{a, b, d, e\}. \end{aligned}$$

Assigning $x_a = b$ means that the successor of vertex a is vertex b; equivalently, after visiting vertex a, the next vertex on the path is b. The path defined by the successor assignments

$$x_a = b, \quad x_b = c, \quad x_c = d, \quad x_d = e, \quad x_e = f, \quad x_f = a$$

is illustrated below.



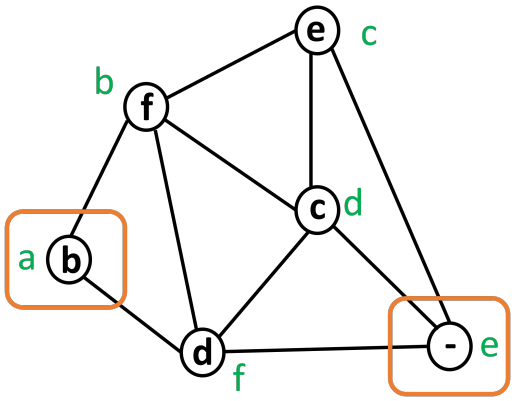
The letter shown inside each vertex indicates the successor assigned to that vertex.

In this case, the constructed path visits every vertex exactly once and returns to the starting vertex, and therefore constitutes a Hamiltonian cycle. The assignment, therefore, satisfies the CIRCUIT constraint.

We now consider several infeasible assignments. The first such assignment is infeasible because it induces a path rather than a cycle. It is given by

$$x_a = b, \quad x_b = f, \quad x_f = d, \quad x_d = c, \quad x_c = e, \quad x_e = \perp,$$

and is depicted below.

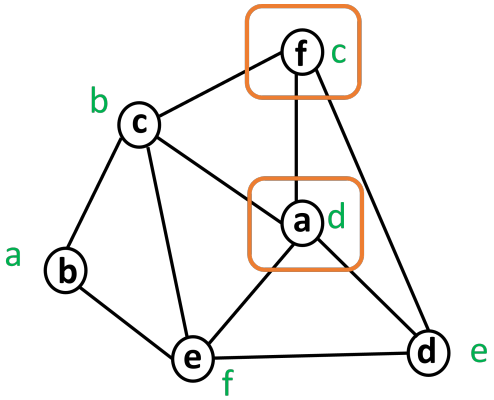


Vertex a has no predecessor, and vertex e has no feasible successor remaining, since each vertex may be assigned as a successor at most once. Consequently, the CIRCUIT constraint declares the assignment infeasible.

Our second infeasible assignment appears to assign a successor to each vertex, but it violates the domain constraints of the variables. It is given by

$$x_a = b, \quad x_b = c, \quad x_c = f, \quad x_f = e, \quad x_e = d, \quad x_d = a,$$

and is shown below.

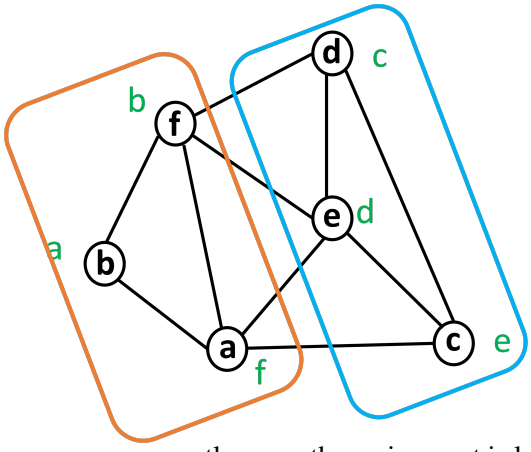


The assignment is infeasible because it violates the variable domains: $x_d = a$ is assigned even though $a \notin D(x_d)$. The same holds for vertex c.

Our third and final infeasible assignment consists of two cycles. It is given by

$$x_a = b, \quad x_b = f, \quad x_f = a, \quad x_d = e, \quad x_e = c, \quad x_c = d,$$

and is shown below.



Although each vertex is a successor exactly once, the assignment is labelled as infeasible because the CIRCUIT constraint stipulates a single cycle rather than two or more cycles.

9.2 Formal Definition

Given a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, we associate with each vertex $i \in V$ an integer variable x_i, called a **successor variable**. The domain of x_i consists of the outgoing edges of vertex i:

$$\forall i \in V: \quad x_i \in \{j \mid (i, j) \in E\}.$$

An assignment $x_i = j$ is interpreted as selecting the edge (i, j); consequently, an assignment to all successor variables determines a set of selected edges.

Given a graph G and its associated successor variables X, the constraint

$$\text{CIRCUIT}(X)$$

enforces that the successor variables X represent a Hamiltonian cycle:

1. Each vertex is visited exactly once:

$$\forall i \in V: \quad \sum_{j \in V} \langle x_j = i \rangle = 1.$$

2. The successor variables encode a single directed cycle.

In the propagation algorithm, the first requirement—namely, that each variable takes a unique value—is handled by the ALL-DIFFERENT propagator (Section 8). In the following, we focus on the second requirement, namely, cycle detection.

9.3 Conflict Detection: Subcycles

Algorithm

The algorithm raises a conflict as soon as a subcycle is formed, that is, a cycle that does not include all vertices. Note that this check assumes that no vertex may have more than one incoming edge, which is enforced by the ALL-DIFFERENT constraint discussed just above (Section 8).

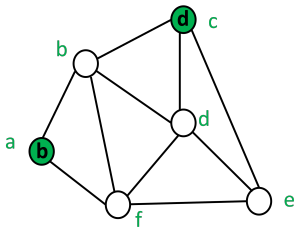
Since variable assignments correspond to edges, conflicts can be detected by traversing the selected edges to identify subcycles. This can be done in linear time using depth-first search and requires visiting only assigned variables.

Example 5 (Running example)**.** *The figures below show successive snapshots of the search process, in which variables are assigned in an arbitrary order. In the leftmost figure, the solver has constructed the partial assignment $x_a = b$ and $x_c = d$. In the next iteration (middle figure), the solver extends the assignment by setting $x_d = e$. After subsequently assigning $x_e = c$ (rightmost figure), the propagator detects a subcycle.*

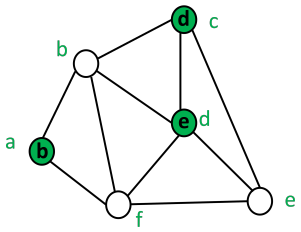
The subcycle is detected by performing a depth-first search starting at vertex a and determining that no subcycle involving vertex a is present. Then a new depth-first search is initiated from vertex c, discovering the cycle

$$c \to d \to e \to c,$$

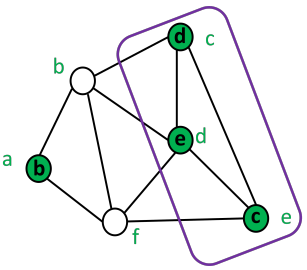
which raises a conflict.



$\langle x_a = b \rangle, \langle x_c = d \rangle$.
No conflict.



Assigning $\langle x_d = e \rangle$.
Still no conflict.



After assigning $\langle x_e = c \rangle$,
cycle detected, conflict.

Explanations

There are two ways of explaining the conflict.

Direct approach. A conflict can be viewed as a set of edges forming a subcycle, which directly corresponds to the associated variable assignments. The direct approach asserts that the partial assignment participating in the cycle is infeasible.

Example 6 (continued)**.** *The conflicting subcycle can be expressed by the variable assignments participating in the cycle:*

$$\langle x_c = d \rangle \wedge \langle x_d = e \rangle \wedge \langle x_e = c \rangle \longrightarrow \bot.$$

Formally, let K denote the set of variables that form a subcycle under the current domain D, where for each $x_i \in K$ the domain is a singleton, i.e., $D(x_i) = \{v_i\}$. The resulting explanation is then given by

$$\bigwedge_{x_i \in K} \langle x_i = v_i \rangle \implies \bot.$$

Indirect approach. For the purpose of verifying correctness, the explanations presented above are sufficient. However, as we will see later in Chapter 4, the generality of an explanation is an important consideration, as it enables stronger reasoning through the combination of multiple propagators.

The explanation above characterises a specific (sub)path. Nevertheless, different permutations of the vertices along this path give rise to essentially the same conflict, which is not captured by the explanation as stated.

Example 7 (continued)**.** *The explanation*

$$\langle x_c = d \rangle \wedge \langle x_d = e \rangle \wedge \langle x_e = c \rangle \implies \bot$$

does not capture the path in reverse order

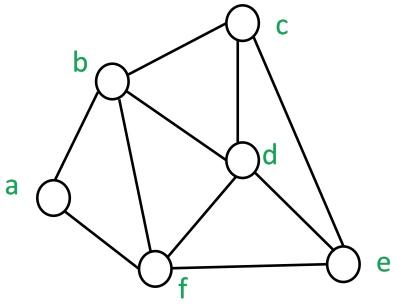
$$e \to d \to c \to e.$$

To address the permutation issue of the direct approach, we consider a potentially longer but more general explanation. Rather than explaining the conflict using the edges that participate in the subcycle, this approach explains the conflict in terms of the edges that are *not* involved. In other words, the explanation enforces that at least one edge incident to a vertex outside the subcycle must be selected.

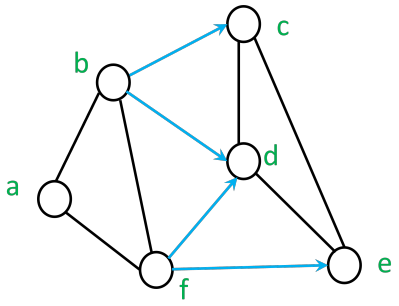
Example 8 (continued)**.** *A more general explanation is*

$$\begin{aligned}
&\langle x_c \neq b \rangle \wedge \langle x_c \neq a \rangle \wedge \langle x_c \neq f \rangle \\
&\wedge \langle x_d \neq b \rangle \wedge \langle x_d \neq f \rangle \wedge \langle x_d \neq a \rangle \\
&\wedge \langle x_e \neq f \rangle \wedge \langle x_e \neq a \rangle \wedge \langle x_e \neq b \rangle \implies \bot.
\end{aligned}$$

Although the atomic constraints highlighted in blue do not appear in the current graph, they must be included in the explanation if the corresponding edges exist in the original graph but have been temporarily pruned by earlier search decisions/propagations. The explanation is visualised below.



Original graph.



After applying the indirect explanation from the above example, it is clear that no Hamiltonian cycle exists, which confirms the explanation.

Formally, let K denote the set of variables that form the subcycle, let H be the set of variables that do not participate in the subcycle (i.e., $H = V \setminus K$), and let D be the current domain. The explanation is then given by

$$\bigwedge_{\substack{x_i \in K \\ x_j \in H}} \langle x_i \neq j \rangle \implies \bot.$$

This approach is more general than the direct method and, in practice, performs significantly better when combined with the analysis techniques introduced in Chapter 4.

Verification

The verification procedure may depend on the form of explanations that the propagator produces.

Verifying the direct approach. A direct explanation explicitly encodes a forbidden subcycle. Verification reduces to checking that the subcycle described in the explanation indeed exists and is strictly smaller than the full cycle.

Concretely, the verifier must ensure that:

1. the edges specified in the explanation are present in the graph,
2. the subcycle involves strictly fewer vertices than the total number of vertices, and
3. starting from any vertex x_s occurring in the explanation and following the edges specified, the traversal eventually returns to x_s, thereby forming a directed cycle.

Direct explanations come with an implicit requirement: they must contain *only* the variables belonging to the subcycle and no redundant vertices or edges. Any redundancy may invalidate the simple verification procedure described above. This requirement is consistent with our general preference for minimal, non-redundant explanations, as discussed in Chapter 4 in the context of conflict analysis.

Verifying the indirect approach. The indirect approach produces more general explanations and therefore requires a more flexible verification procedure. The procedure described here also subsumes the direct case, albeit being more complex.

The verifier begins with the original graph and removes the edges listed in the explanation. If the explanation is correct, the resulting graph will contain a pair of vertices (x_i, x_j) such that x_i is not reachable from x_j or vice versa. This serves as a certificate of infeasibility: if two vertices are not mutually reachable, then no directed Hamiltonian cycle (i.e., a cycle covering all vertices) can exist.

Detecting such a pair of vertices can be done in linear time with respect to the number of vertices and edges. The simplest method is to run a depth-first search (DFS) from an arbitrary vertex, and then run a second DFS on the graph with all edges reversed. If either DFS cannot reach all vertices, the graph cannot contain a Hamiltonian cycle.

An alternative is to compute the strongly connected components, although it is slightly more involved than performing two DFS traversals.

9.4 Propagation: Subcycle Prevention

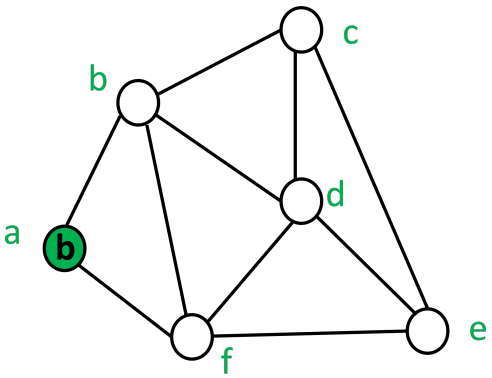
We can design a propagation algorithm that proactively removes edges whose assignment would immediately create a cycle.

This variant extends the conflict detection algorithm by adding a propagation step that prevents cycles from forming between the first and last variables of a *chain*, where a chain is a directed subpath that does not yet close a cycle.

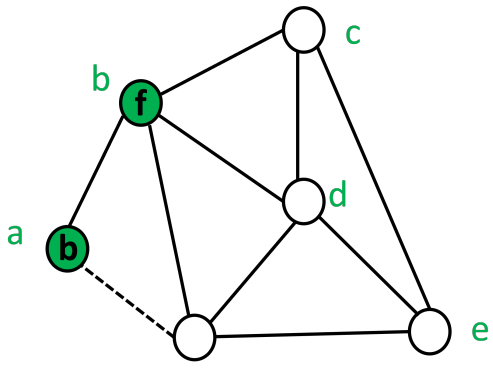
Example 9. *The figures below show two consecutive snapshots of the search process. In the left-hand figure, the solver constructs the partial assignment $x_a = b$, which does not trigger any propagation. After assigning $x_b = f$, the situation changes: choosing $x_f = a$ would now complete the cycle*

$$a \to b \to f \to a.$$

To prevent this, the propagation algorithm eagerly prunes $x_f \neq a$.



No propagation from $\langle x_a = b \rangle$.



After assigning $\langle x_b = f \rangle$, the propagator prunes $\langle x_f \neq a \rangle$.

Subcycle prevention can be implemented in linear time. A first linear scan identifies all chain-start vertices, and a second scan locates the terminal vertex of each chain and performs

the appropriate propagation. This ensures that no chain can be extended into a forbidden subcycle through a single additional assignment.

Explanations

The explanations for propagation follow the same general reasoning as in the conflict detection case.

Direct approach. The chain responsible for the propagation can be represented directly by the assignments that form it. Propagation then prevents the last vertex of the chain from closing a cycle with the first vertex, assuming the cycle would not cover all vertices.

In the previous example, this results in the explanation

$$\langle x_a = b \rangle \wedge \langle x_b = f \rangle \implies \langle x_f \neq a \rangle.$$

Formally, let K be the set of variables participating in the chain

$$a \to \cdots \to z,$$

where a is the first vertex and z is the last vertex whose successor is not yet fixed. Under the current domain D, each $x_i \in K$ has a singleton domain $D(x_i) = \{v_i\}$. Since propagation occurs only when the chain does *not* contain all vertices, the set K is strictly smaller than the vertex set.

The resulting explanation is

$$\bigwedge_{x_i \in K} \langle x_i = v_i \rangle \implies \langle x_z \neq a \rangle.$$

Indirect approach. As with conflict explanations, the direct explanation is sufficient for verification, but a more general explanation can be obtained by abstracting away from the precise order of vertices in the chain.

The limitation of the direct form is analogous to the conflict case: permuting the internal order of vertices between the first and last vertex results in the same propagation. Because it is not straightforward to express that "a is the first vertex in the chain", we reformulate the key observation more generally.

Each vertex in the chain, except the last, has no outgoing edge to a vertex outside the chain under the current assignments. Therefore, in order to avoid forming a subcycle, the last vertex must select a successor outside the chain. Consequently, it cannot choose a as its successor.

Formally, let K be the variables in the chain

$$a \to \cdots \to z,$$

let $H = V \setminus K$ be the variables not in the chain, and let D be the current domain. The explanation becomes

$$\bigwedge_{\substack{x_i \in K, i \neq z \\ x_j \in H}} \langle x_i \neq j \rangle \implies \langle x_z \neq a \rangle.$$

Verification

The correctness of a propagation explanation can be established by rewriting it as an equivalent conflict explanation and then checking the resulting conflict explanation.

The key observation is that, in the indirect case, the equality atomic constraint that would normally appear on the left-hand side—after being moved from the right-hand side—may be replaced by a collection of disequality atomic constraints that encode the same semantics.

9.5 Further propagation

Additional propagation can be obtained by further exploiting the structural properties of the underlying graph.

As discussed earlier, a Hamiltonian cycle requires the graph to be a single connected component. This immediately provides us with a sufficient condition for conflict detection: if the graph does not form a single *strongly* connected component, then a Hamiltonian cycle is impossible. In this case, the propagator may immediately raise a conflict. The complexity remains linear time and the procedure is similar to verifying propagation explanations using the indirect method.

Further propagation may be obtained by exploiting the structure of the subtrees visited during depth-first search. Interestingly, this yields an example of a propagation algorithm whose fixed point is not unique; instead, it depends on the order in which nodes are visited during the depth-first search. We do not discuss this variant in detail, but refer interested readers to the work of Francis and Stuckey (2014) for further information.

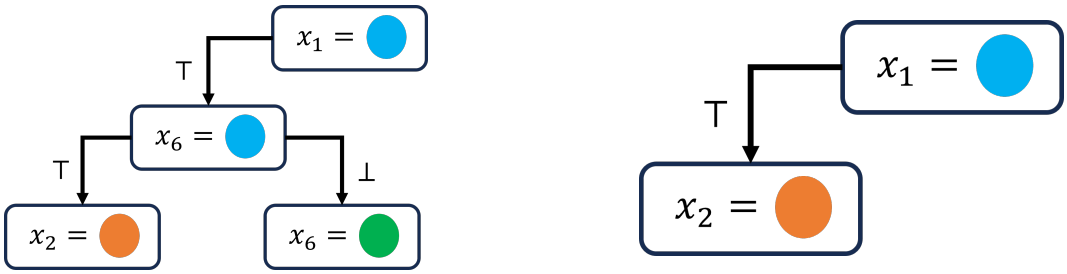
Chapter 3

Branching

The goal of this chapter is to...

- Introduce the high-level ideas behind branching in constraint programming.
- Explain how *variable and value selection* influences the structure of the search tree.

Our search algorithm systematically explores the entire search space. This exploration can be visualised as a binary tree, known as the *search tree*. Below we illustrate two possible search trees for the graph colouring example from Section 1.



Each node in the search tree represents a decision. The search tree summarises the exploration process, showing how the initial problem is recursively split into subproblems.

The size of the search tree depends on how the solver explores the search space. In the example above, both trees solve the problem, but the tree on the right is substantially smaller. Different choices during search may therefore produce significantly different trees.

Constraint programming typically explores the search space using **depth-first search**, although a variation with restarts will be discussed in the next chapter.

We refer to the order in which subproblems are explored as the **branching strategy** (also called the *search strategy*). The branching strategy influences the size of the search tree: in our two examples, the first tree is larger (and incomplete) than the second, making it less effective.

In the graph colouring example, after propagation reached a fixed point, we made an arbitrary assignment of the form $x_i = c$. This splits the current problem into two subproblems: one where $x_i = c$ and one where $x_i \neq c$. If the problem is infeasible, both subproblems must

be infeasible; if it is feasible, at least one subproblem must be feasible. Splitting the problem recursively in this way is called **branching** or **making a decision**.

There are several ways to branch, but all ultimately create two subproblems: one in which a new constraint c is imposed, and one in which its negation of constraint c is imposed. Together, the assignments explored in both branches represent all assignments of the original problem. In the example, we branched on $x_i = k$ and $x_i \neq k$.

In general, we branch using **atomic constraints** of the form $\langle x \otimes k \rangle$, where x is a variable, $\otimes \in \{\geq, \leq, \neq, =\}$, and k is a constant. These constraints typically trigger immediate propagation, which may, in turn, cause further propagation, making them particularly effective branching candidates. More complex constraints (e.g., $\langle x + y \geq 5 \rangle$) could be used, but such branching is uncommon in constraint programming practice and will not be considered further.

Although branching and propagation influence the size of the search tree, neither affects *completeness*. Any correct propagation algorithm and any branching strategy that generates two smaller subproblems still yields an exhaustive exploration of the search space.

For instance, even a naive backtracking search with no propagation will eventually explore the entire search space. However, *in practice*, combining a good branching strategy with effective propagators dramatically improves runtime. With an oracle guiding the branching choices, we could solve all *NP*-complete problems in linear time.

Branching is often decomposed into two interconnected components: 1) *variable selection*, which chooses the variable to branch on, and 2) *subproblem selection*, which determines how to split the problem and which subproblem to explore next. Both components are computationally inexpensive relative to propagation. We discuss each in the following sections.

9.6 Variable Selection

A **variable selection heuristic** selects the next variable on which to branch, taking into account the current domains and possibly statistics gathered during search. In other words, it answers the question: which unassigned variable should be chosen next?

Branching has two conflicting goals: we want to find solutions quickly, but we also want to expose conflicts as early as possible. Effective variable selection seeks to balance these objectives. The heuristics below are among the most widely used. Because we are dealing with NP-hard problems, it is difficult to provide formal guarantees for their performance; instead, we rely on intuition and empirical evidence, acknowledging that their effectiveness is problem-dependent and may vary significantly across instances.

Smallest domain. A natural choice is to select the variable with the smallest domain. For instance, a variable x with domain $\{0,1\}$ might be preferred over a variable y with domain $\{3,5,6,7,8\}$. The idea is that such variables follow the "fail–first" principle: small domains are more constrained and more likely to trigger propagation or reveal inconsistencies early. At the same time, guessing the correct value may require fewer attempts. In this sense, the heuristic supports both finding solutions and conflicts.

Smallest value. Some problems have meaningful domain orderings, and it may be advantageous to select the variable whose domain contains the smallest value. For example, if

$x \in [5, 15]$ and $y \in \{10, 11\}$, and if smaller values represent earlier start times in a scheduling problem, then handling x first corresponds to reasoning about the earliest possible tasks. This can guide the search towards good solutions quickly. Conversely, one may prefer the variable with the largest value in its domain, e.g., selecting tasks that could lead to long schedules first. This is common in scheduling, often paired with a value selection rule that chooses the smallest available value.

Conflicting. Another idea is to prioritise variables that have recently contributed to conflicts. If a variable is frequently involved in conflicts, it is likely tightly constrained and harder to assign. Selecting it sooner may expose conflicts earlier in the search tree. Many variations of this idea exist. In the next chapter once we study conflict analysis, we will discuss VSIDS, a highly successful dynamic heuristic widely used in general-purpose solving.

Fixed order. A simple alternative is to select variables according to a predetermined order, possibly based on domain-specific knowledge or the original variable numbering. Although static orders may be rigid, problem-specific insight can sometimes identify particularly important variables. Moreover, fixed ordering is useful in benchmarking solver configurations, since it ensures that different runs explore the same portion of the search space.

Frequency. One could select the variable that appears most frequently in propagators, on the assumption that branching on such a variable triggers extensive propagation. However, propagators vary greatly in importance, and static frequency counts do not necessarily correlate with pruning power. In practice, frequency alone tends to perform less well than heuristics incorporating dynamic feedback, such as conflict data.

Random. Selecting a variable at random—either uniformly or with a bias such as inversely proportional to domain size—provides a useful baseline against which new heuristics can be evaluated. Although random selection is rarely effective on its own, it serves as an important reference point: at the very least, a new heuristic should consistently outperform a random choice.

Custom. Finally, domain-specific strategies can be crafted by incorporating knowledge of the problem structure. For instance, in our graph colouring example, we consistently chose a node adjacent to a previously selected node, encouraging propagation across the graph. Although tailored heuristics can be effective for specific classes of problems, they can be difficult to design and may not generalise well. They are best considered when general heuristics fail to result in satisfactory performance.

Overall, effective variable selection is crucial for reducing the size of the search tree. Since no heuristic is universally optimal, it is often necessary to adapt the strategy to the specific problem at hand and to evaluate different options empirically.

9.7 Subproblem Selection

Given an unassigned variable chosen by the variable selection heuristic, a **subproblem selection heuristic** determines how to branch by selecting both the comparison operator ($\leq, \geq, =, \neq$) and the right-hand-side value for the resulting atomic constraint. In other words, it answers the question: given a variable, how should the search be continued?

Subproblem selection follows the same guiding principle as variable selection: it should steer the search towards feasible assignments while also encouraging early detection of conflicts, thereby enabling stronger pruning. These goals are inherently contradictory, and the most effective strategy is often problem-dependent.

The first decision concerns the choice of comparison operator. A common approach is to branch using ordering predicates ($\leq$ or $\geq$), as these typically produce more balanced subproblems compared to (in)equality predicates ($=$ or $\neq$). Balanced subproblems reduce the likelihood that one branch contains nearly the entire remaining search space.

The second decision concerns the right-hand-side value, chosen by a **value selection heuristic**. Many options exist, and, as with variable selection, the best choice depends heavily on the nature of the problem. Different variables within the same problem may also benefit from different value selection rules.

Smallest / largest value. Select either the minimum or maximum value in the domain. For example, in scheduling problems, selecting the earliest possible start time can lead to viable schedules quickly. Conversely, selecting the largest value may prioritise tasks that risk pushing the schedule to its limits, which can reveal conflicts earlier.

Reference assignment. Select the value suggested by a given reference assignment. This concentrates the search around an assignment that is already "close" to feasibility, e.g., an assignment produced by a greedy algorithm. This idea is particularly effective in optimisation problems, where the reference assignment may be the current best-known solution.

Split. Select a value that "splits" the domain, such as the median or average. This can create subproblems of comparable size, which may improve balance in the search tree. However, this strategy may be counterproductive for variables closely tied to the objective function, as it can push the search towards poor-quality regions of the search space.

Random. Select the branching value at random, either uniformly or according to a distribution. Random value selection may encourage diversification and serves as a useful baseline for evaluating new heuristics.

Fixed. Define the value ordering in advance, based on domain-specific insight. For each variable, one may specify a preferred sequence of values. When reliable, such domain knowledge can significantly improve performance, though it may not generalise beyond the problem class for which it was designed.

To summarise, variable and subproblem selection together determine the structure of the search tree and thus profoundly influence solving performance. Each heuristic aims to balance the competing goals of guiding the solver towards feasible assignments while also exposing conflicts early enough to enable effective pruning. Although no single strategy is universally superior, a well-chosen combination can dramatically improve efficiency. Ultimately, the choice of branching strategy should reflect both the characteristics of the problem and empirical evidence gathered through experimentation.

Chapter 4

Conflict Analysis

In this chapter, you will learn about:

- **Conflict analysis**, a breakthrough technique that identifies the cause of conflicts and goes beyond traditional backtracking search.
- **Conflict-driven constraint learning (CDCL)**, a solver paradigm that builds its architecture around exploiting information obtained from conflicts.

Propagation reasons locally over individual constraints, with propagators communicating only through shared variable domains. While this local reasoning is highly effective, it has a fundamental limitation: it cannot directly account for interactions between constraints. Consequently, the solver may repeatedly rediscover the same conflicts during search. In the worst case, this can lead to exponentially many repeated failures.

To understand this limitation, recall that our search algorithm reacts to a conflict by undoing the most recent decision and exploring its negation. This strategy makes sense if the last decision was truly responsible for the conflict. However, the conflict may arise from a combination of earlier choices that do not involve the most recent decision, with the most recent decision merely exposing the inconsistency. In such cases, reversing the last decision does not prevent the solver from encountering the same conflict again.

For instance, consider the problem with two linear inequalities

$$x + y \geq 1,$$
$$x - y \geq 0,$$

and domains

$$x \in \{0,1,2,3\}, \qquad y \in \{-1,0,1\}.$$

Viewing the constraints in isolation, no propagation is possible. However, taken together, the constraints imply $x \neq 0$: if $x = 0$, the first constraint requires $y \geq 1$ while the second requires $y \leq 0$, a conflict. Since our propagators cannot reason jointly across both constraints, this conclusion is reached only after explicitly branching on $x = 0$ and encountering a conflict. If the solver branches on variables unrelated to this conflict before branching on x, it may

repeatedly rediscover that x cannot be assigned the value 0. Each such discovery requires encountering a conflict first.

The same phenomenon may arise as part of a larger problem. For example, consider the previous problem extended with a binary variable $z \in \{0,1\}$:

$$x+y+z \geq 1,$$
$$x-y+z \geq 0.$$

The constraints now imply the conditional relationship

$$z = 0 \implies x \geq 1,$$

but this relationship is not represented explicitly. Instead, the solver must rediscover it whenever it encounters the assignment $z = 0$ and subsequently branches on $x = 0$. Consequently, the same conflict may be rediscovered repeatedly in different parts of the search tree.

An even more striking example is the following:

$$x+y = 0,$$
$$x+y = 1,$$

with domains

$$x, y \in \{0,1\}.$$

These constraints are jointly infeasible, regardless of any other constraints in the problem. However, propagators reasoning over the constraints individually cannot detect this immediately. Consequently, any search effort expended before branching on the variables x and y is effectively wasted.

These examples illustrate a common theme: the solver does not retain any information about the underlying reason for a conflict. Once a conflict is encountered, it simply backtracks; there is no mechanism to record the cause of the failure and reuse that information.

This is precisely the purpose of *conflict analysis*, which has become a standard component of modern constraint solvers. Rather than attempting to reason proactively about all possible interactions between constraints (an intractable task in general), conflict analysis reacts *lazily* to conflicts as they occur. It analyses the cause of a conflict and derives new constraints that capture the underlying reason for the failure. In the examples above, such constraints would make explicit conclusions such as $x \neq 0$ or $z = 0 \implies x \geq 1$.

The impact of conflict analysis on solver performance is so significant that entire solver architectures have been built around it. In these solvers, propagation, branching, and learning are tightly integrated, giving rise to the *conflict-driven constraint learning* (CDCL) paradigm.

To develop these ideas, we begin with a concrete example illustrating how conflict analysis derives new constraints, called *nogoods*. We then formalise the conflict analysis algorithm.

As with propagators, correctness is a central concern. We therefore discuss techniques for certifying learned nogoods before turning to the mechanisms that make conflict-driven constraint learning effective in practice, including nogood minimisation, nogood propagation, conflict-guided branching, nogood management, and restarts.

Finally, we discuss recent developments that move beyond traditional nogood learning and allow richer forms of constraints to be derived from conflicts.

10 Illustrative example

Consider the following fragment of a larger problem. The trail is shown on the left and the constraints on the right; additional constraints may exist beyond those shown. Although the example may appear large at first glance, conflict analysis examines it one step at a time, so there is no need to memorise the entire example.

The notation "@ X" denotes the start of decision level X, which begins with a branching decision represented by an atomic constraint. All subsequent propagations inherit the same decision level until the next branching decision. The notation "$c_i \to A$" denotes that constraint c_i propagated the atomic constraint A.

$$
\begin{aligned}
&\langle x_9 \le 0\rangle && @1\\
&\langle x_1 \le 2\rangle && @2\\
&\langle x_2 \le 3\rangle && @3\\
&\qquad c_1 \to \langle x_3 \ge 1\rangle\\
&\qquad c_2 \to \langle x_4 \ge 1\rangle\\
&\langle y \ge 2\rangle && @4\\
&\langle x_5 \le 2\rangle && @5\\
&\qquad c_3 \to \langle x_6 \ge 2\rangle\\
&\qquad c_4 \to \langle x_7 \ge 2\rangle\\
&\qquad c_5 \to \langle x_8 \ge 1\rangle\\
&\qquad c_6 \to \bot
\end{aligned}
\qquad\qquad
\begin{aligned}
&c_1 : 3x_2 + x_3 + x_9 \ge 10\\
&c_2 : -x_3 + 5x_4 \ge 4\\
&c_3 : x_1 + x_5 + 2x_6 \ge 8\\
&c_4 : x_7 - x_6 \ge 0\\
&c_5 : 2x_8 - x_6 + x_9 \ge 0\\
&c_6 : -x_3 - x_4 - x_7 - x_8 \ge -4\\
&\quad x_i, y \in \{-10, 10\}
\end{aligned}
$$

The trail ends in a conflict at decision level 5. Our goal is to analyse this conflict and derive the following nogood, where each atomic constraint is annotated with the decision level at which it was introduced on the trail:

$$\langle x_3 \overset{@3}{\ge} 1\rangle \wedge \langle x_4 \overset{@3}{\ge} 1\rangle \wedge \langle x_9 \overset{@1}{\le} 0\rangle \implies \langle x_6 \le 1\rangle .$$

Although this nogood was not present in the original model, it is logically implied by the existing constraints. As we will see, conflict analysis does not change the set of feasible solutions; it merely makes an implicit consequence of the model explicit.

The nogood reveals that the conflict is not caused by the decisions at levels 4 and 5. Consequently, backtracking a single decision level would not resolve the underlying cause of the conflict, since the nogood would remain violated. Instead, we can *backjump* directly to decision level 3 and use the nogood to propagate $\langle x_6 \le 1\rangle$, effectively avoiding searching at decision levels 4 and 5, which would inevitably lead back to the same conflict. We now examine how such a nogood can be derived mechanically.

A natural starting point is the conflicting constraint

$$c_6 : -x_3 - x_4 - x_7 - x_8 \ge -4$$

and the atomic constraints that lead to its violation

$$\overset{@3}{\langle x_3 \geq 1\rangle} \wedge \overset{@3}{\langle x_4 \geq 1\rangle} \wedge \overset{@5}{\langle x_7 \geq 2\rangle} \wedge \overset{@5}{\langle x_8 \geq 1\rangle} \implies \bot$$

We refer to constraints of this form as *nogoods*. The above nogood captures the immediate cause of the conflict: when all four atomic constraints hold simultaneously, the constraint c_6 becomes infeasible.

However, this nogood is not yet particularly useful. Since it has two atomic constraints from the current decision level 5, it does not yet isolate the earlier decisions responsible for the conflict. More useful nogoods are obtained by combining multiple nogoods from different constraints, progressively eliminating dependencies on the current decision level.

The next step is to refine the nogood. We consider the most recent atomic constraint on the trail, $\langle x_8 \geq 1\rangle$, which was propagated by the constraint

$$c_5 : 2x_8 - x_6 + x_9 \geq 0$$

and its inference can be explained by

$$\overset{@5}{\langle x_6 \geq 2\rangle} \wedge \overset{@1}{\langle x_9 \leq 0\rangle} \implies \overset{@5}{\langle x_8 \geq 1\rangle}.$$

The left-hand side is referred to as the *reason* for the propagation $\langle x_8 \geq 1\rangle$. We may now combine the conflict nogood and the propagation explanation. This means that in our conflict nogood

$$\overset{@3}{\langle x_3 \geq 1\rangle} \wedge \overset{@3}{\langle x_4 \geq 1\rangle} \wedge \overset{@5}{\langle x_7 \geq 2\rangle} \wedge \overset{@5}{\langle x_8 \geq 1\rangle} \implies \bot$$

we *substitute* the propagated atomic constraint $\langle x_8 \geq 1\rangle$ with its propagation reason to obtain a new nogood

$$\overset{@3}{\langle x_3 \geq 1\rangle} \wedge \overset{@3}{\langle x_4 \geq 1\rangle} \wedge \overset{@5}{\langle x_7 \geq 2\rangle} \wedge \overset{@5}{\langle x_6 \geq 2\rangle} \wedge \overset{@1}{\langle x_9 \leq 0\rangle} \implies \bot.$$

First, observe that the new nogood is logically implied by the problem. We merely replaced one atomic constraint in the original conflict nogood with its propagation reason. Since the reason implies the propagated atomic constraint, the resulting nogood remains a logical consequence of the constraints.

Second, the new nogood provides a different explanation of the conflict. Although it still contains two atomic constraints from the current decision level 5, we have nevertheless made progress: the propagated atomic constraint $\langle x_8 \geq 1\rangle$ has been eliminated and replaced by atomic constraints that occurred earlier on the trail. Repeating this process allows us to progressively move the explanation away from the conflict itself and towards the atomic constraints earlier on the trail responsible for the conflict.

We repeat the previous procedure by considering the next-most-recent atomic constraint, $\langle x_7 \geq 2\rangle$. It was propagated by the constraint

$$c_4 : x_7 - x_6 \geq 0$$

with the propagation explanation

$$\overset{@5}{\langle x_6 \geq 2\rangle} \implies \overset{@5}{\langle x_7 \geq 2\rangle}.$$

We once again take our current conflict nogood

$$\overset{@3}{\langle x_3 \geq 1\rangle} \wedge \overset{@3}{\langle x_4 \geq 1\rangle} \wedge \overset{@5}{\langle x_7 \geq 2\rangle} \wedge \overset{@5}{\langle x_6 \geq 2\rangle} \wedge \overset{@1}{\langle x_9 \leq 0\rangle} \implies \bot$$

and replace the propagated atomic constraint with its reason.

In this case, because the reason $\langle x_6 \geq 2\rangle$ already appears in the nogood, this effectively means the atomic constraint $\langle x_7 \geq 2\rangle$ is redundant and can be removed from the nogood, resulting in the new nogood:

$$\overset{@3}{\langle x_3 \geq 1\rangle} \wedge \overset{@3}{\langle x_4 \geq 1\rangle} \wedge \overset{@5}{\langle x_6 \geq 2\rangle} \wedge \overset{@1}{\langle x_9 \leq 0\rangle} \implies \bot.$$

This nogood is now *asserting*: it contains exactly one atomic constraint from the current decision level 5. Consequently, had the nogood been available earlier in the search, it would have propagated before decision level 5. In particular, at decision level 3, it would already have forced $\langle x_6 \leq 1\rangle$, triggering additional propagation and preventing the later conflict. We call these nogoods derived during search **learned nogood**.

The solver can now backjump to decision level 3, since the learned nogood remains violated after backtracking to decision level 4. After backjumping, the learned nogood propagates with the explanation:

$$\overset{@3}{\langle x_3 \geq 1\rangle} \wedge \overset{@3}{\langle x_4 \geq 1\rangle} \wedge \overset{@1}{\langle x_9 \leq 0\rangle} \implies \overset{@3}{\langle x_6 \leq 1\rangle}.$$

Search can now continue from this point with the learned nogood added to the constraint database.

This example illustrates the main idea of conflict analysis. Starting from a conflict, we repeatedly replace propagated atomic constraints by the reasons for their propagation until only a single atomic constraint from the current decision level remains. The resulting asserting nogood captures the underlying cause of the conflict, allowing the solver to both learn from the failure and backjump directly to the decision level responsible for the conflict.

The *conflict analysis* procedure is performed at each conflict. Empirically, it is one of the most effective techniques in modern constraint programming and a defining component of state-of-the-art solvers.

11 Algorithm

We may now describe the general mechanism underlying the conflict analysis procedure illustrated in the previous example. Although the algorithm is closely related to conflict analysis as originally developed in SAT solving (see [**?**] for an overview), we reformulate it using a novel constraint-programming perspective [**?**]. In particular, we reason over atomic

constraints and nogoods rather than propositional variables and clauses. This perspective will also serve as the basis for later sections, where we discuss its benefits and explore recent developments that go beyond traditional nogood learning.

The procedure starts with the conflict nogood. Following the backwards reasoning process illustrated in the previous example, we select the atomic constraint in the nogood that was most recently assigned.

This atomic constraint is then replaced in the nogood by the sufficient condition that implied it, namely, the reason (left-hand side) of its propagation explanation. This results in a new logically implied nogood that is still conflicting.

The replacement process is repeated until the current nogood becomes *asserting*, i.e., until it contains exactly one atomic constraint assigned at the current decision level. At this point, the nogood will propagate after backtracking to an earlier decision level. The remaining atomic constraint assigned at the current decision level is called the *asserting* atomic constraint.

The conflict analysis algorithm can be summarised by the following pseudocode, which is remarkably compact given its substantial impact on solver performance. The algorithm operates on the current trail and assumes that every propagated atomic constraint is associated with an explanation. Given an initial conflict explanation, the procedure derives an asserting nogood.

Algorithm 1 Conflict Analysis

Input: conflict explanation $(A_1 \wedge \cdots \wedge A_n \implies \bot)$
Output: asserting nogood

0: $N \leftarrow$ conflict explanation
0: **while** N is not asserting **do**
0: Let A be the most recently assigned atomic constraint in N
0: Let $R \implies A$ be the explanation of A
0: Replace A in N by R
0: **end while**
0: **return** N =0

After deriving the nogood, the solver backtracks to the decision level at which the nogood propagates, performs the corresponding propagation, and adds the nogood to the constraint database.

We can view the conflict analysis procedure as a form of syntactic rewriting: we iteratively rewrite the conflict nogood until we obtain an asserting nogood. The resulting nogood explicitly encodes the reason for the conflict, allowing the solver to backtrack to the point in the search where this nogood would propagate. This, in turn, enables backtracking across multiple decision levels (*backjumping*).

Because conflict analysis derives new nogoods, it is also referred to as *nogood learning*. Note that "learning" in this context should not be confused with machine learning. A learned nogood is a direct logical consequence of the constraints rather than an approximation inferred from data.

Conflict analysis is valuable for three main reasons.

1. It provides a generic mechanism for combining information from multiple constraints. This can lead to exponential reductions in search effort.

2. Empirically, atomic constraints that frequently appear in conflict analysis make effective branching decisions. This observation forms the basis of modern branching strategies, which we discuss later in the chapter.

3. The set of learned nogoods can serve as a *proof* of infeasibility or optimality, which can be checked independently to certify the correctness of infeasibility claims made by the solver. This will be explored in Chapter 5.

Given the relatively low computational cost of conflict analysis compared to propagation, these benefits have made conflict analysis a defining component of modern constraint solvers.

11.1 Properties of conflict analysis

To understand why conflict analysis works (Section 11), it is useful to examine several key properties of the procedure. These properties show that conflict analysis preserves correctness and is guaranteed to terminate with an asserting nogood.

The properties below are stated with respect to a trail that remains unchanged throughout the conflict analysis procedure. The initial conflict nogood is the conflict explanation that was provided after detecting the conflict. We additionally assume that every propagated atomic constraint has an explanation. This allows conflict analysis to rewrite propagated atomic constraints using their reasons.

Propagation-based rewriting Conflict analysis only rewrites propagated atomic constraints. Atomic constraints introduced as branching decisions are never replaced.

Atomic constraints are processed in reverse order of assignment. Suppose the most recently assigned atomic constraint appearing in the current nogood were a branching decision. Since branching decisions are the first assignments at their decision levels, there could be no other atomic constraints from the same decision level remaining in the nogood. Consequently, the nogood would already be asserting, and the conflict analysis procedure would terminate. Therefore, every atomic constraint selected for rewriting must by necessity be a propagated atomic constraint.

Conflict preservation Every nogood produced during conflict analysis remains conflicting with respect to the current trail.

Indeed, each rewriting step replaces a satisfied atomic constraint with the atomic constraints appearing in its propagation reason. Since all atomic constraints in a valid propagation reason must also be satisfied on the current trail, the resulting nogood remains violated. Therefore, every nogood produced during conflict analysis remains conflicting with respect to the current trail.

Soundness Every nogood produced during conflict analysis is logically implied by the constraints, which include previously learned nogoods.

Each rewriting step replaces an atomic constraint by its reason, i.e., a set of atomic constraints that are a sufficient condition to propagate the replaced atomic constraint. Consequently, every solution violating the rewritten nogood would also violate the original nogood, so the rewritten nogood is a logical consequence of the constraints.

Since conflict analysis consists solely of rewriting steps, every intermediate nogood (including the final learned nogood) is logically implied by the original problem. Therefore, learned nogoods remove no feasible solutions.

Termination Conflict analysis is guaranteed to terminate and does so with an asserting nogood.

In each iteration, the most recently assigned propagated atomic constraint from the conflict nogood is replaced with the atomic constraints appearing in its reason. The reason atomic constraints must have been assigned strictly earlier on the trail; otherwise, the propagation could not have occurred. Consequently, each rewriting step moves strictly backwards through the trail.

Since the trail contains only finitely many atomic constraints, the rewriting process cannot continue indefinitely. Eventually, only a single atomic constraint from the current decision level remains in the nogood, at which point the nogood is asserting and the procedure terminates.

11.2 Learned nogoods and backjumping

Conflict analysis makes the implicit conditions that caused the conflict explicit in the form of a nogood. The solver can therefore *backjump* directly to the point where the nogood can influence the search, rather than backtrack to intermediate decision levels that are already known to lead to failure.

Since an asserting nogood contains exactly one atomic constraint from the current decision level, it determines a unique backjump level. The solver backtracks to the highest decision level appearing in the nogood other than the current one. If no such decision level exists, i.e., the learned nogood contains only a single atomic constraint, the solver backtracks to decision level 0.

After backtracking, all atomic constraints in the learned nogood except the asserting atomic constraint remain satisfied with respect to the trail. Consequently, the learned nogood propagates the negation of the asserting atomic constraint. The solver then performs propagation to a fixed point and continues the search.

The derived nogood is referred to as the **learned nogood**. It is added to the constraint database and may subsequently participate in propagation like any other constraint. Since modern solvers can learn thousands of nogoods per second, learned nogoods are routinely deleted, retaining only those most beneficial to search. We discuss nogood database management in more detail in **??**.

11.3 Learning schemes

The learning procedure described earlier is known as *first unique implication point* (1UIP) learning. Its defining characteristic is that conflict analysis terminates as soon as the current nogood becomes asserting.

However, conflict analysis need not stop at the first asserting nogood. Instead, the rewriting process may continue, leading to alternative learning schemes that differ only in their stopping criteria.

One such scheme is *all-decision learning*. As before, propagated atomic constraints are replaced by their reasons, but the procedure continues until no propagated atomic constraints remain in the nogood. The resulting learned nogood contains only branching decisions. Other learning schemes exist as well, such as *all-UIP learning* [**?**].

Empirical studies have shown that 1UIP learning is generally the most effective learning scheme. Nevertheless, alternative schemes can be useful in particular settings. For example, all-decision nogoods play an important role in optimisation techniques such as *core-guided search*.

12 Verifying the correctness of learned nogoods

When we studied propagation algorithms in Chapter 2, we adopted a healthy scepticism toward implementations. Even if an algorithm is conceptually correct, its implementation may contain subtle errors. To mitigate this, we used *checkers*: whenever a propagator produced an explanation, the checker verified that the explanation was indeed sufficient to justify the propagation.

For learned nogoods, we would like to apply the same principle. Given a learned nogood, we want to be able to verify that it was derived correctly.

Our approach is to check correctness by reconstructing how the nogood was produced.

There are three key ideas:

1. The nogood must be conflicting. If we assume all atomic constraints appearing in the nogood are true, then we expect to reach a conflict.

2. The explanations used during conflict analysis must be sufficient to derive this conflict. In other words, we must be able to follow the same sequence of explanations to reach the conflict.

3. The explanations themselves must be correct. This part was already addressed earlier when discussing explanation correctness, so we now focus on (1) and (2).

We illustrate the procedure on our previous example. For convenience, we reproduce the relevant parts here.

We obtained the following learned nogood:

$$\langle x_3 \geq 1 \rangle \wedge \langle x_4 \geq 1 \rangle \wedge \langle x_6 \geq 2 \rangle \wedge \langle x_9 \leq 0 \rangle \implies \bot.$$

To verify the learned nogood, we begin by assuming that all atomic constraints in the learned nogood are true, which results in the following domains:

$$x_3 \in [1,\infty],\ x_4 \in [1,\infty],\ x_6 \in [2,\infty],\ x_9 \in [-\infty,0].$$

We know that the learned nogood was derived using the following explanations, considered in this order during conflict analysis:

$$\langle x_3 \geq 1\rangle \wedge \langle x_4 \geq 1\rangle \wedge \langle x_7 \geq 2\rangle \wedge \langle x_8 \geq 1\rangle \implies \bot$$

$$\langle x_6 \geq 2\rangle \wedge \langle x_9 \leq 0\rangle \implies \langle x_8 \geq 1\rangle$$

$$\langle x_6 \geq 2\rangle \implies \langle x_7 \geq 2\rangle$$

We now consider the explanations in *reverse order*, beginning with the last one. We first check that the reason holds under the current domains. Since it does, we apply the right-hand side, modifying the previous domains:

$$x_3 \in [1,\infty],\ x_4 \in [1,\infty],\ x_6 \in [2,\infty],\ x_9 \in [-\infty,0],\ x_7 \in [2,\infty].$$

We then move to the next explanation:

$$\langle x_6 \geq 2\rangle \wedge \langle x_9 \leq 0\rangle \implies \langle x_8 \geq 1\rangle$$

Since the reason again holds, we extend the domains:

$$x_3 \in [1,\infty],\ x_4 \in [1,\infty],\ x_6 \in [2,\infty],\ x_9 \in [-\infty,0],\ x_7 \in [2,\infty],\ x_8 \in [1,\infty].$$

Finally, we consider the first explanation, which is a conflict explanation from the original constraints:

$$\langle x_3 \geq 1\rangle \wedge \langle x_4 \geq 1\rangle \wedge \langle x_7 \geq 2\rangle \wedge \langle x_8 \geq 1\rangle \implies \bot$$

Given the current domains, all the reason constraints are satisfied. Thus, a conflict is reached, confirming that the learned nogood is indeed correct. This conclusion assumes, of course, that every explanation used during conflict analysis is itself correct; however, we have already discussed procedures for verifying explanations in Chapter 2, so this requirement is satisfied.

We can now state the general approach. Given a learned nogood, we define a *certificate* of its correctness as a sequence of explanations which, when processed in order, demonstrate that a conflict arises under the assumption that all atomic constraints in the nogood are true. In the previous example, the sequence of explanations we reconstructed served as such a certificate.

To verify a certificate, we begin by constructing initial domains that reflect the atomic constraints in the nogood. We then process the explanations in the certificate one by one. For

each explanation, we check whether its reason (the left-hand side) is true under the current domains. If so, we apply the right-hand side, updating the domains accordingly.

If the certificate is valid, then after applying all explanations, we will derive an empty domain, confirming a conflict. This demonstrates that the nogood indeed captures an inconsistency implied by the problem constraints, and therefore the nogood is correct.

13 Nogood Minimisation

The conflict analysis procedure described earlier may produce learned nogoods that contain redundancies; that is, some atomic constraints may be implied by the conjunction of other atomic constraints already present. This happens because explanations are generated independently by different propagators as part of the essentially syntactic rewrite process of generating the learned nogood.

For instance, suppose conflict analysis produces the nogood

$$\langle x_1 \geq 1\rangle \wedge \langle x_2 \geq 2\rangle \wedge \langle x_3 \geq 3\rangle \implies \bot.$$

If the second atomic constraint, $\langle x_2 \geq 2\rangle$, was itself propagated by a constraint with the propagation explanation

$$\langle x_1 \geq 1\rangle \implies \langle x_2 \geq 2\rangle,$$

then $\langle x_2 \geq 2\rangle$ is redundant in the nogood, as it follows directly from $\langle x_1 \geq 1\rangle$.

Removing such redundant atomic constraints is highly desirable. Smaller nogoods are more general and lead to more propagation. For example, consider domains $x_1 \in [0,2]$, $x_2 \in [0,3]$, and $x_3 \in [3,5]$. The reduced nogood

$$\langle x_1 \geq 1\rangle \wedge \langle x_3 \geq 3\rangle \implies \bot$$

immediately propagates $\langle x_1 \leq 0\rangle$. By contrast, the unreduced nogood

$$\langle x_1 \geq 1\rangle \wedge \langle x_2 \geq 2\rangle \wedge \langle x_3 \geq 3\rangle \implies \bot$$

does not propagate, because the nogood has two unassigned atomic constraints.

Detecting all redundancies inside a nogood is, in general, a hard problem. Instead, as is standard in constraint programming, we seek methods that are computationally inexpensive while still providing meaningful simplification in practice.

The process of removing or rewriting atomic constraints in a nogood to obtain a smaller one is known as **nogood minimisation**. In this chapter, we consider two approaches:

1. removing or rewriting atomic constraints by leveraging the semantics of the atomic constraints, and

2. analysing the current trail to identify explicit implications that show the redundancy of some atomic constraints.

13.1 Semantic Minimisation

The idea behind semantic minimisation is to simplify a nogood by reasoning directly about the semantics of atomic constraints involving a shared variable. Depending on the comparison operators present, it may be straightforward to determine that some atomic constraints are redundant [10]. The following scenarios illustrate the principle.

Consider a nogood of the form

$$\langle x = 5 \rangle \wedge \langle x \neq 3 \rangle \wedge \ldots \implies \bot.$$

Since $\langle x = 5 \rangle$ logically implies $\langle x \neq 3 \rangle$, the latter is redundant and can be removed.

Similarly, consider the nogood

$$\langle x \geq 5 \rangle \wedge \langle x \geq 10 \rangle \wedge \ldots \implies \bot.$$

Because $\langle x \geq 10 \rangle$ implies $\langle x \geq 5 \rangle$, the constraint $\langle x \geq 5 \rangle$ is unnecessary and may be eliminated.

Semantic reasoning also enables rewriting. For instance,

$$\langle x \leq 5 \rangle \wedge \langle x \neq 5 \rangle \wedge \ldots \implies \bot$$

can be replaced by

$$\langle x \leq 4 \rangle \wedge \ldots \implies \bot,$$

because any value satisfying $\langle x \leq 4 \rangle$ automatically satisfies both $\langle x \leq 5 \rangle$ and $\langle x \neq 5 \rangle$.

Another common scenario arises when combining upper and lower bounds. From

$$\langle x \leq 10 \rangle \wedge \langle x \geq 10 \rangle \wedge \ldots \implies \bot,$$

we may rewrite the nogood as

$$\langle x = 10 \rangle \wedge \ldots \implies \bot,$$

since $\langle x = 10 \rangle$ is equivalent to the conjunction of the two bounds.

These simplifications rely on high-level semantic understanding of the atomic constraints. This stands in contrast to trail-based minimisation, which reasons generically using only information present on the trail, which we discuss below.

Although the preceding examples present semantic minimisation as a system of rewrite rules, the actual implementation is far more efficient. Instead of applying individual rewrites, we construct the domain implied by all atomic constraints involving the same variable and then describe that resulting domain using one or more atomic constraints. This eliminates the need to enumerate specific rewrite rules, leading to a much simpler implementation.

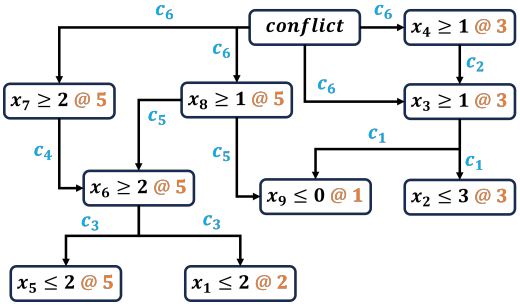
Semantic minimisation is applied to the final learned nogood. Because it is computationally inexpensive and results in smaller nogoods, it is a valuable addition to conflict analysis.

13.2 Trail-Based Minimisation

An atomic constraint may be implied by other constraints in the problem, given the current trail. Rather than querying individual constraints for redundancy, we may *greedily* use the implication structure present in the trail itself [20].

The trail can be represented as an **antecedent graph**, where each node corresponds to an atomic constraint, and there is an edge (a_1, a_2) whenever atomic constraint a_2 is part of the reason for the propagation of atomic constraint a_1. Thus, (a_1, a_2) indicates that "a_1 has a_2 as an antecedent". Decision atomic constraints have no outgoing edges, and edges are labelled with the constraint that caused the propagation. Analysing this graph allows us to detect implications that can be used to simplify learned nogoods.

$\langle x_9 \leq 0 \rangle$ @1
$\langle x_1 \leq 2 \rangle$ @2
$\langle x_2 \leq 3 \rangle$ @3
$C_1 \rightarrow \langle x_3 \geq 1 \rangle$
$C_2 \rightarrow \langle x_4 \geq 1 \rangle$
$\langle y \geq 2 \rangle$ @4
$\langle x_5 \leq 2 \rangle$ @5
$C_3 \rightarrow \langle x_6 \geq 2 \rangle$
$C_4 \rightarrow \langle x_7 \geq 2 \rangle$
$C_5 \rightarrow \langle x_8 \geq 1 \rangle$
$C_6 \rightarrow$ conflict



Trail and corresponding antecedent graph. The notation "A @ X" indicates decision level X for atomic constraint A. Labelled arrow $C_i \rightarrow A$ indicates that A was propagated by constraint C_i. An edge (A_1, A_2) means A_1 has A_2 as an antecedent.

We use the antecedent graph to detect redundant atomic constraints in a learned nogood. An atomic constraint is redundant if the graph shows that it is implied by other constraints already in the nogood.

A few special situations illustrate when redundancy *can* or *cannot* be concluded from the trail:

- If two atomic constraints a_1 and a_2 appear in the nogood and a_1 has only one outgoing edge (a_1, a_2), then a_1 is implied by a_2 and may be removed. For example, in the graph above, $\langle x_4 \geq 1 \rangle$ is implied by $\langle x_3 \geq 1 \rangle$.

- If an atomic constraint in the nogood is a *decision* assignment, then nothing in the antecedent graph can prove it redundant.

- If an atomic constraint a_1 in the nogood has an antecedent that is a decision constraint not appearing in the nogood, redundancy cannot be concluded.

- If a_1 has an antecedent from a decision level not represented in the nogood, redundancy also cannot be concluded.

These examples motivate the general principle: an atomic constraint a_i in a learned nogood is *redundant* if and only if *all paths* starting from a_i in the antecedent graph eventually reach only atomic constraints already in the nogood. Because the graph is acyclic, this leads to a linear-time recursive redundancy check for each atomic constraint.

A naïve implementation repeats work because many subgraphs are visited multiple times. To avoid this, we cache results using four labels for atomic constraints: *keep*, *redundant*, *poison*, and *seen*. These labels guide the recursion:

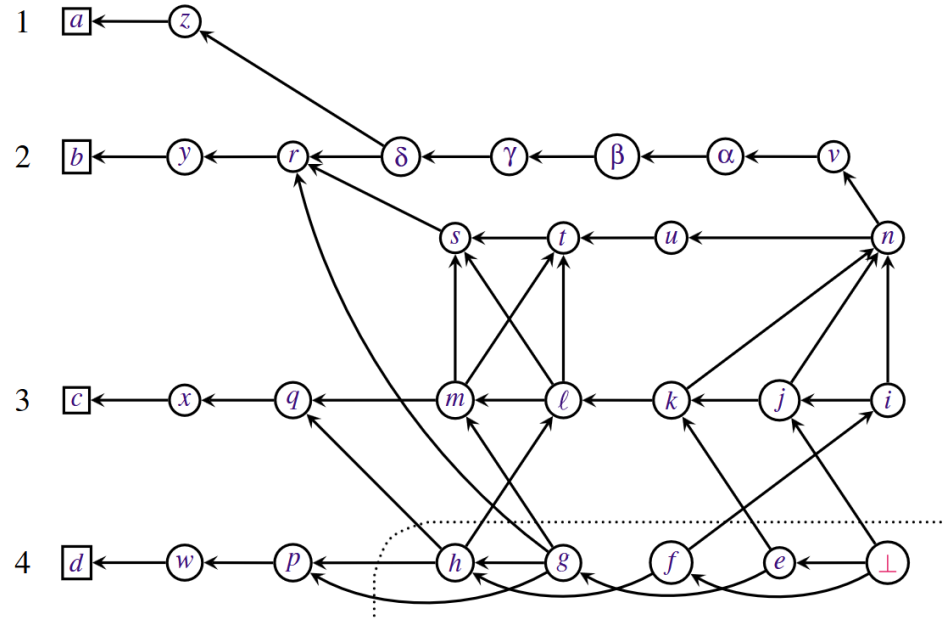
- The asserting atomic constraint is labelled *keep* since it cannot be redundant by definition.
- Atomic constraints in the nogood that are decisions are labelled *keep*, because we cannot conclude redundancy about decisions from the implication graph.
- Decision atomic constraints not in the nogood are labelled *poison*, since the graph lacks enough information about them.
- Atomic constraints from decision levels not represented in the nogood are labelled *poison*, since we cannot conclude redundancy for these atomic constraints.
- Similarly, an atomic constraint constraint becomes *poison* if it reaches a *poison* node.
- The exception to the above are atomic constraints from the initial nogood, which become *keep* if they reach any *poison* node. This means that it has at least one atomic constraint in its (in)direct antecedents, which cannot be used to reason about redundancy, and as a consequence, we cannot conclude redundancy for this atomic constraint.
- Propagated atomic constraints appearing in the nogood (except the asserting one) start as *seen*, a temporary label to distinguish unlabelled atomic constraints that are not in the initial nogood.
- A *seen* atomic constraint is relabelled *keep* if it reaches a *poison* node.
- A *seen* constraint is relabelled *redundant* if all reachable nodes are labelled *keep* or *redundant*.

The recursive algorithm starts from each atomic constraint labelled *seen* and uses the above definitions to compute its final label. Once all *seen* atomic constraints have been relabelled, those marked *redundant* are removed from the nogood. In practice, atomic constraints are labelled as they arise during recursion, and the graph is never explicitly stored. To limit deep recursion, any constraint more than k steps away from the nogood (e.g., $k = 500$) is labelled *poison*.

To illustrate the procedure in detail, consider the antecedent graph in the figure below together with the corresponding learned nogood

$$p \wedge q \wedge r \wedge m \wedge l \wedge k \wedge j \wedge i \implies \bot,$$

obtained using the 1UIP scheme, where p is the asserting atomic constraint.



An antecedent graph where atomic constraints at the same decision level appear on the same horizontal line. The conflict nogood can be identified by the outgoing edges from the conflict node. Dotted lines show the cut produced in the graph by the learned nogood produced by the 1UIP scheme $(p \wedge q \wedge r \wedge m \wedge l \wedge k \wedge j \wedge i \implies \bot)$*, with p as the asserting atom. Adapted from a paper by Van Gelder [20].*

The algorithm begins by labelling p as *keep*, since it is the asserting atomic constraint. All other atomic constraints in the learned nogood (q, r, m, l, k, j, i) are initially labelled *seen*.

For readability, we adopt the following shorthand: "a (label)" means that recursion reaches atomic constraint a and stops because it already has a label; "$a \downarrow$ (label)" means that recursion both reaches a and assigns it the label indicated.

A possible (valid) execution of the algorithm proceeds as follows.

1. Processing q: The algorithm starts with the *seen* atomic constraint q. It explores its antecedents in the graph:

- q - x - $c \downarrow$ (poison) - $x \downarrow$ (poison) - $q \downarrow$ (keep)

Here, c is labelled *poison* because it is a decision atomic constraint outside the nogood; this forces x to also become *poison*. As q reaches a poison node, it is labelled *keep*. This recursive call finishes.

2. Processing m: Next, the algorithm picks another *seen* atomic constraint, m:

- m - q (keep) - s - r - y - $b \downarrow$ (poison) - $y \downarrow$ (poison) - $r \downarrow$ (keep) - $s \downarrow$ (redundant) - t - s (redundant) - $t \downarrow$ (redundant) - $m \downarrow$ (redundant)

Thus, m is determined to be *redundant*. The recursion terminates for this branch.

3. Processing k: The algorithm moves on to another *seen* constraint, k:

- k - l - m (redundant) - t (redundant) - s (redundant) - $l\downarrow$ (redundant) - n - v - α - β - γ - δ - r (keep) - $z\downarrow$ (poison, decision level not present in nogood) - $\delta\downarrow$ (poison) - $\gamma\downarrow$ (poison) - $\beta\downarrow$ (poison) - $\alpha\downarrow$ (poison) - $v\downarrow$ (poison) - $n\downarrow$ (poison) - $k\downarrow$ (keep)

Because k eventually reaches a *poison* node, it is labelled *keep*. This recursive call finishes.

4. Processing j: The next *seen* constraint is j:

- j - n (poison) - $j\downarrow$ (keep)

Thus, j becomes *keep*.

5. Processing i: Finally, the algorithm processes i:

- i - j (keep) - n (poison) - $i\downarrow$ (keep)

At this point, no atomic constraints remain with the *seen* label, so the algorithm terminates. The atomic constraints labelled *redundant* are m, l. These may be safely removed from the learned nogood.

Although different traversal orders are possible, they all lead to the same final classification of redundant atomic constraints.

Trail-based minimisation is standard in SAT solvers and often very effective, routinely removing many redundant constraints. In constraint programming, its impact varies across problem instances. Semantic minimisation quickly removes many obvious redundancies, leaving trail-based minimisation to detect the deeper, implicit ones encoded in the structure of the trail.

14 Nogood Propagation Algorithms

Since the solver may accumulate a large number of nogoods during search, it is essential that the associated propagation algorithms are highly efficient.

Nogood propagation in constraint programming is inspired by clausal propagation in SAT solvers [16], which operate over propositional variables and clauses. The SAT community has developed exceptionally efficient propagation algorithms, as clause-based reasoning is central to SAT solving.

To simplify the exposition, let us assume the SAT setting, where all variables are binary. In this case, atomic constraints simply correspond to assignments, namely $\langle x = 0 \rangle$ and $\langle x = 1 \rangle$. We will later briefly discuss how the approach presented below can be generalised to constraint programming.

Recall that a nogood propagates when all but one of its atomic constraints are true; in that situation, the remaining unassigned atomic constraint must be set to false. This is known as **unit propagation**.

Adjacency-list propagation. A straightforward approach is to maintain, for each atomic constraint, a list of nogoods in which it appears. Whenever the atomic constraint becomes true, the solver inspects all of its nogoods to determine whether unit propagation can occur. We refer to this as the *adjacency-list* approach.

Counter-based propagation. A more refined alternative is to maintain, for each nogood, a counter recording how many of its atomic constraints are unassigned, together with a flag indicating whether any of them is currently assigned false. Propagation occurs precisely when the counter reaches one and no atomic constraint is false. We call this the *counter-based* approach. Its advantage is that the counters and flags can be maintained incrementally as the search progresses, often avoiding redundant scans that occur with the adjacency-list approach.

However, the counter-based approach has a non-trivial drawback: the counters and flags must be updated when the solver backtracks. In contrast, the adjacency-list approach has static data structures that remain valid across backtracking. Although the counter-based scheme may reduce work on average, the overhead of maintaining its data structures may be wasted if a nogood never propagates.

Two-watched propagation. To avoid maintenance work, we adopt a *lazy* strategy that only performs detailed checks when propagation is genuinely plausible. A key observation is that a nogood can only propagate or become conflicting if it has at most one unassigned atomic constraint. As long as two or more atomic constraints remain unassigned, propagation is not possible.

This motivates the **two-watcher scheme**, which assigns two distinguished atomic constraints per nogood—its *watchers*. For each atomic constraint, the solver maintains a *watched list* of nogoods that currently watch it. The scheme maintains the invariant that each nogood always has two watchers, and each watcher is either unassigned or assigned false. In this state, the nogood cannot propagate.

When a watcher becomes true, it can no longer serve as a watcher. The solver must attempt to replace it with another atomic constraint from the nogood. If such a replacement is found, the nogood remains non-propagating. If no replacement exists, because all remaining atomic constraints are either true or are already serving as watchers, then the nogood must either propagate (if exactly one atomic constraint remains unassigned, which corresponds to the other watcher) or report a conflict (if none remain unassigned).

A distinctive advantage of the two-watcher scheme is that it is *backtrack-free*: backtracking does not require restoring or updating any of the watcher structures, as the invariants continue to hold when assignments are undone. This makes the scheme extremely attractive in practice.

The two-watcher scheme is a prime example of a lazy propagation algorithm. In the worst case, it may perform more work than the counter-based approach, but in typical cases where nogoods contain many atomic constraints, it provides a much more efficient propagation mechanism.

Two-watchers for atomic constraints. The discussion above assumed that all variables were binary, making it straightforward to determine when an atomic constraint becomes true.

To generalise this idea for constraint programming, most solvers in the literature ("lazy clause generation" solvers) maintain a dual representation of the problem: a CP view and a SAT view. This dual representation introduces a Boolean variable for each atomic constraint together with a dedicated channeling mechanism that synchronises the truth value of the Boolean variable with the status of the corresponding atomic constraint. As a result, classical

SAT-style two-watcher propagation can be applied directly by watching the Boolean variables that encode the atomic constraints.

In our solver, *Pumpkin*, we take a different approach. Instead of relying on SAT variables, we use a specialised notification mechanism that triggers whenever an atomic constraint in the nogood database becomes true. Despite this architectural difference, the logic of the two-watcher scheme remains similar.

15 Conflict-Driven Variable Selection (VSIDS)

The *fail first* principle has long guided the design of variable-selection strategies: it encourages choosing branching variables that are likely to cause conflicts early. The intuition is that failing sooner leads to stronger propagation and more pruning. There are many ways to operationalise this idea; here we focus on an approach that works hand-in-hand with conflict analysis. After outlining several conceptual issues, we present the *VSIDS* strategy, originally developed for SAT solving [17] and later adopted in constraint programming.

A central difficulty is that being "active in conflicts" is not well-defined. One possibility is to treat a variable as active if it appears in any propagator that reports a conflict. However, this counts variables that did not actually contribute to the conflict, lowering the precision of the activity measure.

A second difficulty concerns the interpretation of "often". A straightforward approach is to associate a counter (the *activity*) with each variable and increment it whenever the variable is involved in a conflict. Branching would then select the variable of highest activity. But such an approach fails to consider that variables active in a distant part of the search tree may no longer be relevant; conversely, variables that have recently appeared in conflicts should be emphasised. A pure counting mechanism does not distinguish these cases.

Conflict analysis provides a more accurate notion of activity. A variable can be considered active in a conflict if it appears in the conflict explanation. Since this misses variables implicitly responsible for the conflict via chains of propagation, we broaden the definition to include any variable that participates in the conflict analysis process, either within the initial conflict nogood or any propagation explanation. Furthermore, rather than recording conflict variables, we may want to track which specific atomic constraints were involved in conflicts. For each atomic constraint encountered in an explanation during conflict analysis, we increase its activity by one.

To address the second concern, we apply an exponential decay to activities. In each step (e.g., after every conflict), the activity of every atomic constraint is multiplied by a constant $\beta \in (0,1)$, such as 0.95 or 0.99. This sharply decreases the influence of atomic constraint that were not recently active, ensuring that the most recent conflicts dominate the activity scores.

Applying decay naively may be expensive if the problem contains many variables. Instead, solvers use an equivalent and more efficient mechanism: instead of decaying all activities, they *increase the increment amount* used when updating atomic constraint activities. After each conflict, this increment is multiplied by a constant $\gamma > 1$, e.g., $\gamma = 1.01$. This produces similar behaviour as decay. To prevent numerical overflow, the increment is

occasionally reset and all activities are divided by a large constant. Only the *relative ordering* of activities matters.

When branching, the solver chooses the atomic constraint with the highest activity score. This strategy is called **VSIDS (Variable State Independent Decaying Sum)**.

Two caveats should be kept in mind. First, all atomic constraints start with an activity of 0. With random tie-breaking, the early part of the search behaves almost randomly until enough conflicts have been encountered to differentiate variables. Since early decisions heavily influence the search, this may be undesirable. Second, if the solver enters a poor region of the search space, improvements may be slow: VSIDS focuses on resolving local conflicts, even if a much better region of the search space lies elsewhere.

Two approaches alleviate these drawbacks. The first is to rely on *restarts* (see next section) to escape from poor regions of the search space. The second is to use a domain-specific branching strategy for a short period to initialise the activity profile before switching to VSIDS, e.g., until the first solution is found or until a certain number of conflicts has occurred. Another option is to alternate between VSIDS and a domain-specific heuristic.

Despite these subtleties, VSIDS has been empirically shown to be very effective and is widely used as a robust, domain-independent branching strategy.

16 Learned Nogood Management

Learned nogoods are valuable because they capture interactions between constraints that are not explicitly enforced elsewhere in the model. Each conflict produces a new learned nogood, and we add every nogood to the constraint database so it can contribute to future propagation.

However, modern solvers may generate thousands of conflicts per second. Keeping *all* learned nogoods would quickly exhaust memory, and even before reaching memory limits, maintaining and propagating a very large set of nogoods would slow down the solver significantly. We must therefore be selective about which nogoods to keep and which to delete.

Since all learned nogoods are logical consequences of the problem constraints, deleting them never removes any feasible solution. But deletion means a nogood might be learned again later.

Intuitively, not all nogoods are equally valuable. Ideally, one wants to keep nogoods that propagate frequently. Unit nogoods, containing a single atomic constraint, are especially useful because they propagate at the root level. In contrast, a large nogood involving almost all the problem variables is unlikely to be of use again.

As with propagation algorithms, we must balance propagation strength with propagation speed. Storing too many nogoods slows the solver; storing too few leaves it susceptible to repeating poor choices.

A simple policy is to periodically delete learned nogoods, keeping only those deemed *most useful*. For our purposes, we assume the solver maintains a fixed number of learned nogoods and, at predefined intervals, deletes half of them. More sophisticated schemes exist [18], but this captures the essential idea.

Assessing the quality of a learned nogood is inherently heuristic. We discuss three commonly used metrics.

16.1 Metrics for Learned Nogoods

Nogood Size. A natural first metric is the *size* of the nogood: the number of atomic constraints it contains. Unit nogoods are the strongest; binary nogoods are also very useful.

This metric is simple and cheap to compute. However, the size may not reflect true strength for larger nogoods because of correlations among the atomic constraints.

For instance, consider the nogood

$$\langle x_1 \geq 1\rangle \wedge \langle x_2 \geq 2\rangle \wedge \langle x_3 \geq 3\rangle \longrightarrow \bot,$$

and suppose the problem constraints enforce

$$\langle x_1 \geq 1\rangle \longrightarrow \langle x_2 \geq 2\rangle .$$

Then setting $\langle x_1 \geq 1\rangle$ implicitly enforces $\langle x_2 \geq 2\rangle$, so the nogood effectively behaves like a nogood of size two rather than three. Thus nogood size can be misleading.

Literal Block Distance (LBD). This motivates the **LBD** metric (*Literal Block Distance*) [3], originally from SAT solving. Instead of counting the number of atomic constraints in a nogood, we count how many *distinct decision levels* appear among the atomic constraints of the nogood.

This captures its intrinsic strength: a nogood spanning many decision levels is intuitively less likely to propagate again, while a nogood concentrated on few levels is more likely to propagate.

Computing an exact LBD in constraint programming is difficult, and its meaning evolves over time as new nogoods are learned or deleted. We therefore use an approximation.

When a nogood is first learned, all of its atomic constraints have assignments. Its initial LBD is the number of distinct decision levels appearing in the nogood (excluding the current decision level). Each time the nogood participates in conflict analysis, either as the conflict nogood or as part of an explanation, we recompute its number of decision levels. If the new value is lower, we update the stored LBD. Thus, the LBD of a nogood records the smallest number of decision levels ever observed for that nogood when it was active.

LBD has become the *de facto* standard criterion for nogood quality.

Nogood Activity. A third metric, useful as a tie-breaker when several nogoods share the same LBD, is **activity**. Nogoods that repeatedly participate in conflict analysis are presumably useful and should be preserved.

Each nogood maintains an activity value, incremented every time it contributes to conflict analysis. This mechanism mirrors the VSIDS heuristic for variable activity.

These three metrics—LBD, activity, and occasionally size—guide the solver in deciding which nogoods to retain and which to discard. A common policy is to periodically delete

half of the stored nogoods, keeping those with the lowest LBD and highest activity. Nogoods with LBD equal to two are often kept permanently, as they are typically extremely powerful, although other learned nogood management strategies exist [18].

17 Restarts

The behaviour of a search procedure is heavily influenced by the sequence of decisions it makes. An unfortunate set of early decisions may lead the solver into a subproblem with no feasible solutions, where proving infeasibility is difficult. In contrast, a different sequence of decisions for the same problem instance may lead to a quick solution. This phenomenon, where some branches of the search tree require vastly more work than others, is known as *heavy-tail behaviour*.

To mitigate heavy-tail behaviour, the solver must detect it during runtime and *undo* early decisions. We first focus on undoing decisions, and then discuss how to identify heavy-tail behaviour.

Traditionally, undoing decisions is only possible after fully exploring a subtree or via backjumping triggered by conflict analysis. While backjumping can help, it only addresses part of the problem: conflict analysis still operates within the current region of the search space, which may itself be unproductive.

A complementary technique is to periodically *restart* the search by backtracking to the root level. For restarts to be effective, we must ensure the solver does not simply reproduce the same search path. Conflict analysis provides this mechanism.

When conflict learning is enabled, restarting is not the same as beginning the search anew. Learned nogoods and any activity scores (e.g., in VSIDS) remain in the solver. Variables that were active in recent conflicts, which were previously located near the bottom of the search tree, are promoted to earlier decision positions after the restart. This reshapes the search tree and typically results in exploring different regions after each restart.

Empirically, restarts are a crucial component of modern solvers. Their theoretical foundations are less well understood, though the intuition above provides some justification.

The main practical question is how often to restart.

17.1 Restart Strategies

Restarting too frequently wastes useful search effort, while restarting too infrequently increases the risk of heavy-tail behaviour. A suitable restart strategy must balance these concerns.

The simplest strategy is to restart every k conflicts, for a chosen constant k such as 256, 512, 1024, or 2048. This strategy, known as **constant restarts**, is easy to implement and performs well in practice. The underlying hope is that conflict learning eventually guides the solver toward a solution.

A natural extension increases the restart interval by a fixed percentage after each restart (e.g., by 10%). This allows the solver to gradually reach deeper areas of the search tree. This approach is called **geometric restarts**.

A third option combines short and long intervals in a structured way. A widely used example is the **Luby restart sequence**, which generates restart intervals by the pattern

$$1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots$$

multiplied by a constant factor (e.g., 256). The pattern is recursive: the initial sequence "1" is repeated twice, followed by "2"; this entire block is repeated twice, followed by "4"; and so on. The strength of the Luby sequence is that it provides a robust mix of short and long restart intervals, making it effective across a broad range of problem types. In particular, Luby restarts have been shown to perform well on feasible instances.

All these strategies are static: the restart schedule does not adapt to the state of the solver. A more recent and influential strategy incorporates dynamic behaviour.

One of the popular dynamic approaches is known as **Glucose restarts** [4], named after the SAT solver in which it was introduced. This strategy regularly tests whether the solver should restart, but may *block* a restart if the solver appears to be in a promising part of the search space.

The solver restarts after a fixed number of conflicts unless the restart is blocked. In the original Glucose solver, a restart is blocked when the solver appears to be *close* to finding a solution. This is detected by comparing the current number of assigned variables to a moving average; if the current number is significantly higher, the solver delays restarting. In constraint programming, an analogous measure would be an unusually large number of pruned domain values.

The solver also identifies whether it is in a *poor* region of the search space by monitoring the average LBD of recently learned nogoods. If the recent average is significantly lower than the global average, then restarts are blocked because learning appears to be unusually effective.

The Glucose strategy balances these two signals, dynamically delaying or allowing restarts, and has proven effective on both feasible and infeasible instances. Variants of this approach are widely used in modern solvers.

Chapter 5

Certificates

After a solver reports that a problem is feasible, we may request the solution and verify it. Checking that all constraints are satisfied is straightforward, and for optimisation problems, we can also confirm that the objective value matches the value reported by the solver. In short, feasibility claims are easy to verify.

The situation changes drastically when a solver claims *infeasibility*, that is, that no solution exists for the given instance. How can we verify such a claim? Similarly, how can we confirm that an optimisation result is indeed optimal? Ideally, the solver would provide evidence for these claims in such a way that we could verify *independently* of the solver. Since infeasibility and optimality claims are properties of the *problem instance*, their verification should not require trusting the solver that produced the result.

Verifying solver claims is essential for increasing trust in their results. Solvers are complex systems, and it is reasonable to suspect that an infeasibility claim may stem from an implementation error or a conceptual flaw in a propagator. A faulty propagator might prune the search space too aggressively, causing the solver to incorrectly conclude that no solution exists.

These concerns arise naturally in high-stakes applications. Suppose you design a new digital circuit that is functionally equivalent to an existing model but uses fewer gates. Before deploying it, you must ensure that both circuits behave identically for all possible inputs, as even a single discrepancy could be catastrophic. Checking functional equivalence reduces to a constraint satisfaction problem: find an input for which the two circuits differ. If the solver claims the problem is infeasible, it asserts that no such input exists and the circuits are equivalent. Consequently, the correctness of the infeasibility claim becomes critical: an unsound propagation algorithm could erroneously eliminate a valid counterexample, leading to a false claim of equivalence.

Similar concerns arise when verifying optimality claims. Consider, for example, a combinatorial auction, where bidders submit bids on *sets* of items. As the auctioneer, your goal is to select a set of non-conflicting bids that maximises revenue. If you rely on a solver to determine the winning bids, you must be confident that the reported optimum is truly optimal. A single solver bug could invalidate the entire auction outcome, potentially leading to serious legal or financial consequences.

These examples generalise to many other domains: security protocols, aircraft control

systems, kidney exchange, AI systems, and more. Such problems are embedded in broader socio-technical contexts, but even within that larger picture, the technical question of solver correctness remains fundamental: solvers drive critical decisions, and unreliable results are unacceptable.

Unfortunately, even heavily tested software can contain bugs. Indeed, numerous studies document errors in state-of-the-art optimisation tools. Fuzz testing [19] revealed bugs in nearly all solvers participating in the MaxSAT Evaluation 2022. Similar issues arise in SAT and SMT solvers [7], constraint programming solvers [2, 1], and long-standing MIP solvers as incorrect answers may arise due to numerical instability [9]. A quick look at the issue tracker of any open source project reveals a steady stream of bug reports.

Given the complexity of modern solvers, scepticism about correctness is justified. Although software engineering techniques help reduce errors, they cannot *prove the absence* of bugs. Verification aims to provide such guarantees, but current methods cannot yet produce verified implementations with performance comparable to leading solver technology.

Instead of proving the correctness of solver implementations, we shift focus to verifying the *output* they produce. For feasible instances, checking a solution is easy and requires no further discussion. The real challenge concerns infeasibility and optimality claims. Rather than accepting such claims at face value, we require the solver to produce a *certificate* (also called a *proof*) that can be checked both *efficiently* and *independently* of the solver that provided the certificate. Given our sceptical position, we demand tangible evidence.

We have already adopted this sceptical perspective in previous chapters through checkers for propagators and conflict analysis. However, such checks remain internal to the solver and therefore cannot serve as independent evidence for an external user.

We now take this perspective a step further by formalising the behaviour of the solver as *an algorithm that derives mathematical proofs*, rather than viewing it purely as a backtracking search procedure. Viewed in this way, every step taken by the solver must correspond to a valid mathematical reasoning step. This motivates the introduction of a formal framework in which the solver operates: a *proof system*. This framework allows us to view solver execution as the construction of proofs that certify infeasibility or optimality, while also providing a foundation for analysing the computational limits of solver reasoning.

The previous chapter already introduced the key ingredients needed for certification. Through conflict analysis, the solver derived learned nogoods, each of which was a logical consequence of the constraints. By accumulating enough nogoods, the solver could eventually encounter a conflict without making any decisions and conclude that the problem is infeasible. Although we viewed this process primarily as a search procedure, it can equally be viewed as a sequence of logical derivations.

In this chapter, we make this perspective explicit. We formalise the derivation of learned nogoods as proof steps and show how the resulting derivations can be recorded, verified, and used as certificates of infeasibility and optimality. We will also see how different conflict-analysis procedures naturally give rise to different proof systems.

The key advantage of producing certificates is that verifying a proof is dramatically simpler than verifying the solver that constructed it. We have already encountered initial forms of this idea through checkers. Because proof checking is so lightweight, we can even apply formal verification techniques to ensure the checker itself is correct [**?**], e.g., using

interactive theorem provers such as Rocq. While this does not show that the solver is free of bugs, it significantly increases our confidence that, for a given instance, the claim made by the solver is justified.

In this chapter, we study solver behaviour through the lens of formal proofs. We begin by introducing a proof system that models the logical reasoning steps performed by the solver. We then explain how proofs in this system can be verified efficiently and independently. Finally, we examine how such proofs can be produced during solving. We also touch upon stronger proof systems that, while more complex, can result in exponentially shorter proofs, although it remains an open research question how to exploit these theoretical advantages in practice.

We will first illustrate these ideas through an extended example to build intuition, and then formalise the underlying concepts.

18 Illustrative Example: Proof

Before we formally introduce our proof system, we first work through a concrete example. This example motivates the formalisation and will be referenced throughout the next sections.

Consider the following CSP:

$$\begin{aligned} c_1 &: 5x + 3y \leq 6, \\ c_2 &: x + y + z \geq 3, \\ & x \in \{0,1,2\}, \\ & y,z \in \{0,1\}. \end{aligned}$$

The CSP is infeasible, and our goal is to obtain a certificate that demonstrates this fact. Such a certificate should be straightforward to read, making it immediately clear why the problem has no solution. There are naturally many ways to prove that our CSP is infeasible. We discuss a recent approach tailored to how constraint programming solvers work [**?**].

The proof is given in the table below and should be read from top to bottom. Each line derives a constraint in the same form as the explanations we have seen earlier,

$$a_1 \wedge \cdots \wedge a_{n-1} \implies a_n,$$

where each a_i is an atomic constraint. As before, the left-hand side is called the *premise* (or reason), and the right-hand side is the *consequent*. Within the context of our proof system, we refer to such derived constraints as *facts* rather than explanations. The term "explanation" is reserved for justifications generated during search, which must satisfy both *validity* and *applicability*. In contrast, "facts" belong to our mathematical proof system and are required only to satisfy *validity*.

The proof is organised into blocks, called *proof steps*, each of which combines several inferences to derive a new fact. These derived facts play a role analogous to learned nogoods in a solver, whereas the individual inferences correspond to propagation explanations. The analogy is not exact, however, since proofs in our system do not involve search. The final proof step derives the implication $\top \implies \bot$, establishing that the original CSP is infeasible.

ID	Derived constraint			
c_3	c_1	$\vDash$	$\langle y \geq 0 \rangle \implies \langle x \leq 1 \rangle$	{Linear bounds}
c_4	c_1	$\vDash$	$\langle x \geq 1 \rangle \implies \langle y \leq 0 \rangle$	{Linear bounds}
c_5	c_2	$\vDash$	$\langle x \leq 1 \rangle \wedge \langle z \leq 1 \rangle \implies \langle y \geq 1 \rangle$	{Linear bounds}
c_6	$c_3 \wedge c_4 \wedge c_5$	$\vDash$	$\langle x \geq 1 \rangle \wedge \langle y \geq 0 \rangle \wedge \langle z \leq 1 \rangle \implies \bot$	{Deduction}
c_7	c_6	$\vDash$	$\langle y \geq 0 \rangle \wedge \langle z \leq 1 \rangle \implies \langle x \leq 0 \rangle$	{Nogood}
c_8	c_2	$\vDash$	$\langle y \leq 1 \rangle \wedge \langle z \leq 1 \rangle \implies \langle x \geq 1 \rangle$	{Linear bounds}
c_9	$c_7 \wedge c_8$	$\vDash$	$\langle y \geq 0 \rangle \wedge \langle y \leq 1 \rangle \wedge \langle z \leq 1 \rangle \implies \bot$	{Deduction}
c_{10}	$\top$	$\vDash$	$\top \implies \langle y \geq 0 \rangle$	{Problem domains}
c_{11}	$\top$	$\vDash$	$\top \implies \langle y \leq 1 \rangle$	{Problem domains}
c_{12}	$\top$	$\vDash$	$\top \implies \langle z \leq 1 \rangle$	{Problem domains}
c_{13}	c_9	$\vDash$	$\langle y \geq 0 \rangle \wedge \langle y \leq 1 \rangle \wedge \langle z \leq 1 \rangle \implies \bot$	{Nogood}
c_{14}	$c_{10} \wedge c_{11} \wedge c_{12} \wedge c_{13}$	$\vDash$	$\top \implies \bot$	{Deduction}

We begin with the first proof step, which starts with the derivation of the fact c_3. The fact

$$\langle y \geq 0 \rangle \implies \langle x \leq 1 \rangle$$

follows from the original constraint $c_1 : 5x + 3y \leq 6$ using linear bounds reasoning, the same reasoning used by linear inequality propagators. Under the assumption $y \geq 0$, any assignment with $x \geq 2$ violates fact c_1. Hence, the implication $x \leq 1$ is a valid consequence of the constraint, and c_3 is a valid fact.

Although the initial domain already satisfies $y \geq 0$, the premise is explicitly included when deriving fact c_3. This is a deliberate design choice that keeps the proof system explicit and easier to inspect: every assumption required to justify a derived fact must appear in the premise. Bound reasoning does not rely on domain restrictions, so initial domain facts must be included wherever they are needed by the inference. Note that in this case, fact c_3 holds regardless of the initial domains of the variables.

The same reasoning applies to facts c_4 and c_5. Having established those facts, we may now combine them to derive a new fact.

The fact c_6 combines c_3, c_4, c_5 to conclude:

$$\langle x \geq 1 \rangle \wedge \langle y \geq 0 \rangle \wedge \langle z \leq 1 \rangle \implies \bot.$$

Verifying this uses the same logic as verifying learned nogoods. We construct a domain object $\mathcal{D}_{\mathcal{A}}$ based on the premise $\mathcal{A} = \{\langle x \geq 1 \rangle, \langle y \geq 0 \rangle, \langle z \leq 1 \rangle\}$ of the derived fact. Initially:

$$\mathcal{D}_{\mathcal{A}}(x) = [1, \infty), \quad \mathcal{D}_{\mathcal{A}}(y) = [0, \infty), \quad \mathcal{D}_{\mathcal{A}}(z) = (-\infty, 1].$$

We then apply facts c_3, c_4, and c_5 in order with the expectation of encountering a contradiction.

Processing fact c_3, we observe that its premise is satisfied, so we add its consequent $\langle x \leq 1 \rangle$, tightening x's domain to $[1, 1]$. The same holds for c_4, tightening y's domain to $[0, 0]$.

Finally, c_5 also applies, and its consequent forces $y \geq 1$, producing an empty domain. This shows the premise of c_6 is inconsistent, so the fact is valid.

The proof steps deriving c_9 and c_{14} follow the same overall pattern as the earlier steps, but introduce two new elements. First, the fact c_7 is obtained by rewriting the fact c_6 using the nogood rule. Second, in the derivation of c_{14}, we explicitly use the initial domains of the variables as specified in the CSP. These domain facts, together with the intermediate nogood c_{13}, allow us to conclude c_{14}, which states that infeasibility follows without requiring any premises.

When a solver reports infeasibility for our example CSP, it may also provide a certificate such as the one above. The certificate may not need to be unique; for example, there are different ways to show that the above CSP is infeasible.

Although the certificate is designed to capture the reasoning steps performed by the solver, the key point is that it is not tied to the solver that produced it: it is a standalone object that can be verified independently.

We cannot tell whether the proof was generated by a solver or written by hand, nor do we need to. The solver itself may be arbitrarily complex, and it may even contain bugs. What matters is the certificate. If the certificate is correct, then the claim for this specific instance is guaranteed to hold, regardless of any errors we might later discover and fix in the solver.

19 The CP Proof System

We now formalise the concepts introduced in the example. Our goal is to describe a proof system in which a solver can express the reasoning that leads to an infeasibility claim or an optimality bound.

Facts. The basic building block of a proof is a *fact*, which has the same syntactic form as an explanation. However, facts are part of the proof system, whereas explanations arise during search.

Facts are used to make the reasoning explicit. Although facts and explanations are similar, there is a core difference. Facts must be valid with respect to the entire set of constraints in the problem, as well as any facts that have already been derived. In contrast, explanations are valid with respect to the specific constraint that generated them, and, in addition, they must be applicable given the trail.

Every step in a proof derives one fact. This gives us a uniform object that can capture different forms of reasoning used by a solver.

Inferences. An inference consists of:

- the fact being justified;
- a single constraint from the original CSP or an earlier deduction that implies the fact;
- an *inference rule* that explains the reasoning step used to derive the fact from the given constraint.

Inference rules include the reasoning used by propagators, and the initial domains of variables, which effectively restate the information in the CSP. We require inference rules to be verifiable in polynomial time.

Inference rules serve as modular "hooks" that allow arbitrary propagation methods to be incorporated into the proof system. A solver may implement many different propagation algorithms, and each algorithm may correspond to a separate inference rule.

We do not assume completeness: an inference rule does not have to derive all possible consequences of its constraint. This mirrors solver behaviour, where propagators aim for efficiency rather than full logical strength.

Proof Stages. A *proof stage* derives a new fact by combining inferences. A proof stage consists of:

- a final implication $a_1 \wedge \cdots \wedge a_n \implies \bot$, called the *deduction*;
- a sequence of facts, called the *inferences*, that justify the deduction.

When a solver proves infeasibility, the deduction of the last stage is $\top \implies \bot$. For a minimisation optimisation problem where the optimal assignment has cost v, the value of the objective function is represented by an integer variable ℓ, and then proving a bound on the objective such as $\langle \ell \geq v \rangle$ can be written as the deduction $\langle \ell \leq v-1 \rangle \implies \bot$, effectively claiming that no better assignment exists.

A proof stage is valid if we can derive its deduction fact by following the sequence of inferences, as we did when justifying learned nogoods. We start with the atomic constraints in the deduction's premise and then apply the facts in the proof stage in order. If the premise of a fact is satisfied by the currently induced domains, we add its consequent. If not, we skip it. At the end, if the resulting set of atomics is conflicting, the proof stage is valid.

It is important to note that this implication is one-directional: if the evaluation does not produce a conflict, we can only conclude that the justification provided is insufficient, not that the deduction itself is false.

Proofs. A proof is a sequence of valid proof stages. Each stage can use the deductions of earlier stages as additional constraints. One of the deductions in the sequence is designated as the *conclusion*. This is the claim that the solver wishes to justify. For example, $\top \implies \bot$ for infeasibility, or $\top \implies \langle \ell \geq v \rangle$ for optimality.

Properties of the Proof System. The proof system is *sound*: any conclusion justified by a proof is logically implied by the original constraint problem. Intuitively, each inference rule guarantees that its fact is implied by the constraints; each deduction shows that its premise leads to a contradiction; and therefore the conclusion must also be implied by the CSP.

We can also show that this proof system is *complete* under the assumption of finite domains, although we will not go into the details.

Verification. Verifying that a proof is correct amounts to checking that it satisfies the definitions and conditions introduced above. Although producing a proof is a difficult problem, checking a proof is polynomial with respect to the size of the proof.

In earlier chapters, we discussed checkers, and much of that machinery carries over directly to proof verification. For this reason, we do not revisit those technical details here.

We now turn to the question of how to efficiently *generate* certificates during search.

20 Illustrative Example: Proof Production

Proofs typically, though not always, encapsulate a particular solver trace. The deduction steps correspond to the nogoods obtained through conflict analysis, and the inference steps correspond to explanations.

In other words, rather than viewing the solver execution as a search tree, we may also represent the search process as a proof. Since there are many ways to solve a problem, there may be many different proofs for the same problem.

In our previous example, the CSP was infeasible at the root level, and the reasoning steps recorded in the proof could be the sequence of operations performed by the solver to conclude infeasibility.

We will now look at another example. The CSP is defined as follows.

$$
\begin{aligned}
c_1 &: 2x+y+2z \geq 2,\\
c_2 &: 2x+y-2z \geq 0,\\
c_3 &: 2x-y+2z \geq 0,\\
c_4 &: 2x-y-2z \geq -2,\\
c_5 &: -2x+y+2z \geq 2,\\
c_6 &: -2x+y-2z \geq 0,\\
x,z &\in \{0,1\},\\
y &\in \{0,1,2\}.
\end{aligned}
$$

The problem is infeasible, and as before, we seek a certificate to support this claim. A possible proof is as follows.

ID	Derived constraint			
c_7	c_3	$\models$	$\langle x \leq 0\rangle \wedge \langle y \geq 2\rangle \implies \langle z \geq 1\rangle$	{Linear bounds}
c_8	c_4	$\models$	$\langle x \leq 0\rangle \wedge \langle y \geq 2\rangle \implies \langle z \leq 0\rangle$	{Linear bounds}
c_9	$c_7 \wedge c_8$	$\models$	$\langle x \leq 0\rangle \wedge \langle y \geq 2\rangle \implies \bot$	{Deduction}
c_{10}	c_1	$\models$	$\langle x \leq 0\rangle \wedge \langle y \leq 1\rangle \implies \langle z \geq 1\rangle$	{Linear bounds}
c_{11}	c_2	$\models$	$\langle x \leq 0\rangle \wedge \langle y \leq 1\rangle \implies \langle z \leq 0\rangle$	{Linear bounds}
c_{12}	$c_{10} \wedge c_{11}$	$\models$	$\langle x \leq 0\rangle \wedge \langle y \leq 1\rangle \implies \bot$	{Deduction}
c_{13}	c_9	$\models$	$\langle x \leq 0\rangle \implies \langle y \leq 1\rangle$	{Nogood}
c_{14}	c_{12}	$\models$	$\langle x \leq 0\rangle \implies \langle y \geq 2\rangle$	{Nogood}
c_{15}	$c_{13} \wedge c_{14}$	$\models$	$\langle x \leq 0\rangle \implies \bot$	{Deduction}
c_{16}	$\top$	$\models$	$\top \implies \langle y \geq 0\rangle$	{Problem domains}
c_{17}	c_{15}	$\models$	$\top \implies \langle x \geq 1\rangle$	{Nogood}
c_{18}	c_5	$\models$	$\langle x \geq 1\rangle \wedge \langle y \geq 0\rangle \implies \langle z \geq 1\rangle$	{Linear bounds}
c_{19}	c_6	$\models$	$\langle x \geq 1\rangle \wedge \langle y \geq 0\rangle \implies \langle z \leq 0\rangle$	{Linear bounds}
c_{20}	$c_{16} \wedge c_{17} \wedge c_{18} \wedge c_{19}$	$\models$	$\top \implies \bot$	{Deduction}

We can interpret the proof as arising from a possible solver execution. For instance, the solver might have first branched on $\langle x \leq 0 \rangle$ and then on $\langle y \geq 2 \rangle$, after which conflict analysis resulted in the learned nogood given by fact c_9. The solver then backtracked, posted the learned nogood, and propagated $\langle y \leq 1 \rangle$, which immediately leads to another conflict, corresponding to fact c_{12}. The solver then learned that the unit nogood $\langle x \leq 0 \rangle \implies \bot$ (fact c_{13}), after which it concluded infeasibility.

This walkthrough illustrates a basic approach to proof production: record each learned nogood as a deduction step, and the explanations used to justify the nogood then correspond to the inferences. These steps can be written to a file and used as the certificate upon termination.

There are two practical challenges. The first is that proofs are only relevant when the solver either proves infeasibility or optimality. If fails to do either within the time limit, any time spent recording the proof is wasted and slows down the solver. The second challenge is that solvers may perform substantial redundant work before finding the proof. Recording each step can, in this case, be prohibitively expensive, since the proof may become very large—sometimes gigabytes, even for moderate instances.

To illustrate this, consider an alternative proof for the CSP above. It is identical to the previous proof, except that it contains one additional proof stage:

ID	Derived constraint			
r_1	c_2	$\vDash$	$\langle y \leq 0 \rangle \wedge \langle z \geq 1 \rangle \implies \langle x \geq 1 \rangle$	{Linear bounds}
r_2	c_6	$\vDash$	$\langle y \leq 0 \rangle \wedge \langle z \geq 1 \rangle \implies \langle x \leq 0 \rangle$	{Linear bounds}
r_3	$c_7 \wedge c_8$	$\vDash$	$\langle y \leq 0 \rangle \wedge \langle z \geq 1 \rangle \implies \bot$	{Deduction}

Since the original proof was already correct, adding this extra proof step does not invalidate it; it simply introduces redundancy. This step could realistically have been produced during solving, but it is unnecessary for establishing infeasibility. Any effort spent recording such redundant steps is therefore wasted, however during the solver's execution, it is not obvious which steps will ultimately matter for the final proof.

Ideally, we only record the steps we truly need for the proof. Since directly reducing redundant work is fundamentally difficult, we will take a *lazy* approach [12]: we will record only the learned nogoods during search (referred to as a *proof scaffold*), and, as a post-processing step, if the solver made an infeasibility or optimality claim, we will remove redundant nogoods and compute justifications for the remaining nogoods, expanding the scaffold into a full proof. This ensures that only the nogoods that genuinely contribute to the final proof are processed, and they incur overhead only if the proof is eventually needed.

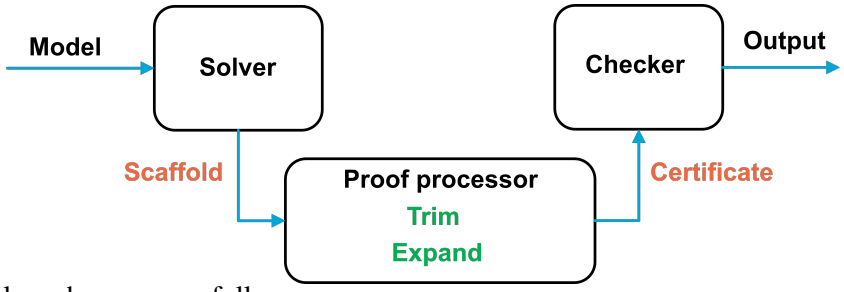
We now proceed with a more detailed view of our approach for producing certificates.

21 Multi-Stage Proof Production

Our goal is to construct certificates during search while keeping the runtime overhead as small as possible. Solvers may perform many redundant steps during search, and they may not even finish the proof within the allotted time. As a result, the effort spent on producing proofs can easily be wasted: a satisfaction problem may ultimately be feasible, most of the

recorded proof steps are redundant, or in an optimisation problem, the solver may never reach an optimality claim.

To address these issues, our proof production approach is split into several phases. A schematic overview is shown below.



The two key phases are as follows.

1. **Scaffolding**. The solver runs normally and produces a *proof scaffold*, a sequence of nogoods discovered during search (including registering nogood deletions).

   The requirement is that the i-th scaffold step must be justifiable in polynomial time: assuming the atomic constraints in the nogood should lead to a contradiction by propagation, considering the original constraints and the first $i-1$ steps of the scaffold. Learned nogoods naturally satisfy this property, referred to as *reverse unit propagation*.

2. **Processing** The scaffold is transformed into a full proof by a separate *proof elaborator* (proof processor). Conceptually, the elaborator performs two transformations:

   Trimming The scaffold often contains many nogoods that are redundant to prove infeasibility. Since inference introduction may add many steps, it is beneficial to remove redundant nogoods first.

   Inference Introduction For each remaining nogood C in the trimmed scaffold, the processor generates all inferences required to justify C. The output is a complete proof containing all reasoning steps needed to establish infeasibility.

   Two points are worth noting. First, the elaborator must be able to reason at least as strongly as the solver; otherwise it might be unable to justify the scaffold. Second, the processor is free to use reasoning that differs from the solver's runtime reasoning, as long as it reaches the same justified conclusion.

Although each phase could contain bugs, a sound proof checker will accept only correct proofs; any bug that produces unsound reasoning will be detected.

21.1 Reverse Unit Propagation and Nogood Justification

Learned nogoods satisfy a key property referred to as *reverse unit propagation*: given the constraints in the database at the time when the learned nogood has been derived, assuming the premises from the learned nogood leads to a conflict by propagation without any search.

Our proof production process relies on this property. It guarantees that we can compute a justification for the nogood in polynomial time.

The procedure is as follows. Given a nogood that requires justification, assume its atomic constraints from its premise, and propagate. If the nogood satisfies the reverse unit propagation property, this will result in a conflict. We can then use our conflict analysis procedure to compute a justification.

There are two points to highlight. First, the justification obtained during proof elaboration is not necessarily unique: the proof processor may compute a different justification than the one the solver used during search. This is not a problem, since we are only concerned with correctness, and any valid justification suffices. Second, conflict analysis may produce a justification for a nogood that is more general than the original one. In such cases, the more general justification can naturally be used to justify the original nogood as well.

Considering the antecedent graph, traversing this graph in topological order produces all necessary inference steps to justify the nogood.

The procedure above for justifying nogoods is a central operation in proof processing, and we will rely on it repeatedly in the steps that follow.

21.2 Trimming

Trimming removes redundant nogoods from the scaffold. The resulting scaffold can still be completed into a full proof by adding the appropriate inferences. Trimming itself does not determine those inferences; it merely establishes that it is possible, assuming the input scaffold was correct.

The underlying assumption is that the scaffold contains many redundant nogoods, and by removing them in this phase, we can avoid justifying nogoods that do not contribute to the final proof, saving memory and computational time.

The trimmed scaffold is not unique; different valid subsets may exist. One effective technique, adapted from SAT solving [13], is known as **backwards trimming**. The idea is as follows.

We begin by marking the nogood corresponding to the final infeasibility or optimality claim. Then, recursively, for each marked nogood that has not yet been considered, we mark all nogoods that were directly used to derive it. Identifying which nogoods participate in the derivation follows the same idea as justifying nogoods: if a nogood appears in the justification, then it clearly contributes to the derivation. We continue processing marked nogoods until every marked nogood has been processed.

In practice, rather than starting from the final infeasibility nogood, we mark the nogoods that participated in the last conflict (after inserting all scaffold nogoods into the constraint database and propagating).

Although described recursively, backwards trimming can be implemented as a single linear pass from the end of the scaffold toward the beginning. Whenever a nogood is marked, we identify which earlier nogoods were used to justify it and mark them as well. Nogoods that remain unmarked at the end of the pass are redundant and can be removed. Because propagation order can vary, different passes may produce different trimmed scaffolds.

One subtlety is ensuring that nogoods are not incorrectly used to justify earlier ones. To avoid this, nogoods are removed from the propagation engine once they have been processed

in the backwards direction. This enforces the invariant that the i-th nogood can only justify nogoods with those of a greater index, and not of a smaller index.

21.3 Inference Production

Inference production expands the trimmed scaffold into a complete proof by adding all inferences needed to justify each remaining nogood. The construction process is identical to justifying nogoods. Justifying each nogood in the scaffold leads to a complete proof.

21.4 Summary

Proofs serve as certificates that justify the claims made by solvers and allow us to trust the results of complex optimisation techniques. Once a proof is independently verified, ideally using a formally verified checker, we gain strong assurance that the solver's conclusion is correct. This level of confidence is especially important in high-stakes applications, and proofs also provide a useful way to uncover bugs during solver development.

In SAT solving, proof logging is now standard practice [14], and reliable checkers exist. For constraint programming, proofs are a more recent development. In this chapter, we introduced our recent CP-specific approach based on a multi-stage process: during solving we record a lightweight proof scaffold, and afterwards a proof processor trims and completes the proof.

This multi-stage approach limits overhead during search while still producing fully checkable proofs, making proof production practical for constraint programming.

22 Computational Limits

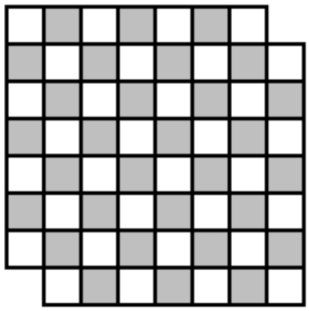
The algorithms we have developed so far form the basis of modern constraint programming solvers. On the one hand, these techniques are impressively effective: they can solve demanding real-world tasks such as timetabling, scheduling, and chip design. Constraint programming delivers strong performance across many practical applications.

On the other hand, these same algorithms may perform poorly on seemingly simple problems. Understanding these limitations helps us identify the fundamental strengths and weaknesses of our reasoning techniques, and ultimately guides us toward more powerful algorithms. We illustrate this with three classic infeasible problems.

The pigeon hole problem. Consider the question: can we place n pigeons into $n-1$ holes such that no two pigeons share a hole? The answer is obviously no. With $n-1$ holes and one pigeon per hole, we can place at most $n-1$ pigeons, so placing n pigeons is impossible.

If we model this using integer variables and an ALLDIFFERENT constraint, a domain-consistent propagator can prove infeasibility in polynomial time. But if we model it with binary variables (one variable for each pigeon–hole pair), the solver may need to explore an exponential number of assignments, essentially trying every way of placing n pigeons into $n-1$ holes before it finally fails. In this encoding, conflict analysis does not help, because the counting structure is not explicit in the model.

The mutilated chequerboard problem. Consider an $N \times N$ chequerboard with two diagonally opposite corners removed. Is it possible to tile the remaining squares with dominoes, each covering one black and one white square?



The answer is no: removing opposite corners removes two squares of the same colour, leaving an imbalance between black and white squares. Since each domino must cover exactly one of each colour, tiling is impossible. Yet depending on the modelling choice, solvers may again require exponential time to detect this infeasibility.

Contradictory inequalities. Finally, consider a problem with just two constraints:

$$\sum w_i x_i \geq k \qquad \wedge \qquad \sum w_i x_i \leq k-1.$$

The contradiction is immediate. However, our reasoning so far is not equipped to use this simple counting argument directly; the solver may again need to enumerate exponentially many assignments before it concludes infeasibility.

Across all three examples, the common theme is that a simple counting argument proves infeasibility immediately. However, our proof system reasons only over assignments, not over aggregates such as sums. Unless the model exposes such structure explicitly, the solver may be forced into exponential behaviour. While modelling can help, such structures may be hidden by other constraints, making the difficulty unavoidable.

22.1 Core Issues

Recall that our solvers may be viewed as *proof-producing algorithms*: whenever the solver concludes infeasibility, it could also output a proof of that claim. If our algorithms require exponential time on the problems above, then the corresponding proofs will also be exponentially long.

This raises a natural question: is the exponential proof length a consequence of the solver, or of the underlying *proof system*? To study such questions, we turn to *proof complexity*, the area of theoretical computer science that investigates the sizes of proofs within formal proof systems.

A proof system specifies the language of proofs, the basic starting objects (axioms), and the allowed derivation rules. Different systems have different strengths, and proofs that are short in one system may be long or even impossible in another.

To understand the limits of our constraint programming proof system, it is useful to examine an important special case: propositional satisfiability.

The Resolution Proof System

In the **SAT problem**, we are given Boolean variables x_i and clauses in conjunctive normal form (CNF). The task is to determine whether there exists an assignment of truth values that makes all clauses true.

The **Resolution proof system** takes the clauses of the formula as axioms and derives new clauses using a single inference rule. If one clause contains x and another contains $\overline{x}$, we may derive the *resolvent*:

$$\frac{(a_1 \vee \cdots \vee a_n \vee x) \qquad (b_1 \vee \cdots \vee b_m \vee \overline{x})}{(a_1 \vee \cdots \vee a_n \vee b_1 \vee \cdots \vee b_m)}.$$

This rule can simulate unit propagation, derive tautologies, and ultimately derive the empty clause, demonstrating unsatisfiability.

As an illustration, consider the four clauses:

$$(x \vee y), \quad (x \vee \overline{y}), \quad (\overline{x} \vee y), \quad (\overline{x} \vee \overline{y}).$$

Resolving the first two yields (x); resolving the last two yields $(\overline{x})$; resolving these results in the empty clause. Thus, the formula is infeasible.

A classic result shows that any resolution proof of unsatisfiability for the pigeonhole formula must be *exponentially* large. This means that no SAT solver whose reasoning is limited to resolution can avoid exponential behaviour on those instances, regardless of search heuristics or conflict analysis.

Back to Constraint Programming

Our constraint programming proof system generalises the classical resolution system. However, when the input is a propositional formula, conflict analysis together with the replacement of atomic constraints by their explanations amounts to applying the resolution rule, only written in a different notation. As a consequence, our proof system inherits the same fundamental limitations as resolution: there exist families of problems for which any proof must be exponentially long.

At the same time, our proof system supports high-level concepts that go beyond pure resolution, such as integer variables, atomic constraints, and specialised propagators. These features can dramatically reduce the size of proofs for some problems. Crucially, however, they do not help in all cases, and the inherent exponential lower bounds remain whenever the reasoning essentially collapses back to propositional resolution.

More expressive proof systems have been developed, including those based on linear inequalities, pseudo-Boolean reasoning, and *extended resolution*. These systems can yield dramatically shorter proofs for many problems. However, it remains an open question how to incorporate these more powerful techniques into practical, general-purpose constraint programming solvers, where constraints range over richer domains and operations are more varied.

In the next section, we discuss alternative proof systems to better understand how these proof systems could support solvers in the future.

23 Alternative Proof Systems

23.1 Extended Resolution

We now turn to ideas inspired by **extended resolution**, a technique from proof complexity that strengthens the classical resolution proof system by allowing the introduction of new auxiliary variables. These auxiliary variables are defined in terms of existing variables, so they do not change the feasibility of the problem. However, they can make important relationships explicit and therefore enable stronger reasoning.

The key idea is to introduce new variables that capture more complex combinations of existing variables. These new variables can then participate in conflict analysis, allowing the solver to derive facts (nogoods) that would not be expressible in the original variable space.

For instance, suppose our CSP has variables $x_1, \ldots, x_n$, and we introduce a new binary variable y with the meaning

$$y \leftrightarrow \sum_i w_i x_i \geq k.$$

This means that $y = 1$ implies the linear inequality $\sum_i w_i x_i \geq k$, while $y = 0$ implies $\sum_i w_i x_i \leq k - 1$. The new variable does not affect feasibility, but it allows us to express a compact nogood such as

$$\langle y = 1 \rangle \wedge \langle x_5 \geq 3 \rangle \implies \bot.$$

Without the auxiliary variable, expressing this same information would require a large number of separate nogoods—one for each assignment that violates the inequality. In this way, the auxiliary variable summarises an exponential set of assignments.

When we encode problems into SAT, we implicitly make use of extended resolution. SAT encodings introduce auxiliary variables to keep the encoding compact, and a beneficial side effect is that these variables allow the solver to learn stronger nogoods than would be possible without them. However, SAT encodings also introduce many variables and discard much of the high-level problem structure. Both of these factors can hinder search, so it is not always clear that encoding a problem into SAT is a good approach.

In constraint programming, we know that auxiliary variables can help, but we do not yet understand which variables should be introduced in general. Preliminary research shows promise [8], but developing general techniques is still an open question. Intuitively, we would like to introduce new variables that capture "important" sub-relationships among the original variables.

This brings us to a key difference between constraint programming and SAT solving. Constraint programming models are typically compact and can exploit high-level structure through specialised propagators. SAT solvers, in contrast, require a propositional encoding, which is often much larger, but this encoding introduces many auxiliary variables that can be leveraged during learning. In short, constraint programming benefits from rich modelling and strong propagation, whereas SAT benefits from the abundance of auxiliary variables that can result in stronger nogood learning.

A natural question is how to combine these advantages: can we introduce extended variables inside constraint programming models to enable stronger learning, without giving up the high-level benefits of CP? This remains an open research problem.

Since we know that our proof system has difficulty with counting arguments, we now explore ways to introduce auxiliary variables specifically for linear inequalities. We discuss two standard encodings from SAT but reinterpret them from a constraint programming perspective.

23.1.1 Sequential Encoding

Suppose we wish to introduce extended variables for a constraint of the form

$$\sum_i w_i x_i \leq k,$$

where the x_i are integer variables. The sequential encoding introduces intermediate variables representing partial sums:

$$y_j = \sum_{i \leq j} w_i x_i.$$

Rewritten differently as a recursive formulation, we obtain:

$$y_0 = w_0 x_0, \qquad y_j = y_{j-1} + w_j x_j.$$

The original constraint can then be rewritten as $y_n \leq k$.

These auxiliary variables may participate in conflict analysis. For example, consider the constraint $x_1 + x_2 + x_3 \leq 10$. If the solver has already established $x_1 \geq 3$ and $x_2 \geq 1$, it must propagate $x_3 \leq 6$. The explanation

$$[x_1 \geq 3] \wedge [x_2 \geq 1] \implies [x_3 \leq 6]$$

is valid but specific to the assignment. With the extended variable $y_2 = x_1 + x_2$, a more general explanation is possible:

$$[y_2 \geq 4] \implies [x_3 \leq 6].$$

This captures the reason more abstractly and generalises better.

23.1.2 Totaliser Encoding

The *totaliser encoding* [6, 15] introduces auxiliary variables using a tree-like structure of partial sums. For example, consider the constraint

$$x_1 + x_2 + \cdots + x_8 \leq 10.$$

The first layer of the totaliser constructs pairwise sums:

$$y_1 = x_1 + x_2, \qquad y_2 = x_3 + x_4, \qquad y_3 = x_5 + x_6, \qquad y_4 = x_7 + x_8.$$

The constraint becomes

$$y_1 + y_2 + y_3 + y_4 \leq 10.$$

The process continues recursively; for example:

$$z_1 = y_1 + y_2, \qquad z_2 = y_3 + y_4,$$

reducing the constraint to

$$z_1 + z_2 \leq 10.$$

Eventually, we introduce a final variable

$$m = z_1 + z_2,$$

and rewrite the constraint simply as $m \leq 10$.

Both the sequential and totaliser encodings rewrite the problem in different ways. Each has strengths, and which one performs better depends on the structure of the instance. For example, in the constraint

$$x_1 + x_2 + x_3 + x_4 \leq 10,$$

the totaliser may produce explanations such as

$$[x_3 + x_4 \geq 3] \Longrightarrow [x_1 + x_2 \leq 7],$$

while the sequential encoding may produce

$$[x_1 + x_2 + x_3 \geq 4] \Longrightarrow [x_4 \leq 6].$$

These explanations are not comparable; depending on the search, either may be useful.

Extended variables allow the solver to capture such relationships explicitly, and therefore to learn stronger nogoods. How best to introduce such variables in constraint programming remains an open problem.

23.2 Cutting Planes

Let us reconsider our problem of contradictory inequalities:

$$\sum w_i x_i \geq k \qquad \wedge \qquad \sum w_i x_i \leq k - 1.$$

It is easy to see that this system is infeasible. By rewriting the first inequality into an equivalent form using the $\leq$ operator and then adding the two inequalities will immediately result in a trivially infeasible constraint.

This motivates introducing a proof system in which the elementary constraints are not nogoods but linear inequalities. Such a system may be equipped with *Fourier resolution* (FR). Given two linear constraints

$$R_1 \equiv \sum_{i=1}^{n} w_i x_i \leq r \qquad \text{and} \qquad R_2 \equiv \sum_{i=1}^{n} w'_i x_i \leq r',$$

where for some index j we have $w_j > 0$ and $w'_j < 0$, Fourier resolution eliminates the variable x_j and derives the linear consequence

$$FR(R_1, R_2, x_j) \;\equiv\; \sum_{i=1, i\neq j}^{n} \left(-w'_j w_i + w_j w'_i\right) x_i \;\leq\; -w'_j r + w_j r'.$$

The above operation removes x_j from the system and results in a new valid linear inequality. This can then be used in an analogous way in conflict analysis as resolution.

A proof system based on Fourier resolution can produce proofs that are exponentially shorter than those in our CP proof system. However, FR is sound only over real-valued variables and does not incorporate reasoning specific to discrete domains. As a result, it is not complete for integer variables: some integer infeasibility arguments cannot be derived using Fourier resolution alone. This limits the applicability of the pure FR system in constraint programming.

To address this, we explored combining Fourier resolution with nogood reasoning [5]. The idea is that the solver first attempts to derive a new constraint using Fourier resolution; if this fails, it falls back to nogood resolution. Experiments showed that this hybrid approach can significantly reduce the number of conflicts, suggesting that combining linear reasoning with nogoods can be highly effective. A current research direction is to design practical algorithms that compute such hybrid resolutions efficiently in practice, so that they may become standard tools in constraint programming.

During this exploration, we found that it is possible to construct a new proof system that generalises both Fourier resolution and nogood resolution [11]. This system introduces *hypercube linear constraints* of the form

$$a_1 \wedge \cdots \wedge a_n \implies \sum_{i=1}^{n} w_i x_i \leq r.$$

Using these richer constraints, we can define a unified proof system that is both sound and complete, and in principle capable of yielding exponential improvements in proof length. The challenge ahead is understanding how to apply this system effectively in practice and incorporate it into real CP solvers.

In summary, alternative proof systems such as extended resolution and cutting-planes reasoning offer exciting theoretical possibilities. With further research, these systems may eventually become practical tools for constraint programming as well.

Bibliography

[1] Solvercheck: Declarative testing of constraints. In *Proceedings of the International Conference on the Principles and Practice of Constraint Programming (CP'19)*, Cham. ISBN 978-3-030-30048-7. doi: 10.1007/978-3-030-30048-7_33.

[2] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In *Proceedings of the International Conference on the Principles and Practice of Constraint Programming (CP'18)*.

[3] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *IJCAI*, volume 9, pages 399–404, 2009.

[4] Gilles Audemard and Laurent Simon. Refining restarts strategies for sat and unsat. In *Proceedings of the Conference on the Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012.

[5] Robbin Baauw, Maarten Flippo, and Emir Demirović. Conflict analysis based on cutting-planes for constraint programming. In *31st International Conference on Principles and Practice of Constraint Programming (CP 2025)*, 2025.

[6] Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In *Proceedings of the International conference on principles and practice of constraint programming*, 2003.

[7] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Proceedings of the International Conference on the Theory and Applications of Satisfiability Testing (SAT'10)*.

[8] Geoffrey Chu and Peter J Stuckey. Structure based extended resolution for constraint programming. *arXiv preprint arXiv:1306.4418*, 2013.

[9] William Cook, Thorsten Koch, Daniel E Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, 2013.

[10] Thibaut Feydy, Andreas Schutt, and Peter Stuckey. Semantic learning for lazy clause generation. In *TRICS workshop*, 2013.

[11] Maarten Flippo, Peter Stuckey, and Emir Demirović. Resolution meets cutting planes: Introducing hypercube linear resolution. In *(to appear in the) Proceedings of the International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 2026.

[12] ML Flippo, Konstantin Sidorov, ICWM Marijnissen, Jeff Smits, and Emir Demirović. A multi-stage proof logging framework to certify the correctness of cp solvers. 2024.

[13] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Proceedings of Formal Methods in Computer-Aided Design (FM-CAD'13)*. IEEE, 2013.

[14] Marijn JH Heule. Proofs of unsatisfiability. In *Handbook of Satisfiability*, pages 635–668. IOS Press, 2021.

[15] Saurabh Joshi, Ruben Martins, and Vasco Manquinho. Generalized totalizer encoding for pseudo-boolean constraints. In *Proceedings of the International conference on principles and practice of constraint programming*, 2015.

[16] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 133–182. ios Press, 2021.

[17] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the Design Automation Conference*, pages 530–535, 2001.

[18] Chanseok Oh. *Improving SAT solvers by exploiting empirical characteristics of CDCL*. PhD Thesis at New York University, 2016.

[19] Tobias Paxian and Armin Biere. Uncovering and classifying bugs in maxsat solvers through fuzzing and delta debugging. 2023.

[20] Allen Van Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In *Proceedings of the Conference on Theory and Applications of Satisfiability Testing*, pages 141–146, 2009.