Chapter 7

Global Constraints

Willem-Jan van Hoeve and Irit Katriel

Contents

A *global constraint* is a constraint that captures a relation between a non-fixed number of variables. An example is the constraint `alldifferent`$(x_1, \ldots, x_n)$, which specifies that the values assigned to the variables $x_1, \ldots, x_n$ must be pairwise distinct. Typically, a global constraint is semantically redundant in the sense that the same relation can be expressed as the conjunction of several simpler constraints. Having shorthands for frequently recurring patterns clearly simplifies the programming task. What may be less obvious is that global constraints also facilitate the work of the constraint solver by providing it with a better view of the structure of the problem.

One of the central ideas of constraint programming is the propagation-search technique, which consists of a traversal of the search space of the given constraint satisfaction problem (CSP) while detecting "dead ends" as early as possible. An algorithm that performs only the search component would enumerate all possible assignments of values to the variables until it either finds a solution to the CSP or exhausts all possible assignments and concludes that a solution does not exist. Such an exhaustive search has an exponential-time complexity in the *best case,* and this is where propagation comes in: It allows the constraint solver to prune useless parts of the search space without enumerating them. For example, if the CSP contains the constraint $x + y = 3$ and both x and y are set to 1, we can conclude that regardless of the values assigned to other variables, the partial assignment we have constructed so far cannot lead to a solution. Thus it is safe to backtrack and reverse some of our previous decisions (see also Chapter 3, "Constraint Propagation", and Chapter 4 "Backtracking Search Algorithms for CSPs").

The type of propagation that we will discuss in this chapter is called *filtering* of the variable domains. The filtering task is to examine the variables which were not assigned values yet, and remove useless values from their domains. A value is useless if it cannot participate in any solution that conforms with the assignments already made. Since it is, in general, NP-hard to determine whether or not a value in the domain of a variable is useful for the CSP, the solver filters separately with respect to each of the constraints. If a value is useless with respect to one of the constraints, then it is also useless with respect to the whole CSP, but not vice versa. In other words, filtering separately with respect to each constraint allows false-positives (keeping a value which is useless for the CSP), but not false-negatives (removing a useful value). We then arrive at a tradeoff between the *efficiency* of the filtering (i.e., the running time) and its *effectiveness* (i.e., how many useless values were identified). "Good" constraints are constraints that address this tradeoff by allowing significant filtering with a low computational cost.

A *filtering algorithm* for a constraint C is an algorithm that filters the domains of variables with respect to C. If the algorithm removes every useless value from the domain of every variable that C is defined on, we say that it achieves *complete filtering*. If it removes only some of the useless values, we say that it performs *partial filtering*.

This chapter explores the topic of globals constraints. Our goal is to familiarize the reader with the important concepts of the field, which include different types of constraints, different measures of filtering and different compromises between efficiency and effectiveness of filtering. We will illustrate each of the concepts with some examples, that is, specific global constraints and filtering algorithms. We believe that our (obviously non-exhaustive) selection of constraints and algorithms suffices to provide the reader with an overview of the state of the art of research on global constraints.

The rest of the chapter is organized as follows. Section 7.1 provides notation and preliminaries for the rest of the chapter. In Section 7.2 we describe some useful global constraints. In Section 7.3 we describe efficient algorithms that achieve complete filtering for several global constraints. In Section 7.4 we describe global optimization constraints, i.e., constraints that encapsulate optimization criteria, and filtering algorithms for them. Section 7.5 covers the topic of partial filtering algorithms, beginning with their motivation through definitions of different measures of filtering to actual examples of partial filtering algorithms. In Section 7.6 we describe complex variable types, constraints defined on them and filtering algorithms for such constraints. Finally, in Section 7.7 we review some recent ideas and directions for further research.

7.1 Notation and Preliminaries

7.1.1 Constraint Programming

The *domain* of a variable x, denoted $D(x)$, is a finite set of elements that can be assigned to x. For a set of variables X we denote the union of their domains by $D(X) = \cup_{x \in X} D(x)$.

Let $X = \{x_1, \ldots, x_k\}$ be a set of variables. A *constraint* C on X is a subset of the Cartesian product of the domains of the variables in X, i.e., $C \subseteq D(x_1) \times \ldots \times D(x_k)$. A tuple $(d_1, \ldots, d_k) \in C$ is called a *solution* to C. Equivalently, we say that a solution $(d_1, \ldots, d_k) \in C$ is an assignment of the value d_i to the variable x_i, for all $1 \leq i \leq k$, and that this assignment *satisfies* C. If $C = \emptyset$, we say that it is *inconsistent*. When a constraint C is defined on a set X of k variables together with a certain set p of ℓ parameters, we will denote it by $C(X, p)$, but consider it to be a set of k-tuples (and not $k + \ell$-tuples).

A *constraint satisfaction problem* (*CSP*) is a finite set of variables X, together with a finite set of constraints C, each on a subset of X. A *solution to a CSP* is an assignment of a value $d \in D(x)$ to each $x \in X$, such that all constraints are satisfied simultaneously.

Given a constraint C defined on the variables $\{x_1, \ldots, x_k\}$, the filtering task is to shrink the domain of each variable such that it still contains all values that this variable can assume in a solution to C. An algorithm that achieves complete filtering, computes, for every $1 \leq j \leq k$,

$$D(x_j) \leftarrow D(x_j) \cap \{v_i \mid D(x_1) \times \ldots \times D(x_{j-1}) \times \{v_i\} \times D(x_{j+1}) \times \ldots \times D(x_k) \cap C \neq \emptyset\}.$$

In many applications, we wish to find a solution to a CSP that is optimal with respect to certain criteria. A *constraint optimization problem* (*COP*) is a CSP P defined on the variables $x_1, \ldots, x_n$, together with an *objective function* $f : D(x_1) \times \ldots \times D(x_n) \to \mathbb{Q}$ that assigns a value to each assignment of values to the variables. An *optimal solution* to a minimization (maximization) COP is a solution d to P that minimizes (maximizes) the value of $f(d)$. The objective function value is often represented by a variable z, together with the "constraint" `maximize` z or `minimize` z for a maximization or a minimization problem, respectively.

7.1.2 Graph Theory

Basic Notions

A *graph* or *undirected graph* is a pair $G = (V, E)$, where V is a finite set of vertices and $E \subseteq V \times V$ is a multiset[1] of *unordered* pairs of vertices, called *edges*. An edge "between"

[1]A multiset is a set in which an element may occur more than once.

$u \in V$ and $v \in V$ is denoted by $\{u, v\}$. A graph G is *bipartite* if there exists a partition $S \dot\cup T$ of V such that $E \subseteq S \times T$. We then write $G = (S, T, E)$.

A *walk* in a graph $G = (V, E)$ is a sequence $P = v_0, e_1, v_1, \ldots, e_k, v_k$ where $k \geq 0$, $v_0, \ldots, v_k \in V$, $e_1, \ldots, e_k \in E$ and $e_i = \{v_{i-1}, v_i\}$ for $1 \leq i \leq k$. If there is no confusion, P may be denoted by $v_0, v_1, \ldots, v_k$ or $e_1, e_2, \ldots, e_k$. A walk is called a *path* if $v_0, \ldots, v_k$ are distinct. A closed path, i.e., $v_0 = v_k$, is called a *circuit*.

An *induced subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' = \{\{u, v\} \mid u \in V', v \in V', \{u, v\} \in E\}$. A *component* or *connected component* of a graph $G = (V, E)$ is an induced subgraph $G' = (V', E')$ of G such that there exists a u-v path in G' for every pair $u, v \in V'$, and G' is maximal with respect to V'.

A *digraph* or *directed graph* is a pair $G = (V, A)$ where V is a finite set of vertices and $A \subseteq V \times V$ is a multiset of *ordered* pairs of vertices, called *arcs*. A pair occurring more than once in A is called a multiple arc. An arc from $u \in V$ to $v \in V$ is denoted by (u, v). The set of arcs incoming into a vertex u is denoted by $\delta^{\text{in}}(u) = A \cap (V \times \{u\})$ and the set of arcs outgoing from a vertex u is denoted by $\delta^{\text{out}}(u) = A \cap (\{u\} \times V)$. Similarly to undirected bipartite graphs, a directed graph $G = (V, A)$ is *bipartite* if there exists a partition $S \dot\cup T$ of V such that $A \subseteq (S \times T) \cup (T \times S)$. We then write $G = (S, T, A)$.

A *directed walk* in a directed graph $G = (V, A)$ is a sequence $P = v_0, a_1, v_1, \ldots, a_k, v_k$ where $k \geq 0$, $v_0, \ldots, v_k \in V$, $a_1, \ldots, a_k \in A$ and $a_i = (v_{i-1}, v_i)$ for $1 \leq i \leq k$. Again, if there is no confusion, P may be denoted by $v_0, v_1, \ldots, v_k$ or $a_1, a_2, \ldots, a_k$. A directed walk is called a *directed path* if $v_0, \ldots, v_k$ are distinct. A closed directed path, i.e., $v_0 = v_k$, is called a *directed circuit*.

An induced subgraph of a digraph $G = (V, A)$ is a graph $G' = (V', A')$ such that $V' \subseteq V$ and $A' = A \cap (V' \times V')$. A *strongly connected component* of a digraph $G = (V, A)$ is an induced subgraph $G' = (V', A')$ of G such that there exists a directed u-v path in G' for every pair $u, v \in V'$, and G' is maximal with respect to V'.

Matching Theory

Given an undirected graph $G = (V, E)$, a *matching* in G is a set $M \subseteq E$ of disjoint edges, i.e., no two edges in M share a vertex. A matching is said to *cover* a vertex v if v belongs to some edge in M. For a set $S \subseteq V$, we say that M covers S if it covers every vertex in S. A vertex $v \in V$ is called M*-free* if M does not cover v. The *cardinality* of a matching M is the number of edges in it, $|M|$. The *maximum cardinality matching problem* is the problem of finding a matching of maximum cardinality in a graph.

Let M be a matching in a graph $G = (V, E)$. A path P in G is called M*-augmenting* if P has odd length, its ends are not covered by M, and its edges are alternatingly out of and in M. A circuit C in G is called M*-alternating* if its edges are alternatingly out of and in M. Given an M-augmenting path P, the symmetric difference[2] of M and P gives a matching M' with $|M'| = |M| + 1$. Furthermore, the existence of an M-alternating path is a *necessary* condition for the existence of a matching of larger cardinality:

Theorem 1 (Petersen [50]) *Let $G = (V, E)$ be a graph, and let M be a matching in G. Then M is a maximum-cardinality matching if and only if there does not exist an M-augmenting path in G.*

[2]For two sets A and B, the *symmetric difference* $A \oplus B$ is the set of elements that belong to A or B but not both. Formally, $A \oplus B = (A \cup B) \setminus (A \cap B)$.

Hence, a maximum-cardinality matching can be found by repeatedly finding an M-augmenting path in G and using it to extend M. On a bipartite graph $G = (U, W, E)$, this can be done with the following method, due to van der Waerden [67] and König [38]. Let M be the current matching. Construct the directed bipartite graph $G_M = (U, W, A)$ by orienting all edges in M from W to U and all other edges from U to W, i.e.,

$$\begin{aligned} A = \quad & \{(w, u) \mid \{u, w\} \in M, u \in U, w \in W\} \;\cup \\ & \{(u, w) \mid \{u, w\} \in E \setminus M, u \in U, w \in W\}. \end{aligned}$$

Then every directed path in G_M starting from an M-free vertex in U and ending in an M-free vertex in W corresponds to an M-augmenting path in G. By choosing $|U| \leq |W|$, we need to find at most $|U|$ such paths. As each path can be identified in at most $O(|A|)$ time by breadth-first search, the time complexity of this algorithm is $O(|U|\,|A|)$.

Hopcroft and Karp [28] improved this running time to $O(|U|^{1/2}\,|A|)$, where we choose again $|U| \leq |W|$. Instead of repeatedly augmenting M along a single M-augmenting path, the idea is to repeatedly augment M simultaneously along a collection of disjoint M-augmenting paths. Such a collection of paths can again be found in $O(|A|)$ time. By reasoning on the lengths of the alternating paths, one can show that the algorithm needs only $O(|U|^{1/2})$ iterations, leading to a total time complexity of $O(|U|^{1/2}\,|A|)$.

Flow Theory

Let $G = (V, A)$ be a directed graph and let $s, t \in V$. A function $f : A \to \mathbb{R}$ is called a *flow from s to t*, or an *s-t flow*, if

$$\begin{array}{lll} (i) & f(a) \geq 0 & \text{for each } a \in A, \\ (ii) & f(\delta^{\text{out}}(v)) = f(\delta^{\text{in}}(v)) & \text{for each } v \in V \setminus \{s, t\}. \end{array} \tag{7.1}$$

where for any set S of arcs, $f(S) = \sum_{a \in S} f(a)$. Property (7.1)$(ii)$ ensures *flow conservation*, i.e., for a vertex $v \neq s, t$, the amount of flow entering v is equal to the amount of flow leaving v.
The *value* of an s-t flow f is defined to be

$$\text{value}(f) = f(\delta^{\text{out}}(s)) - f(\delta^{\text{in}}(s)).$$

In other words, the value of a flow is the net amount of flow leaving s, which by flow conservation must be equal to the net amount of flow entering t.

In a flow network, each arc a is associated with a *requirement* $[d(a), c(a)]$ where $c(a) \geq d(a) \geq 0$. Viewing $d(a)$ as the "demand" of a and $c(a)$ as its "capacity", we say that a flow f is *feasible* in the network if $d(a) \leq f(a) \leq c(a)$ for every $a \in A$.

Let $w : A \to \mathbb{R}$ be a "weight" (or "cost") function for the arcs. For a directed path P in G we define $w(P) = \sum_{a \in P} w(a)$. Similarly for a directed circuit. The *weight* of any flow $f : A \to \mathbb{R}$ is defined to be

$$\text{weight}(f) = \sum_{a \in A} w(a) f(a).$$

A feasible flow f is called a *minimum-weight flow* if $\text{weight}(f) \leq \text{weight}(f')$ for any feasible flow f'. Given a digraph $G = (V, A)$ with $s, t \in V$, the *minimum-weight flow problem* is to find a minimum-weight s-t flow in G.

Algorithm 1: Minimum-weight feasible s-t flow in $G = (V, A)$

```
set $f = \vec{0}$
add the arc $(t, s)$ with $d(t, s) = 0, c(t, s) = \infty, w(t, s) = 0$ and $f(t, s) = 0$ to $G$
**while** *there exists an arc* $(u, v)$ *with* $f(u, v) < d(u, v)$ **do**
    compute a directed $v$-$u$ path $P$ in $G_f$ minimizing $w(P)$
    **if** $P$ *does not exist* **then** stop (no feasible flow exists)
    **else** define the directed circuit $C = P, u, v$
    reset $f = f + \varepsilon\chi^C$, where $\varepsilon$ is maximal subject to $\vec{0} \leq f + \varepsilon\chi^P \leq \vec{c}$ and
    $f(u, v) + \varepsilon \leq d(u, v)$
```

Let f be an s-t flow in G. The *residual graph* of G with respect to f is defined as $G_f = (V, A_f)$ where for each $(u, v) \in A$, if $f(u, v) < c(u, v)$ then $(u, v) \in A_f$ with residual demand $\max\{d(u, v) - f(u, v), 0\}$ and residual capacity $c(u, v) - f(u, v)$, and if $f(u, v) > d(u, v)$ then $(v, u) \in A_f$ with residual demand 0 and residual capacity $f(v, u) - d(v, u)$. Intuitively, if the capacity of an arc is not exceeded, then the residual demand indicates how much more flow *must* be sent along this arc for its demand to be fulfilled and the residual capacity indicates how much additional flow *can* be sent along this arc without exceeding its capacity. If the flow on an arc is strictly higher than its demand, then the residual capacity (on an arc which is oriented in the reverse direction) indicates by how much we may reduce the flow on this arc, while still fulfilling its demand.

Let P be a directed path in G_f. Every arc $a \in P$ appears in G either in the same orientation (as the arc a) or in reverse direction (as the arc a^{-1}). The characteristic vector of P is defined as follows:

$$\chi^P(a) = \begin{cases} 1 & \text{if } P \text{ traverses } a, \\ -1 & \text{if } P \text{ traverses } a^{-1}, \\ 0 & \text{if } P \text{ traverses neither } a \text{ nor } a^{-1}, \end{cases}$$

For a directed circuit C in G_f, we define $\chi^C \in \{-1, 0, 1\}^A$ similarly.

Using the above notation, a feasible s-t flow in G with minimum weight can be found using Algorithm 1, which is sometimes referred to as the *successive shortest paths algorithm*, due to Ford and Fulkerson [18], Jewell [31], Busacker and Gowen [11], and Iri [30]. It begins by adding the arc (t, s) to G, with demand 0 and infinite capacity. This simplifies the computations because we no longer need to consider s and t as special vertices; all we need in order to have a feasible flow is to ensure that flow conservation holds at every vertex. Then, the algorithm repeatedly finds an arc whose demand is not respected and adds flow along a cycle in the residual graph that contains this arc. The flow is increased maximally along this cycle, taking into account the demand and capacity requirements of the arcs on the cycle. Note that in order to meet the demand of an arc, it may be necessary to increase the flow along more than one directed cycle. It can be proved that for integer demand and capacity functions and non-negative weights, Algorithm 1 finds an integral feasible s-t flow with minimum weight if it exists; see for example [62, p. 175–176].

The time complexity of Algorithm 1 is $O(\phi \cdot \mathrm{SP})$, where ϕ is the value of the flow found and SP is the time to compute a shortest directed path in G. Although faster algorithms exist for general minimum-weight flow problems, this algorithm suffices for our purposes, because we only need to find flows of relatively small values.

Note that the van der Waerden-König algorithm for finding a maximum-cardinality matching in a bipartite graph is a special case of the above algorithm. Namely, let $G = (U, W, E)$ be a bipartite graph. Similar to the construction of the directed bipartite graph G_M in Section 7.1.2, we transform G into a directed bipartite graph G' by orienting all edges from U to W. Furthermore, we add a “source” s, a “sink” t, and arcs from s to all vertices U and from all vertices in W to t. To all arcs a of the resulting graph we assign a capacity $c(a) = 1$ and a weight $w(a) = 0$. Now the algorithm for finding a minimum-weight s-t flow in G' mimics exactly the augmenting paths algorithm for finding a maximum-cardinality matching in G. In particular, given a flow f in G' and the corresponding matching M in G, the directed graph G_M corresponds to the residual graph G'_f where s, t and their adjacent arcs have been removed. Similarly, an M-augmenting path in G_M corresponds to a directed s-t path in G'_f.

Finally, we mention a result that, as we will see, is particularly useful for designing incremental filtering algorithms. Given a minimum-weight s-t flow, we want to compute the increase that would occur in the weight of solution when an unused arc is forced to be used. The following result shows that this can be done by re-routing the flow through a minimum-cost circuit containing the unused arc, see [2, p. 338].

Theorem 2 *Let f be a minimum-weight s-t flow of value ϕ in $G = (V, A)$ with $f(a) = 0$ for some $a \in A$. Let C be a directed circuit in G_f with $a \in C$, minimizing $w(C)$. Then $f' = f + \varepsilon\chi^C$, where ε is subject to $d \leq f + \varepsilon\chi^C \leq c$, has minimum weight among all s-t flows g in G with value$(g) = \phi$ and $g(a) = \varepsilon$. If C does not exist, f' does not exist. Otherwise, weight$(f') =$ weight$(f) + \varepsilon \cdot w(C)$.*

The proof of Theorem 2 relies on the fact that for a minimum-weight flow f in G, the residual graph G_f does not contain directed circuits with negative weight.
For further reading on network flows we recommend Ahuja et al. [2] or Schrijver [62, Chapter 6–15].

7.1.3 Linear Programming

A *linear program* consists of continuous variables and linear constraints (inequalities or equalities). The objective is to optimize a linear cost function. One of the standard forms of a linear program is

$$\begin{array}{rccccccccc}
\min & c_1x_1 & + & c_2x_2 & + & \dots & + & c_nx_n & & \\
\text{subject to} & a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\
& a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\
& \vdots & & & & & & & & \vdots \\
& a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & = & b_m \\
& x_1, \dots, x_n \geq 0 & & & & & & & &
\end{array}$$

or, using matrix notation,

$$\min \{c^\mathsf{T}x \mid Ax = b, x \geq 0\} \tag{7.2}$$

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$ and $x \in \mathbb{R}^n$. Here c represents the "cost" vector and x is the vector of variables. Every linear program can be transformed into a linear program in the form of (7.2); see for example [61, Section 7.4].

Recall that the *rank* of a matrix is the number of linearly independent rows or columns of the matrix. For simplicity, we assume in the following that the rank of A is m, i.e. there are no redundant equations in (7.2).

Let $A = (a_1, a_2, \dots, a_n)$ where a_j is the j-th column of A. For some "index set" $I \subseteq \{1, \dots, n\}$ we denote by A_I the submatrix of A consisting of the columns a_i with $i \in I$.

Because the rank of A is m, there exists an index set $B = \{B_1, \dots, B_m\}$ such that the $m \times m$ submatrix $A_B = (a_{B_1}, \dots, a_{B_m})$ is nonsingular and is therefore invertible. We call A_B a *basis* of A. Let $N = \{1, \dots, n\} \setminus B$. If we permute the columns of A such that $A = (A_B, A_N)$, we can write $Ax = b$ as

$$A_B x_B + A_N x_N = b,$$

where $x = (x_B, x_N)$. Then a solution to $Ax = b$ is given by $x_B = A_B^{-1}b$ and $x_N = \vec{0}$. This solution is called a *basic solution*. A basic solution is *feasible* if $A_B^{-1}b \geq \vec{0}$. The vector x_B contains the *basic variables* and the vector x_N contains the *nonbasic variables*. If we permute c such that $c = (c_B, c_N)$, the corresponding objective value is $c^\mathsf{T} x = c_B^\mathsf{T} A_B^{-1} b + c_N^\mathsf{T} \vec{0} = c_B^\mathsf{T} A_B^{-1} b$.

Given a basis A_B, we can rewrite (7.2) into the following equivalent linear program

$$\begin{array}{rl} \min & c_B^\mathsf{T} A_B^{-1} b + (c_N^\mathsf{T} - c_B^\mathsf{T} A_B^{-1} A_N) x_N \\ \text{subject to} & x_B + A_B^{-1} A_N x_N = A_B^{-1} b \\ & x_B, x_N \geq 0. \end{array} \tag{7.3}$$

Program (7.3) represents how the objective may improve if we would replace (some) basic variables by nonbasic variables. This means that some basic variables will take value 0, while some nonbasic variables will take a non-zero value instead. If we do so, feasibility is maintained by $x_B + A_B^{-1} A_N x_N = A_B^{-1} b$. The improvement of the objective value is represented by $(c_N^\mathsf{T} - c_B^\mathsf{T} A_B^{-1} A_N) x_N$. This rewritten cost vector for x_N is called the *reduced-cost* vector and is defined on both basic and nonbasic variables as $\overline{c}^\mathsf{T} = c^\mathsf{T} - c_B^\mathsf{T} A_B^{-1} A$. We have the following (cf. [46, pp. 31–32]):

Theorem 3 *(x_B, x_N) is an optimal solution if and only if $\overline{c} \geq \vec{0}$.*

Apart from this result, reduced-costs have another interesting property. Namely, they represent the marginal rate at which the solution gets worse if we insert a nonbasic variable into the solution (by giving it a non-zero value). For example, if we insert nonbasic variable x_i into the solution, the objective value will increase by at least $\overline{c}_i x_i$. This property will be exploited in Section 7.5.2.

To solve linear programs one often uses the *simplex method*, invented by Dantzig [15], which employs Theorem 3. Roughly, the simplex method moves from one basis to another by replacing a column in A_B by a column in A_N, until it finds a basic feasible solution for which all reduced-costs are nonnegative. The method is very fast in practice, although it has an exponential worst-case time complexity. Polynomial-time algorithms for linear programs were presented by Khachiyan [36] and Karmarkar [32, 33].
For further reading on linear programming we recommend Chvátal [14] or Nemhauser and Wolsey [46].

7.2 Examples of Global Constraints

In this section we present a number of global constraints that are practically useful and for which efficient filtering algorithms exist.

7.2.1 The Sum and Knapsack Constraints

The `sum` constraint is one of the most frequently occurring constraints in applications. Let $x_1, \dots, x_n$ be variables. To each variable x_i, we associate a scalar $c_i \in \mathbb{Q}$. Furthermore, let z be a variable with domain $D(z) \subseteq \mathbb{Q}$. The `sum` constraint is defined as

$$\texttt{sum}(x_1, \dots, x_n, z, c) = \{(d_1, \dots, d_n, d) \mid \forall i \; d_i \in D(x_i), d \in D(z), d = \textstyle\sum_{i=1}^{n} c_i d_i\}\,.$$

We also write $z = \sum_{i=1}^{n} c_i x_i$.

The `knapsack` constraint is a variant of the `sum` constraint. Rather than constraining the sum to be a specific value, the `knapsack` constraint states the sum to be within a lower bound l and an upper bound u. Traditionally, one writes $l \leq \sum_{i=1}^{n} c_i x_i \leq u$. Here we represent l and u by a variable z, such that $D(z) = [l, u]$. Then we define the `knapsack` constraint as

$$\begin{aligned}&\texttt{knapsack}(x_1, \dots, x_n, z, c) = \\ &\quad \{(d_1, \dots, d_n, d) \mid \forall i \; d_i \in D(x_i), d \in D(z), d \leq \textstyle\sum_{i=1}^{n} c_i d_i\} \;\cap \\ &\quad \{(d_1, \dots, d_n, d) \mid \forall i \; d_i \in D(x_i), d \in D(z), \textstyle\sum_{i=1}^{n} c_i d_i \leq d\}\,,\end{aligned}$$

which corresponds to $\min D(z) \leq \sum_{i=1}^{n} c_i x_i \leq \max D(z)$.

7.2.2 The Element Constraint

Let y be an integer variable, z a variable with finite domain, and c an array of variables, i.e., $c = [x_1, x_2, \dots, x_n]$. The `element` constraint states that z is equal to the y-th variable in c, or $z = x_y$. More formally

$$\begin{aligned}&\texttt{element}(y, z, x_1, \dots, x_n) = \\ &\quad \{(e, f, d_1, \dots, d_n) \mid e \in D(y), f \in D(z), \forall i \; d_i \in D(x_i), f = d_e\}.\end{aligned}$$

The `element` constraint was introduced Van Hentenryck and Carillon [24]. It can be applied to model many practical problems, especially when we want to model variable subscripts. An example is presented in Section 7.2.8 below.

7.2.3 The Alldifferent Constraint

The `alldifferent` constraint is probably the best-known, most influential and most studied global constraint in constraint programming. Apart from its simplicity and practical applicability, this is probably due to its relationship to matching theory. This important field of theoretical computer science has produced several classical results and provided the basis for efficient filtering algorithms for the `alldifferent` constraint.

Definition 1 (Alldifferent constraint, [39]) *Let $x_1, x_2, \dots, x_n$ be variables. Then*

$$\texttt{alldifferent}(x_1, \dots, x_n) = \{(d_1, \dots, d_n) \mid \forall_i \; d_i \in D(x_i), \; \forall_{i \neq j} \; d_i \neq d_j\}.$$

A famous problem that can be modeled with `alldifferent` constraints is the n-queens problem: Place n queens on an $n \times n$ chessboard in such a way that no queen attacks another queen.

One way of modeling this problem is to introduce an integer variable x_i for every row $i = 1, 2, \dots, n$, which ranges over column 1 to n. This means that in row i, a queen is placed in the x_i-th column. The domain of every x_i is $D(x_i) = \{1, 2, \dots, n\}$ and we express the no-attack constraints by

$$x_i \neq x_j \quad \text{for } 1 \leq i < j \leq n, \tag{7.4}$$
$$x_i - x_j \neq i - j \quad \text{for } 1 \leq i < j \leq n, \tag{7.5}$$
$$x_i - x_j \neq j - i \quad \text{for } 1 \leq i < j \leq n, \tag{7.6}$$

The constraints (7.4) state that no two queens are allowed to occur in the same column and the constraints (7.5) and (7.6) state the diagonal cases. A more concise model can be stated as follows. After rearranging the terms of constraints (7.5) and (7.6), we transform the model into

$$\begin{array}{l}\texttt{alldifferent}(x_1, \dots, x_n),\\ \texttt{alldifferent}(x_1 - 1, x_2 - 2, \dots, x_n - n),\\ \texttt{alldifferent}(x_1 + 1, x_2 + 2, \dots, x_n + n),\\ x_i \in \{1, 2, \dots, n\} \text{ for } 1 \leq i \leq n.\end{array}$$

7.2.4 The Global Cardinality Constraint

The *global cardinality constraint* $\texttt{gcc}(x_1, \dots, x_n, c_{v_1}, \dots, c_{v_{n'}})$ is a generalization of `alldifferent`. While `alldifferent` requires that every value is assigned to at most one variable, the `gcc` is specified on n assignment variables $x_1, \dots, x_n$ and n' count variables $c_{v_1}, \dots, c_{v_{n'}}$ and specifies that each value v_i is assigned to exactly c_{v_i} assignment variables. `alldifferent`, then, is the special case of `gcc` in which the domain of each count variable is $\{0, 1\}$. For any tuple $t \in D^n$ and value $v \in D$, let $occ(v, t)$ be the number of occurrences of v in t.

Definition 2 (Global cardinality constraint, [47]) *Let $x_1, \dots, x_n$ be assignment variables whose domains are contained in $\{v_1, \dots v_{n'}\}$ and let $\{c_{v_1}, \dots, c_{v_{n'}}\}$ be count variables whose domains are sets of integers. Then*

$$\begin{array}{l}\texttt{gcc}(x_1, \dots, x_n, c_{v_1}, \dots, c_{v_{n'}}) = \{(w_1, \dots, w_n, o_1, \dots, o_{n'}) \mid \\ \quad \forall j \ w_j \in D(x_j), \forall i \ occ(v_i, (w_1, \dots, w_n)) = o_i \in D(c_{v_i})\}.\end{array}$$

An example of a problem that can be modeled with a `gcc` is the *shift assignment problem* [13, 57] in which we are given a set of workers $W = \{W_1, \dots, W_s\}$ and a set of shifts $S = \{S_1, \dots, S_t\}$ and the problem is to assign each worker to one of the shifts while fulfilling the constraints posed by the workers and the boss: Each worker W_i specifies in which of the shifts she is willing to work and for each shift S_i the boss specifies a lower and upper bound on the number of workers that should be assigned to this shift. In the `gcc`, the workers would be represented by the assignment variables and the shifts by the count variables. The domain of an assignment variable would contain the set of shifts that the respective worker is willing to work in and the interval corresponding to each count variable would match the lower and upper bounds specified by the boss for this shift.

7.2.5 The Global Cardinality Constraint with Costs

The *global cardinality constraint with costs* [58] combines a `gcc` and a variant of the `sum` constraint. As in Section 7.2.4, let $X = \{x_1, \dots, x_n\}$ be a set of assignment variables and let $c_{v_1}, \dots, c_{v_{n'}}$ be count variables. We are given a function w that associates to each pair $(x, d) \in X \times D(X)$ a "cost" $w(x, d) \in \mathbb{Q}$. In addition, the constraint is defined on a "cost" variable z with domain $D(z)$. Assuming that we want to *minimize* the cost variable z, the global cardinality constraint with costs is defined as

$$\begin{aligned} &\texttt{cost\_gcc}(x_1, \dots, x_n, c_{v_1}, \dots, c_{v_{n'}}, z, w) = \{(d_1, \dots, d_n, o_1, \dots, o_{n'}, d) \mid \\ &\quad (d_1, \dots, d_n, o_1, \dots, o_{n'}) \in \texttt{gcc}(x_1, \dots, x_n, c_{v_1}, \dots, c_{v_{n'}}), \\ &\quad \forall i \; d_i \in D(x_i), d \in D(z), \textstyle\sum_{i=1}^{n} w(x_i, d_i) \le d\}. \end{aligned} \tag{7.7}$$

In other words, the cost variable z represents an upper bound on the sum of $w(x_i, d_i)$ for all i. We want to find only those solutions to the `gcc` whose associated cost is not higher than this bound.

As an example of the practical use of a `cost_gcc` we extend the above shift assignment problem. It is natural to assume that different workers perform shifts differently. For example, suppose that we have a prediction of "work output" when we assign a worker to a shift. Denote this output by $O(W, S)$ for each worker W and shift S. The boss now wants to maximize the output, while still respecting the above preferences and constraints on the shifts. We can model this as

$$\texttt{cost\_gcc}(W_1, \dots, W_n, S_1, \dots, S_t, z, \tilde{O}),$$

where $\tilde{O}(W, S) = -O(W, S)$ for all workers W and shifts S. Namely, maximizing O is equivalent to minimizing $-O$.

7.2.6 Scheduling with Cumulative Resource Constraints

An important application area for constraint solvers is in solving NP-hard scheduling problems. Chapter 22, "Planning and Scheduling", explores the use of constraint programming for scheduling in depth. Here, we mention only one problem of this family; that of scheduling non-preemptive tasks who share a single resource with bounded capacity.

We are given a collection $T = t_1, \dots, t_n$ of tasks, such that each task t_i is associated with four variables: Its *release time* r_i is the earliest time at which it can begin executing, its *deadline* d_i is the time by which it must complete, its *processing time* p_i is the amount of time it takes to complete and its *capacity requirement* c_i is the capacity of the resource that it takes up while it executes. In addition, we are given the capacity variable $\mathcal{C}$ of the resource. (The special case in which $\forall_i \; c_i = 1$ and $\mathcal{C} = 1$ is known as the *disjunctive* case while the general case in which arbitrary capacities are allowed is the *cumulative* case.)

A solution is a schedule, i.e., a starting time s_i for each task t_i such that $r_i \le s_i \le d_i - p_i$ (the task completes before its deadline), and in addition,

$$\forall u \qquad \sum_{i \mid s_i \le u \le s_i + p_i} c_i \le \mathcal{C}$$

i.e., at any time unit u, the capacity of the resource is not exceeded. Note that the starting times s_i are auxiliary variables; instead of s_i we reason about the release times r_i and deadlines d_i.

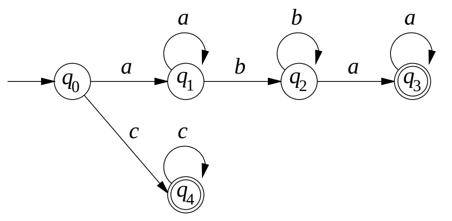


Figure 7.1: A representation of a DFA with each state shown as a circle, final states as a double circle, and transitions as arcs.

The `cumulative`$(\{r_1,\ldots,r_n\},\{d_1,\ldots,d_n\},\{p_1,\ldots,p_n\},\{c_1,\ldots,c_n\},\mathcal{C})$ constraint models scheduling problems as described above [1].

7.2.7 The Regular Language Membership Constraint

The `regular` constraint [49] is defined on a fixed-length sequence of finite-domain variables and states that the sequence of values taken by these variables belongs to a given regular language. The `regular` constraint has applications, for example, in rostering problems and sequencing problems.

Before we formally introduce the `regular` constraint, we need some definitions (see [29]). A *deterministic finite automaton* (DFA) is described by a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final (or accepting) states. Given an input string, the automaton starts in the initial state q_0 and processes the string one symbol at the time, applying the transition function δ at each step to update the current state. The string is *accepted* if and only if the last state reached belongs to the set of final states F. Strings processed by M that are accepted are said to belong to the language defined by M, denoted by $L(M)$. As an example, the DFA M for the regular expression $aa^\star bb^\star aa^\star + cc^\star$ is shown in Figure 7.1. It accepts the strings $aaabaa$ and cc, but not $aacbba$.

Definition 3 (Regular language membership constraint, [49]) *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of variables with $D(x_i) \subseteq \Sigma$ for $1 \leq i \leq n$. Then*

$$\texttt{regular}(X, M) = \{(d_1, \ldots, d_n) \mid \forall i \; d_i \in D(x_i), d_1 d_2 \cdots d_n \in L(M)\}.$$

Returning to our example, consider the CSP

$$x_1 \in \{a,b,c\}, x_2 \in \{a,b,c\}, x_3 \in \{a,b,c\}, x_4 \in \{a,b,c\},$$
$$\texttt{regular}(x_1, x_2, x_3, x_4, M).$$

One solution to this CSP is $x_1 = a$, $x_2 = b$, $x_3 = a$ and $x_4 = a$.

The `regular` constraint allows us to express many relations between the variables of a sequence. For example, it is possible to express the maximum length of identical consecutive values, also known as the `stretch` constraint [48, 23]. A typical application of the `stretch` constraint is to restrict the maximum number of night shifts in a nurse scheduling problem. Pesant [49] discusses even more complicated patterns.

7.2.8 The Circuit Constraint

Before we introduce the `circuit` constraint, we need the following definition. Consider a permutation $S = s_1, \ldots, s_n$ of $\{1, \ldots, n\}$, i.e., $s_i \in \{1, \ldots, n\}$ and $s_i \neq s_j$ whenever $i \neq j$. Define the set C_S as follows:

$$\begin{array}{l} 1 \in C_S, \\ i \in C_S \Rightarrow s_i \in C_S. \end{array}$$

We say that S is *cyclic* if $|C_S| = n$.

Definition 4 (Circuit constraint, [39]) *Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of variables with respective domains $D(x_i) \subseteq \{1, 2, \ldots, n\}$ for $i = 1, 2, \ldots, n$. Then*

$$\texttt{circuit}(x_1, \ldots, x_n) = \{(d_1, \ldots, d_n) \mid \forall i \; d_i \in D(x_i), d_1, \ldots, d_n \text{ is cyclic}\}.$$

To the variables in Definition 4 we can associate the digraph $G = (X, A)$ with arc set $A = \{(x_i, x_j) \mid j \in D(x_i), 1 \leq i \leq n\}$. An assignment $x_1 = d_1, \ldots, x_n = d_n$ corresponds to the subset of arcs $\tilde{A} = \{(x_i, x_{d_i}) \mid 1 \leq i \leq n\}$. The `circuit` constraint ensures that $\tilde{A}$ is a directed circuit.

A famous combinatorial problem that can be modeled with the `circuit` constraint is the Traveling Salesperson Problem, or TSP [40]: A salesperson needs to find a shortest route to visit n cities exactly once, and return in its starting city.

We model the TSP as follows. Let c_{ij} denote the distance between city i and j (where $1 \leq i, j \leq n$). For each city i, we introduce a variable x_i with domain $D(x_i) = \{1, \ldots, n\} \setminus \{i\}$. The value of x_i is the city that is visited by the tour immediately after city i. We also introduce for every $1 \leq i \leq n$ the variable d_i to indicate the distance from city i to city x_i. The TSP can then be modeled as follows.

$$\begin{array}{l} \texttt{minimize } z, \\ \texttt{circuit}(x_1, \ldots, x_n), \\ z = \sum_{i=1}^{n} d_i, \\ d_i = c_{ix_i} \quad 1 \leq i \leq n. \end{array} \tag{7.8}$$

To perform the assignment $d_i = c_{ix_i}$, we use the constraint $\texttt{element}(x_i, d_i, c_{i*})$, where c_{i*} denotes the array $[c_{ij}]_{1 \leq j \leq n}$.

7.2.9 The Soft Alldifferent Constraint

A *soft constraint*, as opposed to a traditional *hard constraint*, is a constraint that may be violated. Instead we measure its violation, and the goal is to minimize the total amount of violation of all soft constraints. Soft constraints are particularly useful to model and solve over-constrained and preference-based problems (see Chapter 9, “Soft Constraints”). In this chapter, we follow the scheme proposed by Régin et al. [60] to soften global constraints.

A *violation measure* for a soft constraint $C(x_1, \ldots, x_n)$ is a function $\mu : D(x_1) \times \ldots \times D(x_n) \to \mathbb{Q}$. This measure is represented by a “cost” variable z, which is to be minimized. There exist several useful violation measures for soft constraints. For the soft `alldifferent` constraint, we consider two measures of violation, see [51]. The first is the *variable-based* violation measure μ_{var} which counts the minimum number of variables that need to change their value in order to satisfy the constraint. The second is the *decomposition-based* violation measure μ_{dec} which counts the number of constraints in the binary decomposition that are violated. For $\texttt{alldifferent}(x_1, \ldots, x_n)$ the latter amounts to $\mu_{\text{dec}}(x_1, \ldots, x_n) = |\{(i,j) \mid \forall i < j \;\; x_i = x_j\}|$.

Definition 5 (Soft alldifferent constraint, [51]) *Let $x_1, x_2, \ldots, x_n, z$ be variables with respective finite domains $D(x_1), D(x_2), \ldots, D(x_n), D(z)$. Let μ be a violation measure for the* `alldifferent` *constraint. Then*

$$\begin{array}{l}\texttt{soft\_alldifferent}(x_1, \ldots, x_n, z, \mu) = \\ \quad \{(d_1, \ldots, d_n, d) \mid \forall i \; d_i \in D(x_i), d \in D(z), \mu(d_1, \ldots, d_n) \leq d\}\end{array}$$

is the soft `alldifferent` *constraint with respect to μ.*

As stated above, the cost variable z is minimized during the solution process. Thus, $\max D(z)$ represents the maximum value of violation that is allowed, and $\min D(z)$ represents the lowest possible value of violation.

As an example, consider the following over-constrained CSP

$$\begin{array}{l}x_1 \in \{a,b\}, x_2 \in \{a,b\}, x_3 \in \{a,b\}, x_4 \in \{b,c\}, \\ \texttt{alldifferent}(x_1, x_2, x_3, x_4).\end{array}$$

We have, for instance, $\mu_{\text{var}}(a,a,b,b) = 2$, while $\mu_{\text{dec}}(a,a,b,b) = 2$, and $\mu_{\text{var}}(b,b,b,b) = 3$, while $\mu_{\text{dec}}(b,b,b,b) = 6$. We soften the `alldifferent` constraint using μ_{dec}, and transform the CSP into the following COP

$$\begin{array}{l}z \in \{0, 1, \ldots, 6\}, \\ x_1 \in \{a,b\}, x_2 \in \{a,b\}, x_3 \in \{a,b\}, x_4 \in \{b,c\}, \\ \texttt{soft\_alldifferent}(x_1, x_2, x_3, x_4, z, \mu_{\text{dec}}), \\ \texttt{minimize}\; z.\end{array}$$

A solution to this COP is $x_1 = a$, $x_2 = a$, $x_3 = b$, $x_4 = c$ and $z = 1$.

7.3 Complete Filtering Algorithms

As mentioned in Section 7.1, the filtering task with respect to a constraint C defined on a set of variables X is to remove values from the domains of variables in X without changing the set of solutions to C. We say that the filtering is complete if the removal of any additional value from the domain of any of the variables in X *would* change the set of solutions to C. Formally:

Definition 6 (Generalized arc consistency) *Let C be a constraint on the variables x_1, …, x_k with respective domains $D(x_1), \ldots, D(x_k)$. That is, $C \subseteq D(x_1) \times \ldots \times D(x_k)$. We say that C is* generalized arc consistent *(arc consistent, for short) if for every $1 \leq i \leq k$ and $v \in D(x_i)$, there exists a tuple $(d_1, \ldots, d_k) \in C$ such that $d_i = v$. A CSP is arc consistent if each of its constraints is arc consistent.*

In the literature, arc consistency is also referred to as *hyper-arc consistency* or *domain consistency*. Note that arc consistency only guarantees that each individual constraint has a solution; it does *not* guarantee that the CSP has a solution.

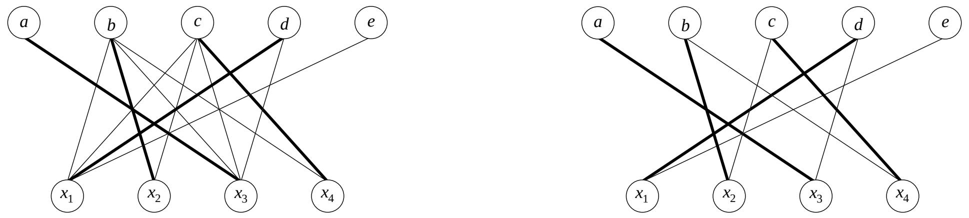


Figure 7.2: Graph representation for the `alldifferent` constraint, before and after filtering. Bold edges represent a matching, corresponding to a solution to the `alldifferent` constraint.

In this section we present filtering algorithms that establish arc consistency. In general, establishing arc consistency for a non-binary constraint (or global constraint) is NP-hard (see Chapter 3, “Constraint Propagation”). For a number of global constraints, however, it is possible to establish arc consistency quite efficiently. We present such filtering algorithms in detail for the `alldifferent`, the `gcc`, and the `regular` constraints.

7.3.1 The Alldifferent Constraint

Régin [56] proposed an arc consistency algorithm for the `alldifferent` constraint which is based on matching theory.

Definition 7 (Value graph, [56]) *Let X be a set of variables and $D(X)$ the union of their domains. The bipartite graph $G = (X, D(X), E)$ with $E = \{\{x, d\} \mid x \in X, d \in D(x)\}$ is called the* value graph *of X.*

As an example, consider the following CSP:

$$x_1 \in \{b, c, d, e\}, x_2 \in \{b, c\}, x_3 \in \{a, b, c, d\}, x_4 \in \{b, c\},$$
$$\texttt{alldifferent}(x_1, x_2, x_3, x_4).$$

The value graph of the variables in this CSP is shown in Figure 7.2.a.

Theorem 4 (Régin [56]) *Let $X = \{x_1, x_2, \dots, x_n\}$ be a set of variables and let G be the value graph of X. Then $(d_1, \dots, d_n) \in \texttt{alldifferent}(x_1, \dots, x_n)$ if and only if $M = \{\{x_1, d_1\}, \dots, \{x_n, d_n\}\}$ is a matching in G.*

Proof: By definition. □

Note that the matching M in Theorem 4 covers X, and is therefore a maximum-cardinality matching.

Consider again the above CSP. A solution to this CSP, i.e., to the `alldifferent` constraint in the CSP, is $x_1 = d$, $x_2 = b$, $x_3 = a$ and $x_4 = c$. This solution corresponds to a maximum-cardinality matching in the value graph, indicated with bold edges in Figure 7.2.a.

Corollary 5 (Régin [56]) *Let G be the value graph of a set of variables $X = \{x_1, x_2, \ldots, x_n\}$. The constraint* `alldifferent`$(x_1, x_2, \ldots, x_n)$ *is arc consistent if and only if every edge in G belongs to a matching in G covering X.*

Proof: Immediate from Definition 6 and Theorem 4. □

The following Theorem identifies edges that belong to a maximum-cardinality matching. The proof follows from [50]; see also [62, Theorem 16.1].

Theorem 6 *Let G be a graph and M a maximum-cardinality matching in G. An edge e belongs to some maximum-cardinality matching in G if and only if $e \in M$, or e is on an even-length M-alternating path starting at an M-free vertex, or e is on an even-length M-alternating circuit.*

Proof: Let M be a maximum-cardinality matching in $G = (V, E)$. Suppose edge e belongs to a maximum-cardinality matching N, and $e \notin M$. The graph $G' = (V, M \oplus N)$ consists of even-length paths (possibly empty) and circuits with edges alternatingly in M and N. If the paths are not of even length, either M or N can be made larger by interchanging edges in M and N along this path (a contradiction because they are of maximum cardinality).

Conversely, let M be a maximum-cardinality matching in G and let P be an even-length M-alternating path starting at an M-free vertex or an M-alternating circuit. Let e be an edge such that $e \in P \setminus M$. Then $M \oplus P$ is a maximum-cardinality matching that contains e. □

Using Theorem 6, we construct the following arc consistency algorithm. First we compute a maximum-cardinality matching M in the value graph $G = (X, D(X), E)$. This can be done in $O(m\sqrt{n})$ time, using the algorithm by Hopcroft and Karp [28], where $m = \sum_{i=1}^{n} |D(x_i)|$. Next we identify the even M-alternating paths starting at an M-free vertex, and the even M-alternating circuits in the following way.

Define the directed bipartite graph $G_M = (X, D(X), A)$ with arc set $A = \{(x, d) \mid x \in X, \{x, d\} \in M\} \cup \{(d, x) \mid x \in X, \{x, d\} \in E \setminus M\}$. In other words, edges in M are oriented from X (the variables) to $D(X)$ (the domain values) and edges not in M are oriented in reverse direction. We first compute the strongly connected components in G_M in $O(n + m)$ time [65]. Arcs between vertices in the same strongly connected component belong to an even M-alternating circuit in G, and are marked as "used". Next we search for the arcs that belong to a directed path in G_M, starting at an M-free vertex. This takes $O(m)$ time, using breadth-first search. Arcs belonging to such a path belong to an M-alternating path in G starting at an M-free vertex, and are marked as "used". For all edges $\{x, d\}$ whose corresponding arc is not marked "used" and that do not belong to M, we update $D(x) = D(x) \setminus \{d\}$. Then, by Theorem 6, the corresponding `alldifferent` constraint is arc consistent.

It follows from the above that the `alldifferent` constraint can be checked for consistency, i.e., determined to contain a solution, in $O(m\sqrt{n})$ time and that it can be made arc consistent in $O(m)$ additional time.

In Figure 7.2.b we have shown the corresponding value graph for our example CSP, after establishing arc consistency. Note that the remaining edges are either in the matching

E_1	E_2	E_3	E_4
[1,3]	[1,2]	[1,1]	[1,1]

$D(x_1)$	$D(x_2)$	$D(x_3)$	$D(x_4)$	$D(x_5)$	$D(x_6)$
{1}	{1,2}	{1,2}	{2}	{2,3,4}	{3,4}

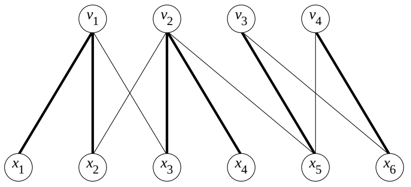


Figure 7.3: `gcc` example: On the left are the domains $D(x_i)$ of the assignment variables and the fixed intervals E_i that replace the count variables. On the right is the corresponding value graph with a solution marked by bold edges.

M (for example x_1d), or on an even-length M-alternating path starting at an M-free vertex (for example ex_1dx_3a), or on an even-length M-alternating circuit (namely $x_2bx_4cx_2$).

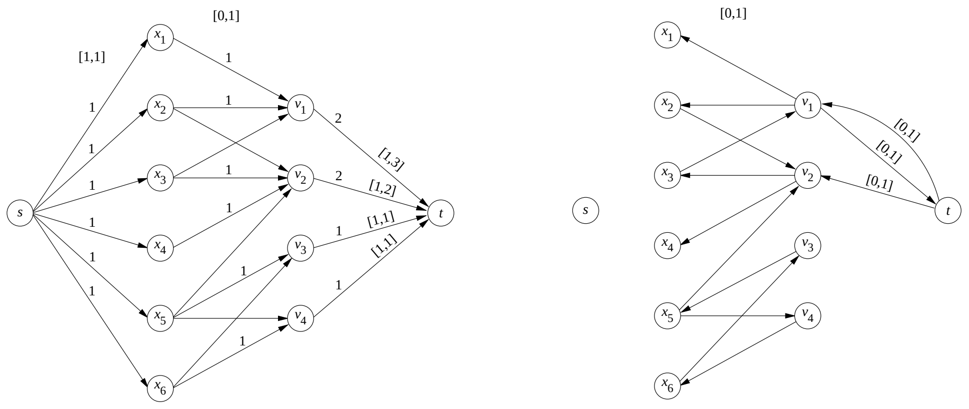
During the whole solution process of the CSP, constraints other than `alldifferent` might also be used to remove values from variable domains. In such cases, we must update the filtering of our `alldifferent` constraint. As pointed out by Régin [56], this can be done incrementally, i.e., we can make use of our current value graph and our current maximum-cardinality matching to compute a new maximum-cardinality matching. For example, if the domain of k variables has changed, we can recompute our matching in $O(\min\{km, m\sqrt{n}\})$ time, and establish arc consistency in $O(m)$ additional time again. The same idea has been used by Barták [4] to make the `alldifferent` constraint dynamic with respect to the addition of variables during the solution process.

7.3.2 The Global Cardinality Constraint

Figure 7.3 shows an example of a `gcc` and one of its solutions. Unfortunately, it is NP-hard to filter the domains of all variables to arc consistency [53]. However, if we replace the count variables $c_{v_1}, \ldots, c_{v_{n'}}$ by constant intervals $E_i = [L_i, U_i]$ ($i = 1, \ldots, n'$), we can use a generalization of the arc consistency algorithm for the `alldifferent` constraint to efficiently filter the domains of all assignment variables to arc consistency [57]: We construct the value graph G as before, orient the arcs from the variables to the values and assign to each of them a requirement of $[0, 1]$. Then, we add two vertices s and t, such that for each variable x_i there is an arc with requirement $[1, 1]$ from s to x_i, and for each value v_j, there is an arc with requirement $[L_j, U_j]$ from v_j to t (see Figure 7.4.a). The following theorem states that a solution to the `gcc` corresponds to an integral feasible s-t flow in this network.

Theorem 7 (Régin [57]) *Let $C =$ `gcc`$(x_1, \ldots, x_n, c_{v_1}, \ldots, c_{v_{n'}})$ and let G be the augmented value graph described above. Then there is a one-to-one correspondence between the solutions to C and integral feasible s-t flows in G.*

Proof: Given a solution $S = (v_{i_1}, \ldots, v_{i_n}, o_1, \ldots, o_{n'})$ to the constraint, we construct a feasible flow in G as follows. For each variable x_j, $f(x_j, v_{i_j}) = 1$ and for any value $v \neq v_{i_j}$, $f(x_j, v) = 0$. For each value v_i, we set $f(v_i, t) = o_i$ and for each variable x_j we set $f(s, x_j) = 1$. It is not hard to verify that the capacities of the arcs are respected by f and that flow conservation holds, so f is an integral feasible s-t flow.



a. Flow network and feasible flow b. Residual graph

Figure 7.4: a. The flow network for the example of Figure 7.3. The requirements of the arcs are shown as intervals above each equal-requirement group. The numbers above the arcs indicate a feasible flow. b. The residual capacity of an arc (v_i, t) indicates how many more variables can be assigned the value v_i without exceeding its capacity and the residual capacity of an arc (t, v_i) indicates how many variables which are assigned v_i can be assigned another value without going below v_i's demand.

Conversely, let f be a feasible flow in G. Then by the demand and capacity requirement, for every arc a from a variable vertex to a value vertex, $f(a) \in \{0, 1\}$. By flow conservation, and by our selection of capacities for the arcs from s to the variable vertex, we know that every variable vertex is incident to exactly one variable-value arc that carries flow 1.

Let $S = (v_{i_1}, \ldots, v_{i_n}, o_1, \ldots, o_{n'})$ be a tuple such that for each $1 \leq j \leq n$, the arc (x_j, v_{i_j}) is the unique arc such that $f(x_j, v_{i_j}) = 1$ and for each $1 \leq j' \leq n'$, $o_{j'}$ is the number of occurrences of the value $v_{j'}$ in $(v_{i_1}, \ldots, v_{i_n})$. To see that S is a solution to the constraint, it remains to show that every variable is assigned a value in its domain. For the assignment variables this is obvious: If a variable-value arc carries flow it must exist in the graph, and this can hold only when the value is in the domain of the variable. For the count variables, this holds, again, by flow conservation and by our choice of capacities for the arcs in the network: The value of the flow on an arc from the value vertex v_i to t is, by construction of the flow network, some value f_i in E_i. By flow conservation, the amount of flow entering this value vertex is also f_i, and since flow can only enter through variable-value arcs, we get that the number of variables that are assigned the value v_i is f_i. $\square$

We say that the arc a belongs to a flow f if $f(a) > 0$. Once again, we conclude that:

Corollary 8 (Régin [57]) *Let G be the value graph of a set of variables $X = \{x_1, \ldots, x_n\}$, augmented into a flow network as described above. The constraint* gcc$(x_1, \ldots, x_n, E_1, \ldots, E_{n'})$*, where each E_i is a fixed interval, is arc consistent if and only if every variable-value arc in G belongs to some feasible integral flow in G.*

The following theorem characterizes the arcs of G that belong to feasible flows, in terms of the residual graph of G with respect to a given flow (see Figure 7.4.b). Its proof is along the same lines as the proof of Theorem 6 and belongs to the folklore of flow theory.

Theorem 9 *Let G be a graph and f a feasible flow in G. An arc belongs to some feasible flow in G if and only if it belongs to f or both of its endpoints belong to the same SCC of the residual graph of G with respect to f.*

Therefore, given a `gcc` whose count variables are fixed intervals, we can filter the domains of the assignment variables to arc consistency by an algorithm that follows the same approach as the arc consistency algorithm for the `alldifferent` constraint, except that the maximum cardinality matching computation is replaced by a feasible flow computation. If we were to use a generic flow algorithm such as Algorithm 1, the running time deteriorates to $O(mn)$. However, Quimper et al. [53] recently showed that the structure of the value graph can be exploited to compute the flow in $O(m\sqrt{n})$ time, using an adaptation of the Hopcroft-Karp algorithm [28] for maximum cardinality bipartite matchings.

7.3.3 The Regular Language Membership Constraint

A filtering algorithm for the `regular` constraint, establishing arc consistency, was presented by Pesant [49]. It makes use of a specific digraph representation of the DFA, which has similarities to dynamic programming.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $X = \{x_1, \dots, x_n\}$ be a set of variables with $D(x_i) \subseteq \Sigma$ for each $1 \leq i \leq n$. We construct the digraph $\mathcal{R}$ representing `regular`(X, M) as follows. The vertex set V consists of $n + 1$ duplicates of the set of states of the DFA:

$$V = V_1 \cup V_2 \cup \dots \cup V_{n+1},$$

where

$$\forall_{1 \leq i \leq n+1} V_i = \{q_k^i \mid q_k \in Q\}.$$

The arc set A of the graph represents the transition function δ of the DFA:

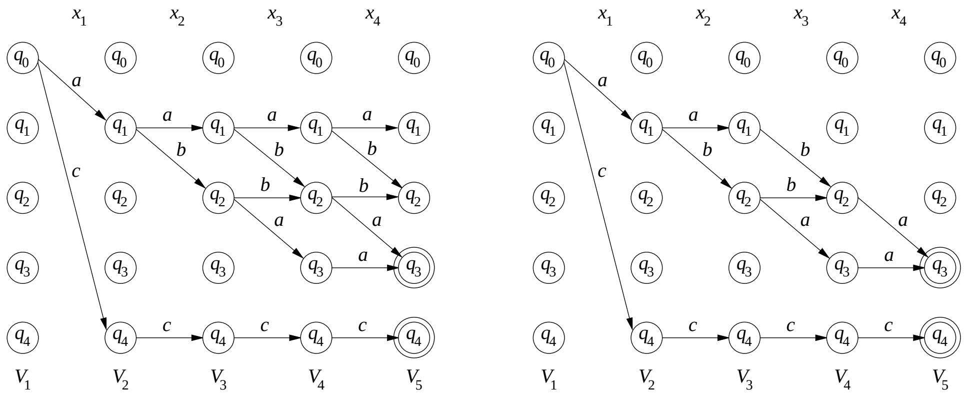
$$A = A_1 \cup A_2 \cup \dots \cup A_n,$$

where

$$\forall_{1 \leq i \leq n} A_i = \{(q_k^i, q_l^{i+1}) \mid \delta(q_k, d) = q_l \text{ for } d \in D(x_i)\}.$$

Figure 7.5.a shows the graph $\mathcal{R}$ corresponding to the DFA in Figure 7.1.

Theorem 10 (Pesant [49]) *A solution to `regular`(X, M) corresponds to a directed path in $\mathcal{R}$ from q_0^1 in V_1 to a final state in V_{n+1}.*



a. Graph representation b. Graph after filtering

Figure 7.5: Graph representation for the `regular` constraint, before and after filtering. A double circle represents a final state. Arcs outgoing from a vertex which is not reachable from q_0^1 were omitted for clarity.

Proof: Follows immediately from the construction of $\mathcal{R}$ and the definition of the `regular` constraint. □

We apply Theorem 10 to establish arc consistency for the `regular` constraint:

Corollary 11 (Pesant [49]) *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $X = \{x_1, \ldots, x_n\}$ be a set of variables with $D(x_i) \subseteq \Sigma$ for $1 \leq i \leq n$. The constraint `regular`(X, M) is arc consistent if and only if for all $x_i \in X$ and $d \in D(x_i)$, there exists an arc $a = (q_k^i, q_l^{i+1})$ such that $\delta(q_k, d) = q_l$ and a belongs to a path from q_0^1 to a final state in V_{n+1}.*

Consider again the example presented in Section 7.2.7, i.e.,

$$x_1 \in \{a, b, c\}, x_2 \in \{a, b, c\}, x_3 \in \{a, b, c\}, x_4 \in \{a, b, c\},$$
$$\texttt{regular}(x_1, x_2, x_3, x_4, M).$$

The CSP is not arc consistent. For example, value b can never be assigned to x_1. If we make the CSP arc consistent we obtain

$$x_1 \in \{a, c\}, x_2 \in \{a, b, c\}, x_3 \in \{a, b, c\}, x_4 \in \{a, c\},$$
$$\texttt{regular}(x_1, x_2, x_3, x_4, M).$$

In Figure 7.5.b, the graph $\mathcal{R}$ corresponding to this example is shown after establishing arc consistency.

Corollary 11 implies the following filtering algorithm. First, we construct the graph $\mathcal{R}$, referred to in [49] as the “forward” phase. During this phase we omit all arcs that are not on a directed path starting in q_0^1. Then we remove all arcs that are not on a path from q_0^1 to a final state in V_{n+1}. This can be done in a “backward” phase, starting from vertices in V_{n+1} which are not final states. The total time complexity of this algorithm is dominated by the time to construct the graph, which is in $O(n\,|\Sigma|\,|Q|)$. This is also the space complexity of the algorithm.

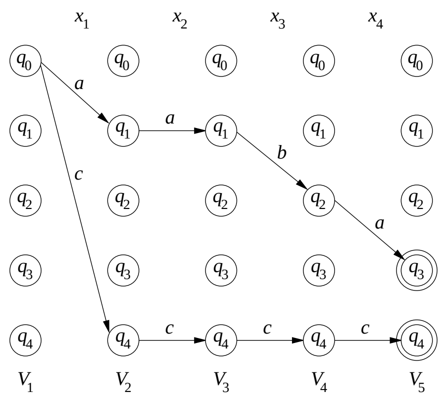


Figure 7.6: Updated graph after the removal of element a from $D(x_3)$.

Note that the algorithm can be made incremental. Whenever the domain of a variable has changed, we remove the corresponding arc from the graph. Then we simply perform a forward and backward phase on the affected parts of the graph, while leaving the rest unchanged. An example is given in Figure 7.6. It shows the updated graph after the removal of element b from $D(x_2)$. As a result, a is removed from $D(x_3)$.

It should be noted that this algorithm resembles the filtering algorithm for the `knapsack` constraint proposed by Trick [66]. Trick's algorithm applies dynamic programming techniques to establish arc consistency on the `knapsack` constraint. The same algorithm can be applied to make the `sum` constraint arc consistent. It has a pseudo-polynomial running time however, as its complexity depends on the actual values of the domain elements of the variable which represents the sum.

7.4 Optimization Constraints

In this section we consider global constraints in the context of constraint optimization problems, or COPs. Recall that a COP contains an objective function to be optimized, and the goal is to find a solution that minimizes or maximizes its value. An *optimization constraint* is a constraint that is linked to the objective function of the problem at hand. For example, the `cost_gcc` is an optimization constraint. Every solution to it induces a "cost" that is represented by a variable z. The assumption is that z appears in the objective function, and is to be minimized. Whenever a solution to the COP is found, we obtain an upper bound for the variable z. Then the domain of z is filtered accordingly, and from that point on, we will only be searching for improving solutions.

Traditionally, COPs were solved in the following way. Assume that the objective function is represented by a variable z, which is to be minimized. If we find a solution to the problem, we compute its corresponding objective value opt and add the constraint $z < opt$. In that way, we search only for improving solutions. By reasoning on the domains of the variables present in the objective function, we may even detect sub-optimality before instantiating all variables, and backtrack. A major deficiency of this method, however, is that there is no inference from the domain of z to the domains of the other variables. Optimization constraints do take this two-way inference into account. They are global constraints, i.e., they specify a complex relation on a set of variables, but in addition they are also defined on a variable such as z above, which represents the value of the best solution found so far. Since we are only interested in improving solutions, a minimization (maximization) constraint is satisfied only when the value of the solution is at most (at least) z.

In this section we present complete filtering algorithms for two types of optimization constraints. First, we consider the `cost_gcc`, which embodies the natural extension of global constraints to optimization constraints. Next, we consider the `soft_alldifferent` constraint, which can be applied to over-constrained and preference-based problems. In Section 7.5.2 we discuss *partial* filtering methods for optimization constraints.

7.4.1 The Global Cardinality Constraint with Costs

The filtering algorithm for the global cardinality constraint with costs (the `cost_gcc`) is an extension of the filtering algorithm of the `gcc` without costs. As in Section 7.3.2, we replace the count variables $c_{v_1}, \dots, c_{v_{n'}}$ by constant intervals $E_1, \dots, E_{n'}$ and filter the domains of the assignment variables.

Let $X = \{x_1, \dots, x_n\}$, $E = \{E_1, \dots, E_{n'}\}$ and let `cost_gcc`(X, E, z, w) be the constraint under consideration in this section. We extend the graph G of Section 7.3.2 by applying a "weight" function to its arcs. The weight of arc (x_i, d) is $w(x_i, d)$ for all $1 \le i \le n$ and $d \in D(x_i)$. To all other arcs we assign a weight 0. The filtering algorithm is based on finding a flow in the weighted version of G, which we denote by $\mathcal{CG}$.

Theorem 12 (Régin [58]) *The constraint* `cost_gcc`(X, E, z, w) *is arc consistent if and only if*

$i)$ *for all $x \in X$ and $d \in D(x)$ there exists an integral feasible s-t flow f in $\mathcal{CG}$ with $f(x, d) = 1$ and weight$(f) \le \max D(z)$, and*

$ii)$ $\min D(z) \ge$ *weight(f) for some integral feasible s-t flow f in $\mathcal{CG}$.*

Proof: If we ignore the costs, we know from the `gcc` case that there is a one-to-one correspondence between integral feasible s-t flows and solutions to the constraint. By our choice of weights for the arcs, the weight of a flow is equal to the cost of the corresponding solution. Hence, a flow corresponds to a solution only if its weight is at most $\max D(z)$ and every value in $D(z)$ (in particular, $\min D(z)$) must be larger than the weight of at least one feasible integral s-t flow. □

Theorem 12 gives rise to the following filtering algorithm for the `cost_gcc`. We first build the digraph $\mathcal{CG}$ that represents the constraint. Then, for every variable-value pair (x_i, d) we check whether the pair belongs to a solution, i.e., whether there exists a flow in $\mathcal{CG}$ that represents a solution containing $x_i = d$, with cost at most $\max D(z)$. If this is not the case, we can remove d from $D(x_i)$. Finally, we update $\min D(z)$ to be the maximum between its current value and the weight of a minimum-weight s-t flow of value n in $\mathcal{CG}$.

By applying the successive shortest paths algorithm described in Section 7.1, we can compute a minimum-weight flow in $\mathcal{CG}$ in $O(n(m + n \log n))$ time. Hence, the time complexity of this filtering algorithm is $O(n^2 d(m + n \log n))$ where d is the maximum domain size. However, we can improve the efficiency by applying Theorem 2, as proposed by Régin [58, 59].

The resulting, more efficient, algorithm is as follows. We first compute an initial minimum-weight flow f in $\mathcal{CG}$ representing a solution. Then for each arc $a = (u, v)$ representing (x_i, d) with $f(a) = 0$, we compute a minimum-weight directed path P from v to u in the residual graph $\mathcal{CG}_f$. Together with a, P forms a directed circuit. Because f represents a solution, it is an integer flow. This means that we can reroute one unit of flow along the circuit and obtain a flow f'. Then cost$(f') =$ cost$(f) +$ cost(P), following Theorem 2. If cost$(f') > \max D(z)$ we remove d from the domain of x_i.

An initial solution is still computed in $O(n(m + n \log n))$ time, but we can reduce the time complexity to establish arc consistency. A first attempt is to compute for all arcs (x_i, d) with $f(x_i, d) = 0$ a shortest path in the residual graph. That would yield a time complexity $O((m-n)(m+n \log n))$. We can do better, however, see [58, 59]. We compute for each (variable) vertex in X the distance to all other vertices in $O(m + n \log n)$ time. Alternatively, this may be done for all (value) vertices in $D(X)$ instead. This gives us the lengths of all paths in $O(\Delta(m + n \log n))$ time, where $\Delta = \min(n, |D(X)|)$.

In addition, this algorithm is incremental. When the domain of k variables has changed, it takes $O(k(m + n \log n))$ time to recompute a feasible flow, starting from the previous flow. Establishing arc consistency is done again in $O(\Delta(m + n \log n))$ additional time.

Note that, by definition (7.7), we don't restrict all values of $D(z)$ to belong to a solution. This would however be the case if we had defined $\sum_{i=1}^{n} w(x_i, d_i) = d$ in (7.7). The reason for omitting this additional restriction on z is that it makes the task of establishing arc consistency NP-hard. This follows from a reduction from the "subset sum" problem (see [20]). Definition (7.7) does allow an efficient filtering algorithm, as we have seen above. In a sense, one could argue that while establishing arc consistency, the algorithm mimics the establishment of bound consistency (see Section 7.5.1) with respect to the cost variable z.

7.4.2 The Soft Alldifferent Constraint

In this section we present filtering algorithms for the `soft_alldifferent` constraint. Each of the violation measures μ_{var} and μ_{dec} gives rise to a different arc consistency problem, and we describe an algorithm for each of them.

Variable-Based Violation Measure

Recall that the variable-based violation measure μ_{var} counts how many variables need to change their values in order for the constraint to be satisfied.

Theorem 13 (Petit et al. [51]) *Let G be the value graph of the variables $x_1, \dots, x_n$ and let M be a maximum-cardinality matching in G. The constraint* `soft_alldifferent`$(x_1, \dots, x_n, z, \mu_{\text{var}})$ *is arc consistent if and only if one of the following conditions holds*

$i)$ $\min D(z) \leq n - |M| < \max D(z)$*, or*

$ii)$ $\min D(z) \leq n - |M| = \max D(z)$ *and all edges in G belong to a matching in G with cardinality $|M|$.*

Proof: We can assign $|M|$ variables to a different value. Thus we need to change the value of at least $n - |M|$ variables, i.e., $\mu_{\text{var}} \geq n - |M|$. Given an assignment with minimum violation, every change in this assignment can only increase μ_{var} by 1. Hence, if $\min D(z) \leq n - |M| < \max D(z)$ all domain values belong to a solution. On the other hand, if $n - |M| = \max D(z)$, only those edges that belong to a matching with cardinality $|M|$ belong to a solution. □

The constraint `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{var}})$ can be filtered to arc consistency by an algorithm which is similar to the one in Section 7.3.1. First we compute a maximum-cardinality matching M in the value graph G in $O(m\sqrt{n})$ time, where $m = \sum_{i=1}^{n} |D(x_i)|$. If $n - |M| > \max D(z)$, the constraint is inconsistent. Otherwise, if $n - |M| = \max D(z)$, we identify all edges that belong to a maximum-cardinality matching. Here we apply Theorem 6, i.e., we identify the even M-alternating paths starting at an M-free vertex, and the even M-alternating circuits. This takes $O(m)$ time, as we saw in Section 7.3.1. Note that in this case vertices in X may also be M-free. Finally, we update $\min D(z) \leftarrow \max\{\min D(z), n - |M|\}$ if $\min D(z) < n - |M|$.

Decomposition-Based Violation Measure

Recall that the decomposition-based violation measure counts the number of constraints in the binary decomposition (i.e., the set of pairwise not-equal constraints) that are violated.

Once again, we construct a directed graph $\mathcal{S} = (V, A)$, this time with

$$V = \{s, t\} \cup X \cup D(X) \quad \text{and} \quad A = A_X \cup A_s \cup A_t$$

where $X = \{x_1, \ldots, x_n\}$ and

$$\begin{array}{rcl} A_X & = & \{(x_i, d) \mid d \in D(x_i)\}, \\ A_s & = & \{(s, x_i) \mid 1 \leq i \leq n\}, \\ A_t & = & \{(d, t) \mid d \in D(x_i), 1 \leq i \leq n\}. \end{array}$$

Note that A_t contains parallel arcs if two or more variables share a domain value. If there are k parallel arcs (d, t) between some $d \in D(X)$ and t, we distinguish them by numbering the arcs as $(d, t)_0, (d, t)_1, \ldots, (d, t)_{k-1}$ in a fixed but arbitrary order.

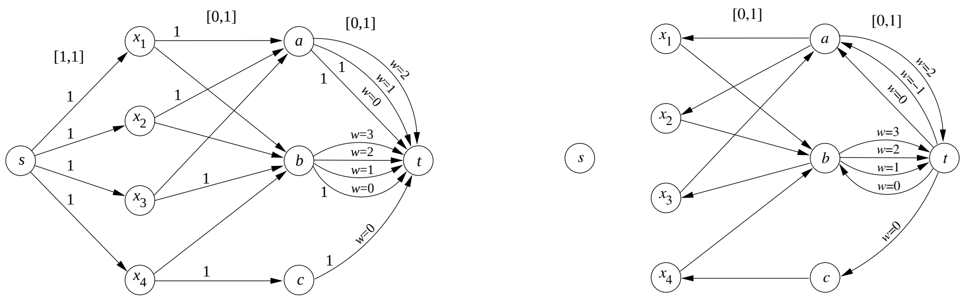
To the arcs in A_s we assign a requirement $[1, 1]$ while the arcs in $A \setminus A_s$ have requirement $[0, 1]$. We also assign a "cost" function w to the arcs. If $a \in A_s \cup A_X$, then $w(a) = 0$. If $a \in A_t$, such that $a = (d, t)_i$ for some $d \in D(X)$ and integer i, the value of $w(a) = i$.

Figure 7.7.a shows the graph $\mathcal{S}$ corresponding to the `soft_alldifferent` example presented in Section 7.2.9.

Theorem 14 (van Hoeve [26]) *The constraint* `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{dec}})$ *is arc consistent if and only if*

$i)$ *every arc $a \in A_X$ belongs to some feasible integral flow f in $\mathcal{S}$ with weight$(f) \leq \max D(z)$, and*

$ii)$ $\min D(z) \geq$ *weight(f) for a minimum-weight s-t flow f in $\mathcal{S}$.*



a. Flow network and feasible flow b. Residual graph

Figure 7.7: Graph representation for the `soft_alldifferent` constraint. The requirements of the arcs are shown as intervals above each equal-requirement group. Unless indicated otherwise, the weight w of an arc is 0. The numbers next to the arcs describe a feasible flow with weight 1.

Proof: Similar to the proof of Theorem 12. The weights on the arcs in A_t are chosen such that the weight of a minimum-cost flow is exactly the smallest possible value of μ_{dec}. Namely, the first unit of flow entering a value $d \in D(X)$ causes no violation and chooses the outgoing arc with weight 0. The k-th unit of flow that enters d causes $k-1$ violations and chooses the outgoing arc with weight $k-1$. $\square$

Once again, we can filter the constraint `soft_alldifferent`$(x_1, \dots, x_n, z, \mu_{\text{dec}})$ to arc consistency by an algorithm which is similar to the one in Section 7.3.1. First we compute a minimum-cost flow f in $\mathcal{S}$. We apply the successive shortest paths algorithm, i.e., we need to compute n shortest paths in the residual graph. Because there are non-zero weights only on arcs in A_t, each shortest path computation takes $O(m)$ time, using a breadth-first search. Hence we can find f in $O(nm)$ time. If weight$(f) > \max D(z)$, we know that the constraint is inconsistent.

To identify the arcs $a = (x_i, d) \in A_X$ that belong to a feasible integral flow g with weight$(g) \le \max D(z)$, we again apply Theorem 2. Thus, we search for a shortest d-x_i path in $\mathcal{S}_f$ that together with a forms a directed circuit C. We can compute all such shortest paths in $O(m)$ time, using again the fact that only arcs $a \in A_t$ contribute to the cost of such paths (more details are given in [26]).

In [27], the above algorithm was extended to other soft global constraints, such as the soft `regular` constraint and the soft `gcc` constraint. The result for the soft `regular` constraint was obtained independently in [6].

7.5 Partial Filtering Algorithms

The algorithms we have presented so far achieve perfect filtering: The removal of any additional value from the domain of any variable would change the solution set of the constraint. Sometimes, achieving this utopic goal is too costly, even intractable, and it makes sense to compromise on a weaker level of filtering. This section describes some of the approaches that have been suggested for partial filtering of global constraints.

7.5.1 Bound Consistency

Assume that the elements of the variable domains are drawn from a total order (e.g., the integers) and that the domain of each variable x_i is an interval of this total order. Thus, a domain $D(x) = [L(x), U(x)]$ is specified by a lower bound and an upper bound on the values that variable x can take.

Definition 8 (Bound consistency) *Let C be a constraint on the variables $x_1, \ldots, x_k$ with respective interval domains $D(x_1), \ldots, D(x_k)$. We say that C is* bound consistent *if for every $1 \leq i \leq k$, there exists a tuple $(d_1, \ldots, d_k) \in C$ such that $d_i = L(x_i)$ and there exists a tuple $(e_1, \ldots, e_k) \in C$ such that $e_i = U(x_i)$.*

Computing bound consistency, then, amounts to shrinking the domain intervals as much as possible without losing any solutions.

Bound Consistency for `alldifferent` and `gcc`

The assumption that the domain of each variable is an interval of the values, implies that the value graph is convex:

Definition 9 (Convex graph) *A bipartite graph $G = (X, Y, E)$ is* convex *if the vertices of Y can be assigned distinct integers from $[1, |Y|]$ such that for every vertex $x \in X$, the numbers assigned to its neighbors form a subinterval of* $[1, |Y|]$.

Algorithms for computing bound consistency exploit this property of the value graph (either directly or implicitly). Naturally, filtering algorithms for `alldifferent` appeared first and the generalizations to `gcc` followed. Two parallel approaches were explored (see Table 7.1). The first is an adaption of the matching/flow method described above and the second is based on Hall's marriage theorem.

Theorem 15 (Hall's Marriage Theorem [22]) *A bipartite graph $G = (X, Y, E)$ has a matching covering X if and only if for any subset X' of X, we have that $|D(X')| \geq |X'|$.*

In our terminology: there is a solution to an `alldifferent` constraint if and only if for every subset of the variables, the union of their domains contains enough values to match each of them with a different value. This theorem implies that if there is a set S of k variables whose domains are contained in a size-k interval I of values, then the values of I can be safely removed from the domain of any variable outside of S. It also implies that this filtering step suffices: If it cannot be applied, the `alldifferent` constraint is bound consistent.

As we saw, the flow-based approach yields both arc consistency and bound consistency algorithms. The second approach, using Hall's marriage theorem, was first applied by Leconte [41] who obtained an algorithm that computes *range consistency*, a filtering level which is stronger than bound consistency but weaker than arc consistency. Subsequently, Hall's theorem was also used in bound consistency algorithms.

In the following, n denotes the number of variables, n' denotes the number of values in the union of their domains and m denotes the sum of the cardinalities of the domains (so the value graph has $n + n'$ vertices and m edges). Since m may be as large as nn', bound consistency algorithms typically do not construct the graph explicitly.

	Hall's Theorem	Matchings/Flows	
	bound consistency	arc consistency	bound consistency
`alldifferent`	Puget [52], López-Ortiz et al. [43]	Régin [56]	Mehlhorn and Thiel [44]
`gcc`	Quimper et al. [54]	Régin [57]	Katriel and Thiel [35]

Table 7.1: The two approaches for filtering of `alldifferent` and `gcc` constraints.

Puget designed the first bound consistency algorithm for `alldifferent`, which is based on Hall's theorem and runs in $O(n \log n)$ time [52]. Mehlhorn and Thiel [44] later showed that since the matching and SCC computations of Régin's algorithm [56] can be performed faster on convex graphs compared to general graphs, it is possible to achieve bound consistency for `alldifferent` using the matching approach in $O(n + n')$ time plus the time required to sort the variables according to the endpoints of their domains. Katriel and Thiel [35] later generalized this algorithm for the `gcc` case. Simultaneously, Quimper et al. [54] discovered an alternative bound consistency algorithm for `gcc`, based on the Hall interval approach. The latter algorithm narrows the domains of only the assignment variables, while the former narrows the domains of the assignment variables as well as the count variables, to bound consistency.

As mentioned in Section 7.3.2, it is NP-hard to filter all variables to arc consistency. It is therefore significant that we can achieve at least *some* filtering for the domains of the count variables.

Glover's Algorithm

In order to demonstrate how much simpler convex bipartite graphs are from general bipartite graphs, we describe a simple, greedy algorithm that finds a maximum cardinality matching in a convex value graph. Glover [21] was the first who suggested this algorithm as an $O(nn')$-time solution. Using sophisticated data structures, the complexity was later reduced to $O(n' + n\alpha(n))$ by Lipski and Preparata [42] and finally to $O(n' + n)$ by Gabow and Tarjan [19]. The latter solutions assume that the values are integers in the interval $[1, n']$ (which can be achieved in $O(n' \log n')$ time by sorting and relabeling them). We will restrict our description to a simple implementation of Glover's algorithm, which uses only a priority queue and does not require that the values are in $[1, n']$. This implementation runs in $O(n' + n \log n)$ time. It is much faster than the best known solution for general value graphs which, recall, runs in $O(m\sqrt{n})$ time [28].

The algorithm traverses the value vertices from smaller to larger and greedily decides, for each value vertex, whether it is to be matched and if so, with which variable vertex. For this purpose, it maintains a priority queue that contains variable vertices which are candidates for matching, sorted by the upper endpoints of their domains. When considering the value vertex v_i, the algorithm first inserts into the queue all variable vertices whose domains begin at v_i; they were not candidates for matching before, but they are now.

Next, there are two cases to consider. If the priority queue is empty, v_i will remain unmatched. Otherwise, the minimum priority variable vertex x_j is extracted, and there are two subcases. If x_j's priority is at least v_i, then it is matched with v_i. Otherwise, it should have been matched earlier, and the algorithm terminates and reports that there is no solution (the graph does not have a matching covering X, or, equivalently, the `alldifferent` constraint does not have a solution).

The intuition behind this algorithm is that it always matches the candidate variable vertex whose domain ends earliest, so when x_j is matched, any candidate vertex that remains unmatched can be matched with at least as many value vertices as x_j, but perhaps more. For a formal proof of correctness see [21] or [44].

7.5.2 Reduced-Cost Based Filtering

Next we consider a partial filtering method for optimization constraints of the following type. Let $X = \{x_1, \ldots, x_n\}$ be a set of variables with corresponding finite domains $D(x_1), \ldots, D(x_n)$. We assume that each pair (x_i, j) with $j \in D(x_i)$ induces a "cost" c_{ij}. We now extend any global constraint C on X to an optimization constraint `opt_`C by introducing a cost variable z and defining

$$\begin{aligned}&\texttt{opt\_}C(x_1, \ldots, x_n, z, c) = \{(d_1, \ldots, d_n, d) \mid \\ &\quad (d_1, \ldots, d_n) \in C(x_1, \ldots, x_n), \\ &\quad \forall i \; d_i \in D(x_i), d \in D(z), \textstyle\sum_{i=1}^n c_{id_i} \leq d\}.\end{aligned}$$

where we assume that z is to be minimized. For example, the `cost_gcc` is a particular instance of such constraint. We have seen that its arc consistency algorithm is efficient because of its correspondence with a minimum-weight flow. For many other optimization constraints of this type, however, such correspondence does not exist, or is difficult to identify. In such situations we may be able to apply *reduced-cost based filtering* instead, using a linear programming relaxation of the optimization constraint. This method was first introduced in this form by Focacci et al. [17], although the technique is part of the linear programming folklore under the name *variable fixing*. Note that in general, such a filtering algorithm does not establish arc consistency.

In order to apply reduced-cost based filtering, we need to infer a linear programming relaxation from the optimization constraint. First, we introduce binary variables y_{ij} for all $i \in \{1, \ldots, n\}$ and $j \in D(x_i)$, such that

$$\begin{aligned}x_i = j \Leftrightarrow y_{ij} = 1, \\ x_i \neq j \Leftrightarrow y_{ij} = 0.\end{aligned} \tag{7.9}$$

To ensure that each variable x_i is assigned to a single value in its domain we state the linear constraints

$$\sum_{j \in D(x_i)} y_{ij} = 1 \quad \text{for } i = 1, \ldots, n.$$

The linear objective function is stated as

$$\sum_{i=1}^{n} \sum_{j \in D(x_i)} c_{ij} y_{ij}.$$

The next, most difficult, task is to rewrite (a part of) the optimization constraint as a system of linear constraints using the binary variables. This is problem dependent, and no general recipe exists. However, for many problems such descriptions are known, see, e.g., [55]. For example, for an `alldifferent` constraint we may add the linear constraints

$$\sum_{i=1}^{n} y_{ij} \leq 1 \text{ for all } j \in \bigcup_{i=1}^{n} D(x_i)$$

to ensure that every domain value is assigned to at most one variable.

Finally, in order to obtain a linear programming relaxation, we remove the integrality constraint on the binary variables and state

$$0 \leq y_{ij} \leq 1 \text{ for } i \in \{1, \ldots, n\}, j \in D(x_i).$$

When we solve this linear programming relaxation to optimality, we obtain a lower bound on z, and reduced-costs $\overline{c}$. Recall from Section 7.1.3 that reduced-costs estimate the increase of the objective function when we force a variable into the solution. Hence, if we enforce the assignment $x_i = j$, the objective function value will increase by at least $\overline{c}_{ij}$. Let z^* be the objective value of the current optimal solution of the linear program. Then we apply the following filtering rule:

$$\text{if } z^* + \overline{c}_{ij} > \max D(z) \text{ then } D(x_i) \leftarrow D(x_i) \setminus \{j\}.$$

A huge advantage of this approach is that it can be applied very efficiently. Namely, reduced-costs are obtained automatically when solving a linear program. Hence, the filtering rule can be applied without additional computational costs.

7.5.3 Intractable Global Constraints

As already noted, global constraints serve to break up the CSP into a conjunction of simpler CSPs, each of which can be filtered efficiently. We show below that if it is NP-hard to determine whether a constraint has a solution, it is also NP-hard to compute arc consistency for the constraint. The following is a special case of a Theorem due to Bessière et al. [10].

Theorem 16 *Let C be a constraint. If there is a polynomial-time algorithm that computes arc consistency for C then there is a polynomial-time algorithm that finds a single solution to C.*

Proof: Assume that we have an algorithm A that prunes the variable domains to arc consistency in polynomial time. Then we can find a solution to the constraint as follows:

1. Use algorithm A to compute arc consistency. The constraint has a solution if and only if all domains are now non-empty.
2. Repeat until a solution is found:
    a) Let x be a variable such that $|D(x)| > 1$ and let $v \in D(x)$.
    b) Set $D(x) \leftarrow \{v\}$
    c) Use algorithm A to compute arc consistency.

In each iteration the value of one variable is determined, so the total number of iterations is at most equal to the number of variables and the running time of the algorithm is polynomial. □

The converse of Theorem 16 does not hold; there are constraints for which arc consistency is NP-hard while checking feasibility is not (see, e.g., [64]). A weaker version which does hold is stated below. The crucial point to note is that there are constraints for which it is possible to efficiently check whether the constraint has a solution, but it is NP-hard to check whether it has a solution in which a certain variable is assigned a specific value in its domain.

Theorem 17 *Let C be a constraint defined on the variables $X = \{x_1, \ldots, x_k\}$. If there is an algorithm A that, for any $x_i \in X$ and $d \in D(x_i)$, determines in polynomial time whether there is a solution to the constraint $C \wedge (x_i \leftarrow d)$, then there is a polynomial-time algorithm that computes arc consistency for C.*

Proof: For every variable x_i and value $d \in D(x_i)$, use algorithm A to check if there is a solution when $x_i \leftarrow d$ and remove d from $D(x_i)$ otherwise. □

A consequence of Theorem 16 is that there is a very large class of practically useful global constraints for which we probably cannot achieve perfect filtering. In some cases, a possible remedy is to compromise on bound consistency; as already mentioned, bound consistency can be computed in almost-linear time for the `gcc`, while arc consistency, for the assignment and count variables, is NP-hard.

Filtering for the Cumulative Constraint

Another method to cope with NP-hardness is to *relax* the constraint. That is, to transform our NP-hard constraint C into a constraint C' such that C' can be efficiently filtered to a guaranteed consistency level (e.g., arc consistency or bound consistency) and $C \subset C'$, i.e., every solution to C is also a solution to C'. For example, the reduced-cost based filtering method described above applies a linear programming relaxation of the constraint. Here we will demonstrate this approach by describing a filtering algorithm for a relaxation of the `cumulative`[3] constraint [45]. We assume for simplicity that the capacity of the resource and the capacity requirements and processing times of the tasks are fixed, i.e., $|D(\mathcal{C})| = 1$ and $|D(c_i)| = |D(p_i)| = 1$ for all i. The filtering task is to increase the minimum start times and decrease the maximum completion times of the tasks, without losing any solutions to the constraint. We will describe the algorithm that tightens the earliest start times; the solution for the latest completion times is symmetric. The relaxation of the `cumulative` constraint will be defined below, but first we wish to build up the intuition behind the definition.

Let the *energy* of task t_i be $e_i = c_i p_i$; it represents the total capacity of the resource that is consumed by the task. For a set $\Omega \subseteq T$ of tasks, let r_Ω be the earliest release time of a task in Ω, d_Ω the latest deadline of a task in Ω and e_Ω the sum of the energies of tasks in Ω. Clearly, if there is a subset $\Omega \subseteq T$ of the tasks such that $e_\Omega > \mathcal{C}(d_\Omega - r_\Omega)$, the

[3]The `cumulative` constraint is, in general, NP-hard. Recently, Artiouchine and Baptiste [3] developed a bound consistency algorithm for the special case in which all processing times are equal.

problem is infeasible: Between time r_Ω and d_Ω, the tasks need more of the resource than is available.

Now, let Ω be a set of tasks and $t_i \notin \Omega$ another task such that $e_{\Omega\cup\{t_i\}} > \mathcal{C}(d_\Omega - r_{\Omega\cup\{t_i\}})$. If t_i is scheduled such that it completes executing before any task in Ω, then it completes before d_Ω, so the total energy of the tasks scheduled in the interval $[r_{\Omega\cup\{t_i\}}, d_\Omega]$ is above the capacity of the resource, a contradiction. So t_i completes execution last among the tasks in $\Omega \cup \{t_i\}$.

Once we have found such a pair (Ω, t_i), we can use it to adjust the starting time of t_i as follows. For each subset $\Theta \subseteq \Omega$, we examine the time interval $I = [r_\Theta, d_\Theta]$ and determine what is the earliest time in this interval at which t_i can start executing. Since we know that t_i cannot complete before any task in Θ, we get that if t_i is scheduled at time unit $u \in I$, then in the interval $[u, d_\Theta]$ the schedule allocates only $\mathcal{C} - c_i$ capacity units of the resource for tasks in Θ.

Conceptually, split the resource into two parts, with capacities $\mathcal{C}_1 = \mathcal{C} - c_i$ and $\mathcal{C}_2 = c_i$. Assume that the schedule placed t_i on the second part and that t_i was the last task scheduled there. Clearly, on the first part we can schedule at most $(\mathcal{C} - c_i)(d_\Theta - r_\Theta)$ units of energy in the time interval I. This means that at least $\mathit{rest}(\Theta, c_i) = e_\Theta - (\mathcal{C} - c_i)(d_\Theta - r_\Theta)$ units of energy must be scheduled in this time interval on the second part just to schedule all the tasks of Θ. Even if all of this energy is scheduled as early as possible, it takes up at least the first $\frac{1}{c_i}\mathit{rest}(\Theta, c_i)$ time units of the second part and therefore t_i cannot begin before time unit $r_\Theta + \frac{1}{c_i}\mathit{rest}(\Theta, c_i)$.

An algorithm that performs all such adjustments to the starting times of tasks is called an *edge-finding* algorithm (because the algorithm discovers edges in the precedence-graph of the completion times of the tasks). The basic idea of such an algorithm is to efficiently identify a small number of pairs (Θ, t_i) for which the rule described above needs to be applied.

Edge-finding algorithms were first developed for the disjunctive case, which is much simpler than the most general case. The fastest algorithm runs in $O(n \log n)$ time [12]. For the cumulative case, the fastest known solution is by Mercier and Van Hentenryck [45] and runs in $O(kn^2)$ where k is the number of different capacity requirements of the tasks (a previously developed $O(n^2)$-time solution was shown to be incomplete).

After giving an outline of the algorithm, we are ready to define the constraint that it filters, i.e., the relaxation of the `cumulative` constraint. Since edge-finding algorithms existed in the scheduling literature before `cumulative` was a global constraint, this definition may seem opportunistic: We define the problem to be whatever we already know how to solve. Nevertheless, scheduling is an important application in constraint programming so we believe that the edge-finding algorithm deserves a description in constraint programming terminology: It is a bound consistency algorithm for the relaxation of the `cumulative` constraint (where the processing times and capacities of the tasks, as well as the capacity of the resource are fixed) which is satisfied if for every task t_i

$$\min\{D(r_i)\} \geq \max_{\substack{\Omega \subseteq T \\ i \notin \Omega \\ \alpha(\Omega, i)}} \; \max_{\substack{\Theta \subseteq \Omega \\ \mathit{rest}(\Theta, c_i)}} \; r_\Theta + \left\lceil \frac{1}{c_i}\mathit{rest}(\Theta, c_i) \right\rceil$$

where $\alpha(\Omega, i) \Leftrightarrow \big(\mathcal{C}(d_\Omega - r_{\Omega\cup\{i\}}) < e_{\Omega\cup\{i\}})\big)$.

Intractable Optimization Constraints

Sellmann [63, 64] suggested two forms of partial consistency, which are specifically motivated by NP-hard optimization constraints. The first is an adaptation of relaxed consistency [64] to optimization constraints. That is, we transform the constraint C into a constraint C' such that $C \subseteq C'$ and C' can be filtered efficiently. The idea is similar to the relaxation of the `cumulative` constraint described above, except that here C and C' are both optimization constraints. The reduced-cost based filtering based on a linear relaxation, which was described in Section 7.5.2, also employs this idea.

Sellmann demonstrates this technique by way of the shorter-path constraint, which is defined on a digraph G, a source vertex s and a target vertex t in G, an upper bound W and a variable P whose domain is all subsets of arcs of G (see Section 7.6.1). The constraint is satisfied if P is a set of arcs that form a path in G from s to t whose length is at most W. Since it is NP-hard to determine whether there is a path from s to t that uses a certain arc (while visiting each node at most once), it is NP-hard to compute bound consistency for the set variables P. However, it is easy to determine whether there is an "almost-path" from s to t that uses the arc (u, v) and whose length is at most the upper bound: Find the length of the shortest path from s to u and the length of the shortest path from v to t. The concatenation of these two paths through the arc (u, v) is a walk from s to t that visits every vertex at most twice. The relaxed shorter-path constraint, then, excludes from the set assigned to P any arc that does not belong to a path or almost-path from s to t in G whose length is at most W.

Sellmann's second form of partial consistency is termed *approximated consistency* [63]. Here, the idea is to use efficient approximation algorithms for NP-hard problems as components of the filtering algorithm. Recall that an α-approximation algorithm for a minimization (maximization) problem P is a polynomial-time algorithm A such that for every instance x of P, A finds a solution whose value is at most $(1 + \epsilon) \cdot Opt(P, x)$ (at least $(1 - \epsilon) \cdot Opt(P, x)$), where $Opt(P, x)$ is the value of the optimal solution to instance x of problem P. Clearly, the smaller the value of α, the better the quality of approximation. $1 + \alpha$ (resp. $1 - \alpha$) is referred to as the *approximation factor* achieved by algorithm A. For more details, see any text on approximation algorithms, such as [25, 68].

For a minimization (maximization) constraint that is defined on a variable z which holds the upper (lower) bound on the value of a solution, we say that C is ϵ-arc consistent if every value in the domain of every variable participates in a solution of value at most $z + \epsilon Opt$ (at least $z - \epsilon Opt$). The motivation behind this definition is that approximation algorithms allow us to efficiently identify problem instances whose optimal solutions are much better or much worse than the best solution found so far, but may give inconclusive replies for instances which are of comparable quality. In such cases, approximate consistency allows one-sided errors: we keep the respective value in the variable domain, to be on the safe side.

7.6 Global Variables

In recent years, some of the work of global constraints, i.e., that of providing more structured information to the solver and simplifying the syntax of CSPs, is taken up by complex variable types, which we will collectively refer to as *global variables*. Our focus in this section is on constraints defined on global variables and the design of filtering algorithms for such constraints. We will discuss two important examples: sets and graphs. Chapter 17, "Beyond Finite Domains", is devoted to the topic of complex variable types, and describes many examples and aspects that are not mentioned here.

7.6.1 Set Variables

Let us revisit the shift-assignment problem for which we used the global cardinality constraint in Section 7.2.4. We assumed that each worker is to work exactly one shift. It is more realistic, however, that we have a lower bound and an upper bound on the number of shifts that each worker is to staff. The result is known as the *symmetric cardinality constraint* [37]:

Definition 10 *The symmetric cardinality constraint* `symcc`$(x_1, \ldots, x_n, c_{x_1}, \ldots, c_{x_n}, c_{v_1}, \ldots, c_{v_{n'}})$ *is defined on a collection of assignment variables* $x_1, \ldots, x_n$ *and two sets of count variables,* $c_{x_1}, \ldots, c_{x_n}$ *and* $c_{v_1}, \ldots, c_{v_{n'}}$*. It specifies that the value assigned to* x_j *is a subset of* $\{v_1, \ldots, v_{n'}\}$ *of cardinality* c_{x_j}*, and that the number of such subsets that contain* v_i *is* c_{v_i}*.*

We still have one variable for each worker, but the value of this variable is the *set* of shifts that the worker will staff. One way to handle this is to say that the domain contains all subsets of the shifts. This results in an exponential growth in the number of values (and hence in the size of the value graph).

An alternative is to use *set variables*. A set variable x is a variable that has a discrete domain $D(x) = [lb(x), ub(x)]$. Thus, the domain of a set variable consists of two sets, the set $lb(x)$ of *mandatory* elements and the set $ub(x) \setminus lb(x)$ of *possible* elements. The value assigned to x should be a set $s(x)$ such that $lb(x) \subseteq s(x) \subseteq ub(x)$.

For a constraint on set variables, we are not interested in arc consistency because the individual values that a set variable can take do not explicitly exist; we only have their intersection (lb) and their union (ub). Viewing the intersection as a lower bound and the union as an upper bound, we speak of bound consistency when filtering the domain of a set variable. A bound consistency computation for a constraint C defined on a set variable x requires that we:

- Remove a value v from $ub(x)$ if there is no solution to C in which $v \in s(x)$.
- Include a value $v \in ub(x)$ in $lb(x)$ if in all solutions to C, $v \in s(x)$.

To demonstrate such a computation[4], we sketch how the flow-based filtering algorithm for `gcc` can be adapted to compute bound consistency for the assignment variables of `symcc`, assuming that the domains of all count variables are fixed intervals. The flow network constructed from the value graph is almost identical, except that the requirement of an arc from s to a variable vertex reflects the cardinality requirement for the set assigned to the variable. That is, the capacity of the arc (s, x_j) is equal to the interval $D(c_{x_j})$. Then, we once again have a one-to-one correspondence between the integral s-t flows in

[4]Additional examples can be found in [9].

the network and the solutions to the constraint. As before, after finding a flow we have that a non-flow arc belongs to some integral s-t flow if and only if its endpoints belong to the same SCC of the residual graph.

However, unlike in the `gcc` case, this does not complete the filtering task: we must also identify arcs that belong to *any* integral s-t flow, and make sure that they are in the lower bounds of the domains of the relevant set variables. It is not difficult to verify that this is exactly the set of flow arcs whose endpoints belong to different SCCs of the residual graph (recall that the requirement of an arc from a variable vertex to a value vertex is $[0, 1]$).

The bottleneck of the algorithm is the flow computation, which takes $O(mn)$ time. It is interesting to note that the cardinality of the domain of any of the set variables may well be exponential in the running time of this algorithm, which handles all of these domains at once.

7.6.2 Graph Variables

A *graph variable* [16] is simply two set variables V and E, with an inherent constraint $E \subseteq V \times V$. As with set variables, the domain $D(G) = [lb(G), ub(G)]$ of a graph variable G consists of mandatory vertices and edges $lb(G)$ (the *lower bound graph*) and possible vertices and edges $ub(G) \setminus lb(G)$ (the *upper bound graph*). The value assigned to the variable G must be a subgraph of $ub(G)$ and a super graph of the $lb(G)$.

The usefulness of graph variables depends on the existence of efficient filtering algorithms for useful constraints defined on them, i.e., constraints that force graph variables to have certain properties or certain relations between them. As a simple example, the constraint $Subgraph(G, S)$ specifies that S is a subgraph of G. Note that both S and G are variables, so computing bound consistency for the $Subgraph$ constraint means the following:

1. If $lb(S)$ is not a subgraph of $ub(G)$, the constraint has no solution.
2. For each $e \in ub(G) \cap lb(S)$, include e in $lb(G)$.
3. For each $e \in ub(S) \setminus ub(G)$, remove e from $ub(S)$.

The conditions above can be checked in time which is linear in the sum of the sizes of $ub(G)$ and $ub(S)$. As with set variables, we are in the interesting situation in which the number of graphs that the bound consistency algorithm reasons about may be exponential in the running-time of the algorithm.

The Spanning Tree Constraint

As a slightly more sophisticated example, we consider the constraint $ST(G, T)$, which states that the graph T is a spanning tree of the graph G. Since a spanning tree is a subgraph, the conditions described above should be checked when computing bound consistency for ST. In addition, (1) the vertex-sets of G and T must be equal, and (2) T must be a tree.

To enforce (1), we remove from $ub(G)$ any vertex which is not in $ub(T)$ and we include in $lb(T)$ any vertex which is in $lb(G)$. As for (2), if $lb(T)$ contains a circuit then T cannot be a tree and if $ub(T)$ is not connected then T cannot be connected. In both cases, the constraint has no solution. Finally, any edge in $ub(T) \setminus lb(T)$ whose endpoints belong to the same connected component of $lb(T)$ must be removed (including it in any solution would introduce a circuit in T) and any bridge[5] in $ub(T)$ must be placed in $lb(T)$ (T cannot be connected if it is excluded).

The running time of the algorithm we described is linear in the sum of the sizes of the upper bounds of G and T. To prove that it achieves bound consistency, one needs to show that the following three conditions hold:

1. Every vertex or edge that was removed, does not participate in any solution.

2. Every remaining vertex or edge in $ub(T)$ or $ub(G)$ participates in at least one solution and every remaining vertex or edge in $ub(T) \setminus lb(T)$ or $ub(G) \setminus lb(G)$ is excluded from at least one solution to the constraint.

3. Every vertex or edge that the algorithm inserts into $lb(G)$ or $lb(T)$ participates in all solutions.

Note that in item 3 above we do not say that every element in $lb(G)$ and $lb(T)$ belongs to all solutions. This is only required of those elements that the filtering algorithm decided to include in the lower bound sets. The input may include any vertex or edge in the lower bound graph, and the filtering algorithm does not ask why: It may only remove values from variable domains, and never add them.

7.7 Conclusion

The search for useful global constraints and the design of efficient filtering algorithms for them is an ongoing research effort that tackles many challenging and interesting problems. We have already mentioned some of the fundamental questions: What are the frequently recurring sub-problems that we would like to capture by global constraints? For a specific constraint, what is the computational complexity of filtering it to arc consistency? Should we compromise on partial consistency? We would like to briefly mention several other ideas on global constraints that have been proposed in recent years.

Given the large number of global constraints that were and will be defined, several researchers are attempting to find generic methods to specify and handle constraints. Beldiceanu et al. [7] describe a constraint solver that views a global constraint in terms of a collection of graph properties (such as the number of strongly connected components in a digraph). Then, the solver uses a database of known graph theoretic results to automatically generate new constraints that strengthen the model by allowing more filtering. They point out that out of the 227 global constraints listed in the global constraints catalog [5], about 200 can be described in terms of graph properties. Therefore, their approach seems to be widely applicable. Bessière et al. [8] defined a declarative language that can be used to specify many known constraints which model counting and occurrence problems. In this language, a constraint is specified as the conjunction of constraints, each of which can be a simple (binary) constraint on scalar or set variables, or one of two globals constraints called `range` and `roots`.

[5]A *bridge* in a graph is an edge whose removal increases the number of connected components.

Another approach is to view the filtering task in the context of the tree search. We have already mentioned the problem of dynamic filtering, i.e., recomputing arc consistency after a small change such as the removal of a few values from variable domains. Recently, Katriel [34] pointed out that in a flow network with n nodes and m edges where every edge belongs to at least one feasible flow, there are only $O(n)$ edges whose removal would render *other* edges useless. This implies that if the filtering is random, i.e., the edge removed from the value graph of an `alldifferent` or `gcc` is always selected at random among all possibilities, the expected number of edges that need to be removed before it makes sense to recompute arc consistency is $\Theta(m/n)$. It would be interesting to evaluate experimentally whether the assumption that filtering is random is realistic, and whether delayed filtering is a good compromise between filtering efficiency and effectiveness. If this approach is to be pursued, it is necessary to either analyze each global constraint independently and determine a reasonable filtering frequency, or find a generic or automated way to do this for many global constraints.

In the area of partial filtering for NP-hard global constraints, there seems to be a lot of potential for enhancements. Here we would like to suggest the idea of approximate filtering. Recall that an approximation algorithm for an optimization problem is an algorithm that finds a solution whose value, according to the objective function of the problem, does not deviate too much from the value of the optimal solution. For a filtering problem, the objective function counts the sum of the cardinalities of the domains of the variables. An optimal solution minimizes this number, and hence an α-approximate solution, for $\alpha \geq 1$, is a solution that removes all but αOpt values from the variable domains. Formally,

Definition 11 (Approximate filtering) *Let $C(x_1, \ldots, x_n)$ be a constraint and assume that after filtering it to arc consistency, the sum of the cardinalities of the domains of $x_1, \ldots, x_n$ is Opt. An α-*approximate filtering algorithm *for C is an algorithm that removes values from the domains of the variables $x_1, \ldots, x_n$ such that the solution set of C remains unchanged and the sum of the cardinalities of the domains of the variables is at most αOpt.*

Note that approximate filtering is different from the notion of approximated consistency that was described in Section 7.5.3 in two ways. First, approximate filtering applies to any constraint while approximated consistency is defined for optimization constraints. In addition, with approximated consistency, what is being approximated is the value of the solutions to the constraint that remain, while approximate filtering directly approximates the effectiveness of the filtering algorithm.

Bibliography

[1] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Journal of Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[2] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows*. Prentice Hall, 1993.

[3] K. Artiouchine and P. Baptiste. Inter-distance Constraint: An Extension of the All-Different Constraint for Scheduling Equal Length Jobs. In P. van Beek, editor, *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP 2005)*, volume 3709 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2005.

[4] R. Barták. Dynamic Global Constraints in Backtracking Based Environments. *Annals of Operations Research*, 118(1–4):101–119, 2003.

[5] N. Beldiceanu, M. Carlsson, and J.X.-Rampon. Global constraint catalog. Technical Report T2005-06, Swedish Institute of Computer Science, 2005.

[6] N. Beldiceanu, M. Carlsson, and T. Petit. Deriving Filtering Algorithms from Constraint Checkers. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2004.

[7] N. Beldiceanu, M. Carlsson, J.-X. Rampon, and C. Truchet. Graph invariants as necessary conditions for global constraints. In P. van Beek, editor, *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP 2005)*, volume 3709 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2005.

[8] C. Bessière, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. The range and roots constraints: Specifying counting and occurrence problems. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 60–65. Professional Book Center, 2005.

[9] C. Bessière, E. Hebrard, B. Hnich, and T. Walsh. Disjoint, partition and intersection constraints for set and multiset variables. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2004.

[10] C. Bessière, E. Hebrard, B. Hnich, and T. Walsh. The tractability of global constraints. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 716–720. Springer, 2004.

[11] R.G. Busacker and P.J. Gowen. A Procedure for Determining a Family of Minimum-Cost Network Flow Patterns. Technical Report ORO-TP-15, Operations Research Office, The Johns Hopkins University, Bethesda, MD, 1960.

[12] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *Euro. J. Oper. Res.*, 78:146–161, 1994.

[13] Y. Caseau, P.-Y. Guillo, and E. Levenez. A Deductive and Object-Oriented Approach to a Complex Scheduling Problem. In *Proceedings of Deductive and Object-Oriented Databases, Third International Conference (DOOD'93)*, pages 67–80, 1993.

[14] V. Chvátal. *Linear programming*. Freeman, 1983.

[15] G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In Tj.C. Koopmans, editor, *Activity Analysis of Production and Allocation – Proceedings of a conference*, pages 339–347. Wiley, 1951.

[16] G. Dooms, Y. Deville, and P. Dupont. CP(Graph): Introducing a Graph Computation Domain in Constraint Programming. In P. van Beek, editor, *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP 2005)*, volume 3709 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2005.

[17] F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP 1999)*, volume 1713 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 1999.

[18] L.R. Ford, Jr and D.R. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations Research*, 6:419–433, 1958.

[19] H.N. Gabow and R.E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of computing (STOC 1983)*, pages 246–251. ACM, 1983.

[20] M.R. Garey and D.S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[21] F. Glover. Maximum matching in convex bipartite graphs. *Naval Research Logistics Quarterly*, 14:313–316, 1967.

[22] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, 10:26–30, 1935.

[23] L. Hellsten, G. Pesant, and P. van Beek. A Domain Consistency Algorithm for the Stretch Constraint. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 290–304. Springer, 2004.

[24] P. Van Hentenryck and J.-P. Carillon. Generality vs. specificity: an experience with AI and OR techniques. In *Proceedings of the National Conference on Articial Intelligence (AAAI)*, pages 660–664, 1988.

[25] D.S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. Brooks / Cole Pub. Co., 1996.

[26] W.-J. van Hoeve. A Hyper-Arc Consistency Algorithm for the Soft Alldifferent Constraint. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 679–689. Springer, 2004.

[27] W.-J. van Hoeve, G. Pesant, and L.-M. Rousseau. On Global Warming: Flow-Based Soft Global Constraints. *Journal of Heuristics*, 2006. To appear.

[28] J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[29] J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[30] M. Iri. A new method of solving transportation-network problems. *Journal of the Operations Research Society of Japan*, 3:27–87, 1960.

[31] W.S. Jewell. Optimal Flows Through Networks. Technical Report 8, Operations Research Center, MIT, Cambridge, MA, 1958.

[32] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing (STOC 1984)*, pages 302–311. ACM, 1984.

[33] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[34] I. Katriel. Expected-case analysis for delayed filtering, 2006.

[35] I. Katriel and S. Thiel. Complete bound consistency for the global cardinality constraint. *Constraints*, 10(3):191–217, 2005.

[36] L.G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[37] W. Kocjan and P. Kreuger. Filtering methods for symmetric cardinality constraint. In J.-C. Régin and M. Rueher, editors, *Proceedings of the First International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combi-*

natorial Optimization Problems (CPAIOR 2004), volume 3011 of *Lecture Notes in Computer Science*, pages 200–208. Springer, 2004.

[38] D. König. Graphok és matrixok. *Matematikai és Fizikai Lapok*, 38:116–119, 1931.

[39] J.-L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[40] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, editors. *The Traveling Salesman Problem – A Guided Tour of Combinatorial Optimization*. Wiley, 1985.

[41] M. Leconte. A bounds-based reduction scheme for constraints of difference. In *Proceedings of the Second International Workshop on Constraint-based Reasoning (Constraint 1996)*, pages 19–28, 1996.

[42] W. Lipski and F.P. Preparata. Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Informatica*, 15:329–346, 1981.

[43] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 245–250. Morgan Kaufmann, 2003.

[44] K. Mehlhorn and S. Thiel. Faster Algorithms for Bound-Consistency of the Sortedness and the Alldifferent Constraint. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP 2000)*, volume 1894 of *Lecture Notes in Computer Science*, pages 306–319. Springer, 2000.

[45] L. Mercier and P. Van Hentenryck. Edge finding for cumulative scheduling, 2005.

[46] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, 1988.

[47] A. Oplobedu, J. Marcovitch, and Y. Tourbier. CHARME: Un langage industriel de programmation par contraintes, illustré par une application chez Renault. In *Proceedings of the Ninth International Workshop on Expert Systems and their Applications: General Conference*, volume 1, pages 55–70, 1989.

[48] G. Pesant. A Filtering Algorithm for the Stretch Constraint. In T. Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP 2001)*, volume 2239 of *Lecture Notes in Computer Science*, pages 183–195. Springer, 2001.

[49] G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer, 2004.

[50] J. Petersen. Die Theorie der regulären graphs. *Acta Mathematica*, 15:193–220, 1891.

[51] T. Petit, J.-C. Régin, and C. Bessière. Specific Filtering Algorithms for Over-Constrained Problems. In T. Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP 2001)*, volume 2239 of *Lecture Notes in Computer Science*, pages 451–463. Springer, 2001.

[52] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI / IAAI)*, pages 359–366. AAAI Press / The MIT Press, 1998.

[53] C.-G. Quimper, A. López-Ortiz, P. van Beek, and A. Golynski. Improved Algorithms

for the Global Cardinality Constraint. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 542–556. Springer, 2004.

[54] C.-G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski, and S.B. Sadjad. An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint. *Constraints*, 10(2):115–135, 2005.

[55] P. Refalo. Linear Formulation of Constraint Programming Models and Hybrid Solvers. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP 2000)*, volume 1894 of *Lecture Notes in Computer Science*, pages 369–383. Springer, 2000.

[56] J.-C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI)*, volume 1, pages 362–367. AAAI Press, 1994.

[57] J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference (AAAI / IAAI)*, volume 1, pages 209–215. AAAI Press / The MIT Press, 1996.

[58] J.-C. Régin. Arc Consistency for Global Cardinality Constraints with Costs. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP 1999)*, volume 1713 of *Lecture Notes in Computer Science*, pages 390–404. Springer, 1999.

[59] J.-C. Régin. Cost-Based Arc Consistency for Global Cardinality Constraints. *Constraints*, 7:387–405, 2002.

[60] J.-C. Régin, T. Petit, C. Bessière, and J.-F. Puget. An Original Constraint Based Approach for Solving over Constrained Problems. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP 2000)*, volume 1894 of *Lecture Notes in Computer Science*, pages 543–548. Springer, 2000.

[61] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[62] A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency*. Springer, 2003.

[63] M. Sellmann. Approximated consistency for knapsack constraints. In F. Rossi, editor, *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *Lecture Notes in Computer Science*, pages 679–693. Springer, 2003.

[64] M. Sellmann. Cost-based filtering for shorter path constraints. In F. Rossi, editor, *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *Lecture Notes in Computer Science*, pages 694–708. Springer, 2003.

[65] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[66] M.A. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Annals of Operations Research*, 118:73–84, 2003.

[67] B.L. van der Waerden. Ein Satz über Klasseneinteilungen von endlichen Mengen. *Abhandlungen aus dem mathematischen Seminar der Hamburgischen Universität*, 5:185–188, 1927.

[68] V. Vazirani. *Approximation Algorithms*. Springer, 2001.