The Alldifferent Constraint: A Survey*

Willem-Jan van Hoeve VANHOEVE@CS.CORNELL.EDU
Cornell University, Department of Computer Science
4130 Upson Hall, Ithaca, NY 14853, USA
`http://www.cs.cornell.edu/~vanhoeve`

Abstract

Constraints of difference are known to the constraint programming community since Lauriere introduced ALICE in 1978. Since then, several strategies have been designed to handle the `alldifferent` constraint. This paper surveys the most important developments over the years regarding this important constraint. First we introduce the underlying concepts and results from combinatorial theory. Then we give an overview and a comparison of different solution strategies achieving arc, range, bounds and hyper-arc consistency, respectively. In addition, three variants of the `alldifferent` constraint are discussed: the symmetric `alldifferent` constraint, the minimum weight `alldifferent` constraint and the soft `alldifferent` constraint. Finally, the convex hull representation of the `alldifferent` constraint in integer linear programming is considered.

*. A preliminary version of this paper appeared as (van Hoeve, 2001).

Contents

1 Introduction **3**
- 1.1 The Alldifferent Constraint in Constraint Programming . . . 3
- 1.2 Historical Overview . . . 4
- 1.3 Outline of the Paper . . . 5

2 Preliminaries **6**
- 2.1 Constraint Programming . . . 6
  - 2.1.1 Basic Notions . . . 6
  - 2.1.2 Solution Process . . . 7
  - 2.1.3 Local Consistency Notions . . . 7
- 2.2 Graph Theory . . . 9
  - 2.2.1 Basic Notions . . . 9
  - 2.2.2 Matching Theory . . . 10
  - 2.2.3 Flow Theory . . . 11

3 Combinatorial Background **13**
- 3.1 Alldifferent and Bipartite Matching . . . 13
- 3.2 Hall's Marriage Theorem . . . 14

4 Filtering Algorithms for Local Consistency Notions **16**
- 4.1 Local Consistency of a Decomposed CSP . . . 16
- 4.2 Bounds Consistency . . . 18
- 4.3 Range Consistency . . . 20
- 4.4 Hyper-arc Consistency . . . 22
- 4.5 Complexity Survey and Discussion . . . 25

5 Variants of the Alldifferent Constraint **26**
- 5.1 The Symmetric Alldifferent Constraint . . . 26
- 5.2 The Weighted Alldifferent Constraint . . . 29
- 5.3 The Soft Alldifferent Constraint . . . 32
  - 5.3.1 Variable-Based Violation Measure . . . 34
  - 5.3.2 Decomposition-Based Violation Measure . . . 34

6 The Alldifferent Polytope **36**

References **38**

1. Introduction

Many combinatorial (optimization) problems can be modeled and solved using the *constraint programming* paradigm (Apt, 2003; Dechter, 2003). In constraint programming, a model is stated by means of *variables* that range over their *domain* of possible values, and *constraints* on these variables. A constraint restricts the space of possible variable instantiations. For example, if x and y are variables which both range over the domain $\{1, 2, 3\}$, the constraint $x + y \leq 4$ forbids the instantiations $(x, y) = (3, 2), (2, 3), (3, 3)$. The *not-equal* constraint $x \neq y$ forbids the instantiations $(x, y) = (1, 1), (2, 2), (3, 3)$.

A constraint programming system consists of several constraint types, each of which may be treated differently by the system. One of the constraints that is present in practically all constraint programming systems, is the famous `alldifferent` constraint. It states that all variables in this constraint must be pairwise different.

In this section we first introduce the `alldifferent` constraint within the context of constraint programming. Then a brief historical overview is presented. This section concludes with an outline of the paper.

1.1 The Alldifferent Constraint in Constraint Programming

We first give a formal definition of the `alldifferent` constraint. Then we model the well-known n-queens problem using the `alldifferent` constraint in Example 1.

Definition 1 (Pairwise difference) *Let $x_1, x_2, \ldots, x_n$ be variables with respective finite domains $D(x_1), D(x_2), \ldots, D(x_n)$. Then*

$$\texttt{alldifferent}(x_1, \ldots, x_n) = \{(d_1, \ldots, d_n) \mid d_i \in D(x_i), d_i \neq d_j \text{ for } i \neq j\}.$$

Example 1 *We want to model the n-queens problem, described as: "Place n queens on an $n \times n$ chessboard in such a way that no queen attacks another queen."*

One way of modelling this problem is to introduce an integer variable x_i for every row $i = 1, 2, \ldots, n$, which ranges over column 1 to n. This means that in every row i a queen is placed in the x_i-th column. We express the no-attack constraints as

$$x_i \neq x_j \quad \text{for } 1 \leq i < j \leq n, \tag{1}$$
$$x_i - x_j \neq i - j \quad \text{for } 1 \leq i < j \leq n, \tag{2}$$
$$x_i - x_j \neq j - i \quad \text{for } 1 \leq i < j \leq n, \tag{3}$$
$$x_i \in \{1, 2, \ldots, n\} \quad \text{for } 1 \leq i \leq n. \tag{4}$$

The constraints (1) state that no two queens are allowed to occur in the same column. The constraints (2) and (3) state the diagonal cases. After rearranging the terms of constraints (2) and (3), we transform the model into

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

(Note that it is shown in (Falkowski & Schmitz, 1986) how to construct a solution to the n-Queens problem for $n > 3$.)

We can find a solution to a model as in the previous example by systematically enumerating all possible variable-value combinations. For each combination we check whether it satisfies all constraints. If this is the case, we have found a solution. Otherwise we continue. This is roughly how a constraint programming system computes a solution. Unfortunately, the systematic enumeration of variable-value combinations leads to a search space of exponential size. As a result, it may take a very long time before we find a solution.

To overcome this problem (at least partially), techniques have been developed to safely prune parts of the search space beforehand. To each constraint of the model, an individual "solver" is attached. It identifies and removes domain values that never appear in any solution to the constraint in the remaining search space. Doing so, the number of combinations is reduced and the search space becomes smaller.

Some of the variables of which the domain has been reduced may also appear in other constraints. In reaction to the domain reduction, these constraints ask their respective solvers to identify and remove inconsistent values again. This process is called *constraint propagation*, because the effect of domain reduction is propagated through the constraints. The solvers for the individual constraints are called *constraint propagation algorithms*, while the domain reduction algorithms are also called *domain filtering algorithms*. If at a certain stage in the search process no more inconsistent values are detected, the constraint propagation algorithms are said have reached some (specified) notion of *local consistency*.

In this paper we focus on filtering algorithms for the `alldifferent` constraint. The `alldifferent` constraint gives rise to several filtering algorithms, depending on the desired notion of local consistency. There exist different notions of local consistency, each notion allowing more or less values in a variable domain. In general it takes more time to obtain a "stronger" local consistency than to obtain a "weaker" local consistency. So with more effort, one could remove more domain values. Naturally, the goal is to design efficient filtering algorithms that establish the strongest possible notion of local consistency. Nevertheless, for each individual problem one has to make a trade-off between the effort (time) and the gain (search space reduction) when choosing a particular notion of local consistency to establish.

1.2 Historical Overview

In 1978 Lauriere introduced ALICE, *"A language and a program for stating and solving combinatorial problems"* (Lauriere, 1978). Already in this system the importance of disequality constraints was recognized. The keyword *"DIS"* applied to a set of variables is used to state that the variables must take different values. It defines a global structure (i.e. the set of variables form a *"clique of disjunctions"*), that is exploited during the search for a solution.

After the introduction of constraints in logic programming, for example in the system CHIP (Dincbas, Van Hentenryck, Simonis, Aggoun, Graf, & Berthier, 1988), it was also possible to express the constraint of difference as the well-known `alldifferent` constraint. In the system ECL^{i}PS^e (Wallace, Novello, & Schimpf, 1997) this constraint was introduced as `alldistinct`. However, in the early constraint (logic) programming systems this constraint was treated internally as a sequence of not-equal constraints; see for example (Van

Hentenryck, 1989). Unfortunately the global information is lost in that way. The global view was retrieved with the algorithm introduced by (Régin, 1994), that considers all not-equal constraints simultaneously.

Throughout the history of constraint programming, the `alldifferent` constraint has played a special role. Various papers and books make use of this special constraint to show the benefits of constraint programming, either to show its modelling power, or to show that problems can be solved faster using this constraint. From a modelling point of view, the `alldifferent` constraint arises naturally in problems that are based upon a permutation or when a directed graph has to be covered with disjoint circuits. Numerous applications exist in which the `alldifferent` constraint is of vital importance, for example quasi-group completion problems (Gomes & Shmoys, 2002), air traffic management (Barnier & Brisset, 2002; Grönkvist, 2004) and rostering problems (Tsang, Ford, Mills, Bradwell, Williams, & Scott, 2004). Finally, many other global constraints can be viewed as an extension of the `alldifferent` constraint, for example the `sort` constraint (Older, Swinkels, & van Emden, 1995; Zhou, 1997), the `cycle` constraint (Beldiceanu & Contejean, 1994), the `diffn` constraint (Beldiceanu & Contejean, 1994) and the global cardinality constraint (Régin, 1996).

Over the years, the `alldifferent` constraint as well as other global constraints has been well-studied in constraint programming; see for example (Beldiceanu, Carlsson, & Rampon, 2005) and (Régin, 2003) for an overview. Special algorithms have been developed that are able to exploit the global information of the constraints. As we will see, for the `alldifferent` constraint at least six different filtering algorithms exist, each achieving a different notion of local consistency, or achieving it faster. These algorithms often make use of the same underlying principles. To make them more understandable, accessible and coherent, this paper presents a systematic overview of the `alldifferent` constraint, which may probably be regarded as the most well-known, most influential and most studied constraint in the field of constraint programming.

1.3 Outline of the Paper

This paper is organized as follows. In Section 2 we present preliminaries on constraint programming and graph theory. In particular, we formally define different notions of local consistency that are applied to the `alldifferent` constraint. The preliminaries on graph theory include results from matching theory and flow theory.

In Section 3 we connect results from combinatorial theory to the `alldifferent` constraint. Many of the considered filtering algorithms rely on these results.

Each of the Sections 4.1 up to 4.4 treats a different notion of local consistency. The sections are ordered in increasing strength of the considered local consistency. The treatment consists of a description of the particular notion of local consistency with respect to the `alldifferent` constraint, together with a description of an algorithm that establishes that particular local consistency notion. Section 4 ends with a time complexity survey and a discussion.

Section 5 gathers a number of variants of the `alldifferent` constraint. First the symmetric version of the `alldifferent` constraint is considered. Then the weighted `alldiff-`

erent constraint is presented, where a linear objective function in conjunction with the alldifferent constraint is exploited. Third, the soft alldifferent constraint is treated, which allows partial violation of the alldifferent constraint.

Finally, Section 6 considers the alldifferent polytope, which is a particular description of the solution set of the alldifferent constraint, using linear constraints. This description can be applied in integer linear programming models.

2. Preliminaries

2.1 Constraint Programming

In this section we recall constraint programming concepts. For more information on constraint programming we refer to (Apt, 2003) and (Dechter, 2003).

2.1.1 Basic Notions

Let x be a variable. The *domain* of x, denoted by $D(x)$, is a set of values that can be assigned to x. As a shorthand to define $D(x) = \{d_1, \ldots, d_m\}$ we often write $x \in \{d_1, \ldots, d_m\}$. In this paper we only consider variables with *finite* domains.

Let $\mathcal{Y} = y_1, y_2, \ldots, y_k$ be a finite sequence of variables, where $k > 0$. A *constraint* C on $\mathcal{Y}$ is defined as a subset of the Cartesian product of the domains of the variables in $\mathcal{Y}$, i.e. $C \subseteq D(y_1) \times D(y_2) \times \cdots \times D(y_k)$. This is written as $C(\mathcal{Y})$ or $C(y_1, y_2, \ldots, y_k)$.[1] A constraint is called a *binary constraint* if it is defined on two variables. A *global constraint* is a constraint that is defined on more than two variables.

A *constraint satisfaction problem*, or a *CSP*, is defined by a finite sequence of variables $\mathcal{X} = x_1, x_2, \ldots, x_n$ with respective domains $\mathcal{D} = D(x_1), D(x_2), \ldots, D(x_n)$, together with a finite set of constraints $\mathcal{C}$, each on a subsequence of $\mathcal{X}$. To simplify notation, we often omit braces "{ }" when presenting a specific set of constraints. A CSP P is also denoted by $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$.

Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CSP where $\mathcal{X} = x_1, x_2, \ldots, x_n$ and $\mathcal{D} = D(x_1), D(x_2), \ldots, D(x_n)$. A tuple $(d_1, \ldots, d_n) \in D(x_1) \times \cdots \times D(x_n)$ *satisfies* a constraint $C \in \mathcal{C}$ on the variables $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$ if $(d_{i_1}, d_{i_2}, \ldots, d_{i_m}) \in C$. If no such tuple satisfies C, we say that C is *inconsistent*. A tuple $(d_1, \ldots, d_n) \in D(x_1) \times \cdots \times D(x_n)$ is a *solution* to a CSP if it satisfies every constraint $C \in \mathcal{C}$.

A *consistent* CSP is a CSP for which a solution exists. An *inconsistent* CSP is a CSP for which no solution exists. A *failed* CSP is a CSP with an empty domain or with only singleton domains that together are not a solution to the CSP. A *solved* CSP is a CSP with only singleton domains that together are a solution to the CSP. Note that a failed CSP is also inconsistent, but not all inconsistent CSPs are failed.

Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and $P' = (\mathcal{X}, \mathcal{D}', \mathcal{C}')$ be CSPs. P and P' are called *equivalent* if they have the same solution set. P is said to be *smaller* than P' if they are equivalent and $D(x) \subseteq D'(x)$ for all $x \in \mathcal{X}$. This relation is written as $P \preceq P'$. P is *strictly smaller* than P', if $P \preceq P'$ and $D(x) \subset D\prime(x)$ for at least one $x \in \mathcal{X}$. This is written as $P \prec P'$. When

1. Sometimes a constraint C is defined on variables $\mathcal{Y}$ together with additional parameters p, for example a set of cost values. In such cases we write $C(\mathcal{Y}, p)$ or $C(y_1, y_2, \ldots, y_k, p)$, for syntactical convenience.

both $P \preceq P'$ and $P' \preceq P$ we write $P \equiv P'$.

We fix the following notation. For a sequence of variables K we denote $D(K) = \bigcup_{x \in K} D(x)$. When the domain $D(x)$ of a variable x is a singleton, say $D(x) = \{d\}$, we also write $x = d$.

2.1.2 Solution Process

In constraint programming, the goal is to find a solution (or all solutions) to a given CSP. The solution process interleaves *domain filtering* and *constraint propagation*, and *search*.

The search for a solution (or all solutions) to a CSP is performed by iteratively splitting a CSP into smaller CSPs. The splitting process is said to construct a *search tree*. A node of the tree represents a CSP. Initially, at the root, we are given a CSP P_0 that is the problem we want to solve. If P_0 is not solved nor failed, we split P_0 into two or more CSPs $P_1, P_2, \ldots, P_k$ $(k > 1)$. We must ensure however that all solutions to P_0 are preserved and no solutions are added to P_0. Thus, the union of the solution sets of $P_1, P_2, \ldots, P_k$ equals the solution set of P_0. Further, each CSP P_i $(i > 0)$ should be strictly smaller than P_0 to ensure that the splitting process terminates. Next we split each of the CSPs $P_1, P_2, \ldots, P_k$ according to the same criteria as above. The splitting proceeds until we have split all CSPs into either failed or solved CSPs. The failed and solved CSPs are the leaves of the search tree.

The size of the search tree is exponential in the number of variables of P_0. To reduce the size of the search tree, constraint programming uses a process called *constraint propagation*. Given a constraint C and a notion of local consistency, a *domain filtering algorithm* removes values that are not consistent with C from the domains of the variables in C. The algorithm must preserve all solutions and not add any solution to C. Whenever an inconsistent domain value has been removed, the effect is *propagated* through all other constraints that share the same corresponding variable. This iterative process continues until no more inconsistent domain values are detected for all constraints in a CSP. We then say that the CSP is *locally consistent* and that we have *established* the notion of local consistency on the CSP. The term "local consistency" reflects that we do not obtain a globally consistent CSP, but a CSP in which all constraints are locally, i.e. individually, consistent. A thorough description of the process of constraint propagation is given in (Apt, 1999); see also (Apt, 2003).

After splitting a CSP, domain filtering and constraint propagation is applied to the smaller CSPs. The removal of domain values leads to a smaller search tree and thus speeds up the solution process. However, the time spent on constraint propagation should be less than the speed-up it induces in order to be effective. Therefore we want to apply efficient filtering algorithms. The efficiency is often determined by the notions of local consistency that are applied to the constraints. It remains problem dependent when to apply which notion of local consistency during the solution process.

2.1.3 Local Consistency Notions

Next we recall four notions of local consistency of a constraint.

Definition 2 (Arc consistency) *A binary constraint $C(x_1, x_2)$ is* arc consistent *if for all values $d_1 \in D(x_1)$ there exists a value $d_2 \in D(x_2)$ such that $(d_1, d_2) \in C$, and for all values $d_2 \in D(x_2)$ there exists a value $d_1 \in D(x_1)$ such that $(d_1, d_2) \in C$.*

Definition 3 (Hyper-arc consistency) *A constraint $C(x_1, \ldots, x_m)$ $(m > 1)$ is* hyper-arc consistent *if for all $i \in \{1, \ldots, m\}$ and all values $d_i \in D(x_i)$, there exist values $d_j \in D(x_j)$ for all $j \in \{1, \ldots, m\} - i$ such that $(d_1, \ldots, d_m) \in C$.*

Note that arc consistency is equal to hyper-arc consistency applied to binary constraints. Both arc consistency and hyper-arc consistency ensure that all values in every domain belong to a tuple that satisfies the constraint, with respect to the current variable domains. In the literature hyper-arc consistency is also referred to as *generalized arc-consistency* and *domain consistency*.

The two following local consistency notions reason on the *bounds* of the variable domains. For that reason, we assume that the involved variable domains are subsets of a fixed, linearly ordered, finite set of domain elements when we apply these definitions. Let D be such a domain. We define $\min D$ and $\max D$ to be its minimum value and its maximum value, respectively. Further, we use braces "{ }" and brackets "[ ]" to indicate a set and an interval of domain values, respectively. Thus, the set $\{1, 3\}$ contains the values 1 and 3 whereas the interval $[1, 3] \subset \mathbb{N}$ contains 1, 2 and 3.

Definition 4 (Bounds consistency) *A constraint $C(x_1, \ldots, x_m)$ $(m > 1)$ is* bounds consistent *if for all $i \in \{1, \ldots, m\}$ and each value $d_i \in \{\min D(x_i), \max D(x_i)\}$, there exist values $d_j \in [\min D(x_j), \max D(x_j)]$ for all $j \in \{1, \ldots, m\} - i$ such that $(d_1, \ldots, d_m) \in C$.*

Definition 5 (Range consistency) *A constraint $C(x_1, \ldots, x_m)$ $(m > 1)$ is* range consistent *if for all $i \in \{1, \ldots, m\}$ and all values $d_i \in D(x_i)$, there exist values $d_j \in [\min D(x_j), \max D(x_j)]$ for all $j \in \{1, \ldots, m\} - i$ such that $(d_1, \ldots, d_m) \in C$.*

Range consistency does not ensure the feasibility of the constraint with respect to the domains, but with respect to intervals that include the domains. It can be regarded as a relaxation of hyper-arc consistency. Bounds consistency can in turn be regarded as a relaxation of range consistency. It does not even consider all values in the domains, but only the minimum and the maximum value. For these values the feasibility of the constraint is again ensured with respect to intervals that include the domains. Consequently, range consistency and bounds consistency do not guarantee the existence of a solution to the constraint, which hyper-arc consistency does. This is formalized in Theorem 1.

Definition 6 (Locally consistent CSP) *A CSP is arc consistent, respectively range consistent, bounds consistent or hyper-arc consistent if all its constraints are.*

If we apply to a CSP P a propagation algorithm that establishes range consistency on P, we denote the result by $\Phi_{\mathrm{R}}(P)$. Analogously, $\Phi_{\mathrm{B}}(P)$, $\Phi_{\mathrm{A}}(P)$ and $\Phi_{\mathrm{HA}}(P)$ denote the application of bounds consistency, arc consistency and hyper-arc consistency on P, respectively.

Theorem 1 *Let P be a CSP. Then $\Phi_{\mathrm{HA}}(P) \preceq \Phi_{\mathrm{R}}(P) \preceq \Phi_{\mathrm{B}}(P)$ and all relations may be strict.*

Proof. If a constraint is hyper-arc consistent then it is also range consistent, since $D(x) \subseteq [\min D(x), \max D(x)]$ for all variables x in P. The converse is not true, see Example 2. Hence $\Phi_{\mathrm{HA}}(P) \preceq \Phi_{\mathrm{R}}(P)$, possibly strict.

If a constraint is range consistent then it is also bounds consistent, because $\{\min D(x), \max D(x)\} \subseteq D(x)$ for all variables x in P. The converse is not true, see Example 2. Hence $\Phi_{\mathrm{R}}(P) \preceq \Phi_{\mathrm{B}}(P)$, possibly strict. □

Example 2 *Consider the following CSP*

$$P = \begin{cases} x_1 \in \{1,3\}, x_2 \in \{2\}, x_3 \in \{1,2,3\}, \\ \texttt{alldifferent}(x_1, x_2, x_3). \end{cases}$$

Then $\Phi_{\mathrm{B}}(P) \equiv P$*, while*

$$\Phi_{\mathrm{R}}(P) = \begin{cases} x_1 \in \{1,3\}, x_2 \in \{2\}, x_3 \in \{1,3\}, \\ \texttt{alldifferent}(x_1, x_2, x_3) \end{cases}$$

and $\Phi_{\mathrm{HA}}(P) \equiv \Phi_{\mathrm{R}}(P)$*. Next, consider the CSP*

$$P' = \begin{cases} x_1 \in \{1,3\}, x_2 \in \{1,3\}, x_3 \in \{1,3\}, \\ \texttt{alldifferent}(x_1, x_2, x_3). \end{cases}$$

This CSP is obviously inconsistent, since there are only two values available, namely 1 and 3, for three variables that must be pairwise different. Indeed, $\Phi_{\mathrm{HA}}(P')$ *is a failed CSP, while* $\Phi_{\mathrm{R}}(P') \equiv P'$*.*

2.2 Graph Theory

We use results from matching theory and flow theory without presenting all underlying algorithms and proofs in detail. This is because many of the results are well-known and there already exist excellent overviews of these topics. More information about matching theory can for example be found in (Lovász & Plummer, 1986), (Gerards, 1995) and (Schrijver, 2003, Chapter 16–38). More information about flow theory can be found in (Ahuja, Magnanti, & Orlin, 1993) and (Schrijver, 2003, Chapter 6–15). However, we do present proofs and algorithms that provide useful insights.

In this paper, we mainly follow the notation of (Schrijver, 2003), unless this conflicts with notation from Section 2 on constraint programming.

2.2.1 Basic Notions

A *graph* or *undirected graph* is a pair $G = (V, E)$, where V is a finite set of vertices and E is a multiset[2] of *unordered* pairs from V, called *edges*. An edge "between" $u \in V$ and $v \in V$ is denoted by uv.

A graph $G = (V, E)$ is *bipartite* if there exists a partition S, T of V such that $E \subseteq \{st \mid s \in S, t \in T\}$. We also write $G = (S, T, E)$.

2. A multiset is a set in which elements may occur more than once. In the literature it is also referred to as *family*.

A *walk* in a graph $G = (V, E)$ is a sequence $P = v_0, e_1, v_1, \ldots, e_k, v_k$ where $k \geq 0$, $v_0, v_1, \ldots, v_k \in V$, $e_1, e_2, \ldots, e_k \in E$ and $e_i = v_{i-1}v_i$ for $i = 1, \ldots, k$. 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 walk, i.e. $v_0 = v_k$, is called a *circuit* if $v_1, \ldots, v_k$ are distinct.

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

A *directed graph* is a pair $G = (V, A)$ where V is a finite set of vertices and A is a multiset of *ordered* pairs from V, 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).

Similarly to undirected bipartite graphs, a directed graph $G = (V, A)$ is *bipartite* if there exists a partition S, T of V such that $A \subseteq \{(s, t) \mid s \in S, t \in T\}$. We also 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 $k \geq 0$, $v_0, v_1, \ldots, v_k \in V$, $a_1, a_2, \ldots, a_k \in A$ and $a_i = (v_{i-1}, v_i)$ for $i = 1, \ldots, k$. 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 walk, i.e. $v_0 = v_k$, is called a *directed circuit* if $v_1, \ldots, v_k$ are distinct.

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

2.2.2 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 set $S \subseteq V$ if all vertices in S belong to an edge in M. A vertex $v \in V$ is called *M-free* if M does not cover v. The *size* of a matching M is $|M|$. The *maximum matching problem* is the problem of finding a matching of maximum size 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.

On an M-augmenting path, we can exchange edges in M and not in M, to obtain a matching M' with $|M'| = |M| + 1$. The following result is due to (Petersen, 1891)[3].

Theorem 2 *Let $G = (V, E)$ be a graph, and let M be a matching in M. Then either M is a maximum-size matching, or there exists an M-augmenting path.*

Proof. If M is a maximum-size matching, then there exists no M-augmenting path, because otherwise exchangement of edges on this path gives a larger matching.

If M' is a matching larger than M, consider the graph $G' = (V, M \cup M')$. In G', each vertex is connected to at most two edges. Hence, each component of G' is either a circuit or

3. In the literature this result is often ascribed to (Berge, 1957). However, it should actually be attributed to Petersen, as for example pointed out by (Mulder, 1992).

a path (possibly of length zero). As $|M'| > |M|$ there is at least one component containing more edges of M' than of M. Because all circuits contain an even number of edges, this component must be an M-augmenting path. □

Hence, a maximum-size matching can be found by iteratively computing M-augmenting paths in G and extending M. This method is due to (van der Waerden, 1927) and (König, 1931). It can be done as follows.

Let $G = (U, W, E)$ be a bipartite graph, and let M be a matching in G. 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.

$$A = \{(w, u) \mid uw \in M, u \in U, w \in W\} \cup \{(u, w) \mid uw \in E \setminus M, u \in U, w \in W\}.$$

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 $O(|U|\,|A|)$.

We can improve the algorithm to run in $O(|V|^{1/2}\,|A|)$ time, as shown by (Hopcroft & Karp, 1973), where $V = U \cup W$. 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. One can show that after $|V|^{1/2}$ iterations, there are at most $O(|V|^{1/2})$ more iterations possible, by reasoning on the length of the paths. This leads to a total time complexity of $O(|V|^{1/2}\,|A|)$.

In a general graph $G = (V, E)$ (not necessarily bipartite), a maximum-size matching can be computed in $O(|V|\,|E|)$ time (Edmonds, 1965) [4] or even $O(|V|^{1/2}\,|E|)$ time (Micali & Vazirani, 1980).

2.2.3 Flow Theory

Let $G = (V, A)$ be a directed graph. For $v \in V$, let $\delta^{\text{in}}(v)$ and $\delta^{\text{out}}(v)$ denote the multiset of arcs entering and leaving v, respectively. Let $s, t \in V$ denote the "source" and the "sink" respectively. A function $f : A \to \mathbb{R}$ is called a *flow from s to t*, or an *$s - t$ flow*, if

$$\begin{array}{lll} (i) & f(a) \geq 0 & \text{for each } a \in A, \\ (ii) & f(\delta^{\text{out}}(v)) = f(\delta^{\text{in}}(v)) & \text{for each } v \in V \setminus \{s, t\}. \end{array} \tag{5}$$

Here $f(S) = \sum_{a \in S} f(a)$ for all $S \subseteq A$. Property (5)(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 as

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

In other words, the value of a flow is the net amount of flow leaving s. This is equal to the net amount of flow entering t.

4. this complexity is not due to edmonds

Let $c : A \to \mathbb{Q}_+$ be a "capacity" function. We say that a flow f is *under* c if

$$f(a) \leq c(a) \text{ for each } a \in A.$$

A *maximum* $s-t$ *flow* is an $s-t$ flow under c of maximum value.

Let $w : A \to \mathbb{R}$ be a "weight" function. 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 function $f : A \to \mathbb{R}$ is defined as

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

A *minimum-weight* $s-t$ *flow* is an $s-t$ flow under c of maximum value with minimum weight. Hence, such a flow has minimum weight among all flows of maximum value.

Let f be an $s-t$ flow under c in G. The *residual graph* of f (with respect to c) is defined as $G_f = (V, A_f)$ where

$$A_f = \{a \mid a \in A, f(a) < c(a)\} \cup \{a^{-1} \mid a \in A, f(a) > 0\}.$$

Here $a^{-1} = (v, u)$ if $a = (u, v)$. We extend w to $A^{-1} = \{a^{-1} \mid a \in A\}$ by defining

$$w(a^{-1}) = -w(a)$$

for each $a \in A$.

In order to compute a minimum-weight $s-t$ flow we use the following notation. Any directed path P in G_f gives an undirected path in $G = (V, A)$. We define $\chi^P \in \mathbb{R}^A$ by

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

for $a \in A$. We define $\chi^C \in \mathbb{R}^A$ similarly for a directed circuit C in G_f.

Using the above notation, a minimum-weight $s-t$ flow in G can be found using the following algorithm; see (Schrijver, 2003, p. 185):

> **Algorithm for minimum-weight $s-t$ flow**
> Starting with $f = \mathbf{0}$ apply the following iteratively:
> *Iteration:* Let P be a directed $s-t$ path in G_f minimizing $w(P)$. Reset $f = f + \varepsilon\chi^P$, where ε is maximal subject to $\mathbf{0} \leq f + \varepsilon\chi^P \leq c$.

The algorithm stops when no more directed $s-t$ paths in G_f can be found or when a predefined maximum value ϕ is attained. For integer capacity function c and weight function w the algorithm finds an integer flow.

The time complexity of this algorithm (for finding a minimum-weight flow of value ϕ) is $O(\phi \cdot \text{SP})$, where SP is the time to compute a shortest directed path in G. Although faster algorithms exist for general minimum-cost flow problems, this algorithm suffices when applied to our problems. This is because in our case the value of all flows is bounded by the number of variables inside the `alldifferent` constraint.

Note that the algorithm for finding a maximum-size 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 2.2.2, 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' mimicks exactly the algorithm described in Section 2.2.2 for finding a maximum-size matching in G. Namely, 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 consider a result that is particularly useful for designing incremental filtering algorithms, as we will see. Given a minimum-weight $s-t$ flow, we want to compute the additional weight when an unused arc is forced to be used.

Theorem 3 *Let f be a minimum-weight $s-t$ flow 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 maximal subject to $\mathbf{0} \leq f + \varepsilon\chi^C \leq c$, is a maximum $s-t$ flow in G with minimum weight among all maximal $s-t$ flows g in G with $g(a) = \varepsilon$. If C does not exist, f' does not exist. Otherwise,* $\text{weight}(f') = \text{weight}(f) + \varepsilon \cdot w(C)$.

The proof of Theorem 3 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.

3. Combinatorial Background

3.1 Alldifferent and Bipartite Matching

This section shows the equivalence of a solution to the `alldifferent` constraint and a matching in a bipartite graph.

Definition 7 (Value graph) *Let X be a sequence of variables. The bipartite graph $G = (X, D(X), E)$ with $E = \{xd \mid d \in D(x), x \in X\}$ is called the* value graph *of X.*

Theorem 4 *Let $X = x_1, x_2, \ldots, x_n$ be a sequence of variables and let G be the value graph of X. Then $(d_1, \ldots, d_n) \in \texttt{alldifferent}(x_1, \ldots, x_n)$ if and only if $M = \{x_1d_1, \ldots, x_nd_n\}$ is a matching in G.*

Proof. An edge x_id_i (for some $i \in \{1, \ldots, n\}$) in M corresponds to the assignment $x_i = d_i$. As no edges in M share a vertex, $x_i \neq x_j$ for all $i \neq j$. □

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

Example 3 *We want to assign four tasks (1, 2, 3 and 4) to five machines (*A, B, C, D *and* E*). To each machine at most one task can be assigned. However, not every task can be*

Task	Machines
1	B, C, D, E
2	B, C
3	A, B, C, D
4	B, C

Table 1: Possible task - machine combinations.

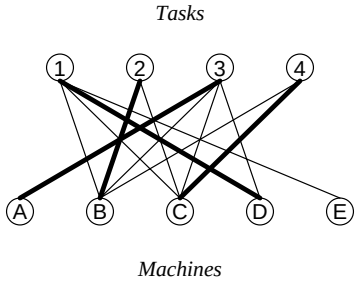


Figure 1: The value graph for the task assignment problem of Example 3. Bold edges form a matching covering all tasks.

assigned to every machine. Table 1 below presents the possible combinations. For example, task 2 can be assigned to machines B *and* C*.*

This problem is modelled as follows. We introduce a variable x_i *for task* $i = 1, \ldots, 4$*, whose value represents the machine to which task* i *is assigned. The initial domains of the variables are defined by the possible combinations in Table 1. Since the tasks have to be assigned to different machines, we introduce an* `alldifferent` *constraint. The problem is thus modelled as the CSP*

$$x_1 \in \{\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}, x_2 \in \{\mathsf{B}, \mathsf{C}\}, x_3 \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}, x_4 \in \{\mathsf{B}, \mathsf{C}\},$$
$$\texttt{alldifferent}(x_1, x_2, x_3, x_4).$$

The value graph of $X = x_1, \ldots, x_n$ *is presented in Figure 1. The bold edges in the value graph denote a matching covering* X*. It corresponds to a solution to the CSP, i.e.* $x_1 = \mathsf{D}, x_2 = \mathsf{B}, x_3 = \mathsf{A}$ *and* $x_4 = \mathsf{C}$*.*

3.2 Hall's Marriage Theorem

A useful theorem to derive filtering algorithms for the `alldifferent` constraint is Hall's *Marriage Theorem*[5] (Hall, 1935):

> If a group of men and women marry only if they have been introduced to each other previously, then a complete set of marriages is possible if and only if every

5. As noted by (Schrijver, 2003, p. 392), the name 'marriage theorem' was introduced by H. Weyl in 1949.

subset of men has collectively been introduced to at least as many women, and vice versa[6].

The following formulation is stated in terms of the `alldifferent` constraint.

Theorem 5 *The constraint* $\texttt{alldifferent}(x_1, \ldots, x_n)$ *has a solution if and only if*

$$|K| \leq |D(K)| \tag{6}$$

for all $K \subseteq \{x_1, \ldots, x_n\}$.

Proof. The direct proof presented here is adapted from (Schrijver, 2003, p. 379), and originally due to (Easterfield, 1946). Call a set K *tight* if equality holds in (6). Necessity of the condition being obvious, we prove sufficiency. We use induction, with the hypothesis that Theorem 5 holds for k variables with $k < n$.

If there is a $d \in D(x_n)$ such that

$$\begin{array}{l} x_1 \in D(x_1) \setminus \{d\}, \ldots, x_{n-1} \in D(x_{n-1}) \setminus \{d\}, \\ \texttt{alldifferent}(x_1, \ldots, x_{n-1}) \end{array} \tag{7}$$

has a solution, then we are done. Hence, we may assume the opposite, i.e. in (7), for each $d \in D(x_n)$, there exists a subset $K \subseteq \{x_1, \ldots, x_{n-1}\}$ with $|K| > |D(K) \setminus \{d\}|$. Then, by induction, $\texttt{alldifferent}(x_1, \ldots, x_{n-1})$, with $x_1 \in D(x_1), \ldots, x_{n-1} \in D(x_{n-1})$, has a solution if and only if for each $d \in D(x_n)$ there is a tight subset $K \subseteq \{x_1, \ldots, x_{n-1}\}$ with $d \in D(K)$. Choose any such tight subset K. Without loss of generality, $K = \{x_1, \ldots, x_k\}$. By induction, $\texttt{alldifferent}(x_1, \ldots, x_k)$ has a solution, using all values in $D(K)$. Moreover,

$$\begin{array}{l} x_{k+1} \in D(x_{k+1}) \setminus D(K), \ldots, x_n \in D(x_n) \setminus D(K), \\ \texttt{alldifferent}(x_{k+1}, \ldots, x_n) \end{array}$$

has a solution. This follows inductively, since for each $L \subseteq \{x_{k+1}, \ldots, x_n\}$,

$$\left| \bigcup_{x_i \in L} (D(x_i) \setminus D(K)) \right| = \left| \bigcup_{x_i \in K \cup L} D(x_i) \right| - |D(K)| \overset{(6)}{\geq} |K \cup L| - |D(K)| = |K| + |L| - |D(K)| = |L| \,.$$

Then $\texttt{alldifferent}(x_1, \ldots, x_n)$ has a solution, using all values in $D(K) \cup D(L)$. □

The following example shows an application of Theorem 5.

Example 4 *Consider the following CSP*

$$\begin{array}{l} x_1 \in \{2,3\}, x_2 \in \{2,3\}, x_3 \in \{1,2,3\}, x_4 \in \{1,2,3\}, \\ \texttt{alldifferent}(x_1, x_2, x_3, x_4). \end{array}$$

For any $K \subseteq \{x_1, x_2, x_3, x_4\}$ *with* $|K| \leq 3$, $|K| \leq |D(K)|$. *For* $K = \{x_1, x_2, x_3, x_4\}$ *however,* $|K| > |D(K)|$, *and by Theorem 5 this CSP has no solution.*

6. Here we assume that marriage is restricted to two persons of different sex.

Algorithm 1: Arc consistency for the binary decomposition of `alldifferent`$(x_1, x_2, \ldots, x_n)$.

```
Q := ∅
for i ∈ {1, ..., n} do
    if |D(x_i) = 1| then Q := Q + x_i
while Q ≠ ∅ do
    select x_i ∈ Q
    Q := Q − x_i
    for j ∈ {1, ..., n} − i do
        if D(x_j) ∩ D(x_i) ≠ ∅ then
            D(x_j) := D(x_j) \ D(x_i)
            if D(x_j) = ∅ then return false
            if |D(x_j)| = 1 then Q := Q + x_j
return true
```

4. Filtering Algorithms for Local Consistency Notions

This section analyzes four local consistency notions that are applied to the `alldifferent` constraint: local consistency of a decomposed CSP, bounds consistency, range consistency and hyper-arc consistency. For each local consistency notion a corresponding filtering algorithm is presented.

4.1 Local Consistency of a Decomposed CSP

In order to apply arc consistency, we decompose the `alldifferent` constraint into a set of binary constraints that preserves the solution set.

Definition 8 (Binary decomposition) *Let C be a constraint on the variables $x_1, \ldots, x_n$. A* binary decomposition *of C is a minimal set of binary constraints $C_{\text{dec}} = \{C_1, \ldots, C_k\}$ (for integer $k > 0$) on pairs of variables from $x_1, \ldots, x_n$ such that the solution set of C equals the solution set of $\bigcap_{i=1}^{k} C_i$.*[7]

The binary decomposition of `alldifferent`$(x_1, x_2, \ldots, x_n)$ is

$$\bigcup_{1 \leq i < j \leq n} \{x_i \neq x_j\}. \tag{8}$$

A filtering algorithm that establishes arc consistency on the binary decomposition (8) is presented as Algorithm 1. It has the following behaviour. Whenever the domain of a variable contains only one value, this value is removed from the domains of the other variables that occur in the `alldifferent` constraint. This procedure is repeated as long as no more changes occur or a domain becomes empty.

7. Note that we can extend the definition of binary decomposition by introducing new variables on which we define (some of) the binary constraints in C_{dec}. In that case we apply a mapping of the solution set of $\bigcap_{i=1}^{k} C_i$ to the solution set of C and vice versa, as proposed by (Rossi, Petrie, & Dhar, 1990). In this paper this extension is not necessary, however.

Theorem 6 *Algorithm 1 establishes arc consistency on the binary decomposition of* `alldifferent`$(x_1, x_2, \ldots, x_n)$ *or proves that it is inconsistent.*

Proof.

Algorithm 1 indeed yields arc consistency on the binary decomposition of the `alldifferent` constraint. Namely, we cannot deduce anything when a domain contains more than one element. □

One of the drawbacks of this algorithm is that one needs $\frac{1}{2}(n^2 - n)$ not-equal constraints to express an n-ary `alldifferent` constraint. Moreover, the worst-case time complexity of this method is $O(n^2)$, as shown by the following example.

Example 5 *For some integer $n > 1$, consider the CSP*

$$P = \begin{cases} x_1 \in \{1\}, x_i \in \{1, 2, \ldots, i\} \;\; \textit{for } i = 2, 3, \ldots, n, \\ \bigcup_{1 \leq i < j \leq n} \{x_i \neq x_j\} \end{cases}$$

If we make P arc consistent, we start by removing $\{1\}$ from all domains other than $D(x_1)$. Next we need to remove value $\{i\}$ from all domains other than $D(x_i)$, for $i = 2, \ldots, n$. This procedure takes in total $n(n-1)$ steps.

Another, even more important, drawback of the above method is the loss of information. When the set of binary constraints is being made arc consistent, only two variables are compared at a time. However, when the `alldifferent` constraint is being made hyper-arc consistent, all variables are considered at the same time, which allows a much stronger local consistency. This is shown in Theorem 7; see also (Stergiou & Walsh, 1999).

Theorem 7 *Let P be a CSP and P_{dec} the same CSP in which all* `alldifferent` *constraints have been replaced by their binary decomposition. Then $\Phi_{\text{HA}}(P) \preceq \Phi_{\text{A}}(P_{\text{dec}})$.*

Proof. To show that $\Phi_{\text{HA}}(P) \preceq \Phi_{\text{A}}(P_{\text{dec}})$, consider a value $d \in D_i$ for some i that is removed after making P_{dec} arc consistent. This removal must be due to the fact that $x_j = d$ for some $j \neq i$. But then $d \in D_i$ is also removed when making P hyper-arc consistent. The converse is not true, as illustrated in Example 6. □

Example 6 *For some integer $n \geq 3$ consider the CSPs*

$$P = \begin{cases} x_i \in \{1, 2, \ldots, n-1\} \;\; \textit{for } i = 1, 2, \ldots, n-1, \\ x_n \in \{1, 2, \ldots, n\}, \\ \texttt{alldifferent}(x_1, x_2, \ldots, x_n), \end{cases}$$

$$P_{\text{dec}} = \begin{cases} x_i \in \{1, 2, \ldots, n-1\} \;\; \textit{for } i = 1, 2, \ldots, n-1, \\ x_n \in \{1, 2, \ldots, n\}, \\ \bigcup_{1 \leq i < j \leq n} \{x_i \neq x_j\}. \end{cases}$$

Then $\Phi_{\text{A}}(P_{\text{dec}}) \equiv P_{\text{dec}}$, while

$$\Phi_{\text{HA}}(P) = \begin{cases} x_i \in \{1, 2, \ldots, n-1\} \;\; \textit{for } i = 1, 2, \ldots, n-1, \\ x_n \in \{n\}, \\ \texttt{alldifferent}(x_1, x_2, \ldots, x_n). \end{cases}$$

Our next goal is to find a local consistency notion for the binary decomposition that is equivalent to the hyper-arc consistency notion for the `alldifferent` constraint. Relational consistency (Dechter & van Beek, 1997) is used for this purpose.

Definition 9 (Relational $(1,m)$-consistency) *A set of constraints $S = \{C_1, C_2, \ldots, C_m\}$ is relationally $(1,m)$-consistent if all domain values $d \in D_i$ of variables appearing in S, appear in a solution to the m constraints, evaluated simultaneously. A CSP $P = (\mathcal{X}, D, C)$ is relationally $(1,m)$-consistent if every set of m constraints $S \subseteq \mathcal{C}$ is relationally $(1,m)$-consistent.*

Note that arc consistency is equivalent to relational $(1,1)$-consistency.

Let $\Phi_{R(1,m)C}(P)$ denote the CSP after achieving relational $(1,m)$-consistency on a CSP P.

Theorem 8 *Let $X = x_1, x_2, \ldots, x_n$ be a sequence of variables with respective finite domains $D = D_1, D_2, \ldots, D_n$. Let $P = (X, D, C)$ be the CSP with $C = \{\texttt{alldifferent}(x_1, \ldots, x_n)\}$ and let $P_{\text{dec}} = (X, D, C_{\text{dec}})$ be the CSP with $C = \bigcup_{1 \le i < j \le n} \{x_i \neq x_j\}$. Then*

$$\Phi_{\text{HA}}(P) \equiv \Phi_{R(1,\frac{1}{2}(n^2-n))C}(P_{\text{dec}}).$$

Proof. By construction, the `alldifferent` constraint is equivalent to the simultaneous consideration of the sequence of corresponding not-equal constraints. The number of not-equal constraints is precisely $\frac{1}{2}(n^2-n)$. If we consider only $\frac{1}{2}(n^2-n)-i$ not-equal constraints simultaneously ($1 \le i \le \frac{1}{2}(n^2-n)-1$), there are i unconstrained relations between variables, and the corresponding variables could take the same value when a certain instantiation is considered. Therefore, we really need to take all $\frac{1}{2}(n^2-n)$ constraints into consideration, which corresponds to relational $(1, \frac{1}{2}(n^2-n))$-consistency. □

As suggested before, the pruning performance of $\Phi_{\text{A}}(P_{\text{dec}})$ is rather poor. Moreover, the time complexity is relatively high, namely $O(n^2)$, as shown in Example 5. Nevertheless, this filtering algorithm applies quite well to several problems, such as the n-queens problem ($n < 200$), as indicated by (Leconte, 1996) and (Puget, 1998).

Further comparison of non-binary constraints and their corresponding decompositions are given by (Stergiou & Walsh, 1999) and (Gent, Stergiou, & Walsh, 2000). In particular the `alldifferent` constraint and its binary decomposition are extensively studied.

4.2 Bounds Consistency

A bounds consistency filtering algorithm for the `alldifferent` constraint was first introduced by (Puget, 1998). We summarize his method in this section. The idea is to use Hall's Marriage Theorem to construct an algorithm that establishes bounds consistency.

Definition 10 (Hall interval) *Let $x_1, x_2, \ldots, x_n$ be variables with respective finite domains $D_1, D_2, \ldots, D_n$. Given an interval I, define $K_I = \{x_i \mid D_i \subseteq I\}$. I is a* Hall interval *if $|I| = |K_I|$.*

Theorem 9 *The constraint* `alldifferent`$(x_1, \ldots, x_n)$ *is bounds consistent if and only if* $|D_i| \geq 1$ $(i = 1, \ldots, n)$ *and*

i) for each interval I*:* $|K_I| \leq |I|$*,*

ii) for each Hall interval I*:* $\{\min D_i, \max D_i\} \cap I = \varnothing$ *for all* $x_i \notin K_I$*.*

Proof. Let I be a Hall interval and $x_i \notin K_I$. If `alldifferent`$(x_1, \ldots, x_n)$ is bounds consistent, it has a solution when $x_i = \min D_i$, by Definition 4. From Theorem 5 immediately follows that $\min D_i \notin I$. Similarly for $\max D_i$.

Conversely, suppose `alldifferent`$(x_1, \ldots, x_n)$ is not bounds consistent. Thus, there exist a variable x_i and a value $d_i \in \{\min D_i, \max D_i\}$ for some $i \in \{1, \ldots, n\}$, such that `alldifferent`$(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ has no solution, where $x_j \in D'_j = [\min(D_j \setminus \{d_i\}), \max(D_j \setminus \{d_i\})]$ for all $j \neq i$. By Theorem 5, there exists some $K \subseteq \{x_1, \ldots, x_n\} \setminus \{x_i\}$ such that $|K| > |D'_K|$. Choose $I = D'_K$ and consider K_I with respect to `alldifferent`$(x_1, \ldots, x_n)$. Then either I is a Hall interval and $d_i \in I$, or $|K_I| > |I|$. □

Example 7 *Consider the following CSP*

$$x_1 \in \{1, 2\}, x_2 \in \{1, 2\}, x_3 \in \{2, 3\},$$
`alldifferent`(x_1, x_2, x_3).

Observe that the variables x_1 *and* x_2 *both have domain* $\{1, 2\}$*, Hence, values 1 and 2 cannot be assigned to any other variable and therefore, value 2 should be removed from* D_3*.*

The algorithm detects this when the interval I *is set to* $I = \{1, 2\}$*. Then the number of variables for which* $D_i \subseteq I$ *is 2. Since* $|I| = 2$*,* I *is a Hall interval. The domain of* x_3 *is not in this interval, and* $\{\min D_3, \max D_3\} \cap I = \{\min D_3\}$*. In order to obtain the empty set in the right hand side of the last equation, we need to remove* $\min D_i$*. The resulting CSP is bounds consistent.*

Theorem 9 is used to construct an algorithm that establishes bounds consistency on the `alldifferent` constraint. Consider all intervals $I = [l, u]$ where l ranges over all minimum domain values and u over all maximum domain values. There are maximally n^2 such intervals, as there are n variables. Count the number of variables that are contained in I. If a Hall interval is detected, update the bounds of the appropriate variables. This can be done in $O(n)$ steps. Hence the time complexity of this algorithm is $O(n^3)$.

A more efficient algorithm, presented as Algorithm 2, is obtained by first sorting the variables. To update the minimum domain values, we sort the variables in increasing order of their maximum domain value. Then we maintain an interval by the minimum and maximum bounds of the variables, which are inserted in the above order. Whenever we detect a *maximal* Hall interval, we update the bounds of the appropriate variables, reset the interval and continue with the next variable. To update the maximum domain values, we can apply the same method after interchanging $[\min D_i, \max D_i]$ with $[-\max D_i, -\min D_i]$ for all i.

The sorting of the variables can be done in $O(n \log n)$ time. As shown by (Puget, 1998), we can use a balanced binary tree to keep track of the number of variables within an interval,

Algorithm 2: Bounds consistency for `alldifferent`$(x_1, x_2, \ldots, x_n)$.

```
sort x_1, x_2, ..., x_n according to max D(x_i)
Q := ∅
for i ∈ {1, ..., n} do
    if |D(x_i) = 1| then Q := Q + x_i
while Q ≠ ∅ do
    select x_i ∈ Q
    Q := Q − x_i
    for j ∈ {1, ..., n} − i do
        if D(x_j) ∩ D(x_i) ≠ ∅ then
            D(x_j) := D(x_j) \ D(x_i)
            if D(x_j) = ∅ then return false
            if |D(x_j)| = 1 then Q := Q + x_j
return true
```

which allows updates in $O(\log n)$ per variable. Hence the total time complexity reduces to $O(n \log n)$.

An improvement on top of this was suggested and implemented by (Lopez-Ortiz, Quimper, Tromp, & van Beek, 2003). They noticed that while keeping track of the number of variables within an interval, some of the used counters are irrelevant. They give two implementations, one with a linear running time plus the time needed to sort the variables, and one with an $O(n \log n)$ running time. Their experiments show that the latter algorithm is faster in practice.

A different algorithm for achieving bounds consistency of the `alldifferent` constraint was presented by (Mehlhorn & Thiel, 2000). Instead of Hall intervals, they exploit the correspondence with finding a matching in a bipartite graph, similar to the algorithm presented in Section 4.4. Their algorithm runs in $O(n)$ time plus the time needed to sort the variables according to the bounds of the domains.

Although the worst-case time complexity of the above algorithms is always $O(n \log n)$, under certain conditions the sorting can be performed in linear time, which makes the algorithms by (Lopez-Ortiz et al., 2003) and (Mehlhorn & Thiel, 2000) run in linear time. This is the case in many practical instances, for example when the variables encode a permutation.

4.3 Range Consistency

(Leconte, 1996) introduced an algorithm that establishes range consistency for the `alldifferent` constraint. To explain this algorithm we follow the same procedure as in the previous subsection. Again we use Hall's Marriage Theorem to construct the algorithm.

Definition 11 (Hall set) *Let $x_1, x_2, \ldots, x_n$ be variables with respective finite domains $D_1, D_2, \ldots, D_n$. Given $K \subseteq \{x_1, \ldots, x_n\}$, define the interval $I_K = [\min D_K, \max D_K]$. K is a* Hall set *if $|K| = |I_K|$.*

Algorithm 3: Range consistency for `alldifferent`$(x_1, x_2, \ldots, x_n)$.

```
sort x_1, x_2, ..., x_n according to max D(x_i)
Q := ∅
for i ∈ {1, ..., n} do
    if |D(x_i) = 1| then Q := Q + x_i
while Q ≠ ∅ do
    select x_i ∈ Q
    Q := Q − x_i
    for j ∈ {1, ..., n} − i do
        if D(x_j) ∩ D(x_i) ≠ ∅ then
            D(x_j) := D(x_j) \ D(x_i)
            if D(x_j) = ∅ then return false
            if |D(x_j)| = 1 then Q := Q + x_j
return true
```

Note that in the above definition I_K does not necessarily need to be a Hall interval.

Theorem 10 *The constraint* `alldifferent`$(x_1, \ldots, x_n)$ *is range consistent if and only if* $|D_i| \geq 1$ $(i = 1, \ldots, n)$ *and* $D_i \cap I_K = \varnothing$ *for each Hall set* $K \subseteq \{x_1, \ldots x_n\}$ *and each* $x_i \notin K$.

Proof. Let K be a Hall set and $x_i \notin K$. If `alldifferent`$(x_1, \ldots, x_n)$ is range consistent, it has a solution when $x_i = d$ for all $d \in D_i$, by Definition 5. From Theorem 5 immediately follows that $D_i \cap I_K = \varnothing$.

Conversely, suppose `alldifferent`$(x_1, \ldots, x_n)$ is not range consistent. Thus, there exist a variable x_i and a value $d_i \in D_i$ for some $i \in \{1, \ldots, n\}$, such that `alldifferent`$(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ has no solution, where $x_j \in D'_j = [\min D_j \setminus \{d_i\}, \max D_j \setminus \{d_i\}]$ for all $j \neq i$. By Theorem 5, there is some $K \subseteq \{x_1, \ldots, x_n\} \setminus \{x_i\}$ with $|K| > |D'_K|$. Note that $D'_K = I_K$. Consider I_K with respect to `alldifferent`$(x_1, \ldots, x_n)$. If K is a Hall set, then $d_i \in I_K$. Otherwise, $|K| > |I_K|$. Then either some domain is empty, or K contains a Hall set K', and $D_j \cap I_{K'} \neq \varnothing$ for some $x_j \in K \setminus K'$. □

We can deduce a first filtering algorithm from Theorem 10 in a similar way as we did for bounds consistency. Namely, consider all intervals $I = [l, u]$ where l ranges over all minimum domain values and u over all maximum domain values. Then we count again the number of variables that are contained in I. If a Hall set is detected, we update the domains of the appropriate variables. This is done in $O(n)$ steps. Hence the time complexity of this algorithm is $O(n^3)$, as there are again maximally n^2 intervals to consider.

A faster algorithm, due to (Leconte, 1996), is presented as Algorithm 3. We first sort (and store) the variables twice, according to their minimum and maximum domain value, respectively. The main loop considers the variables ordered by their maximum domain value. For each such variable, we maintain the interval I_K and start adding variables to K. For this we consider all variables (inner loop), now sorted by their minimum domain value. When we detect a Hall set, updating the domains can be done in $O(1)$ time, either within or after the inner loop, and we proceed with the next variable. As sorting the variables

can be done in $O(n \log n)$ time, the total time complexity of this algorithm is $O(n^2)$. This time complexity is optimal, as is illustrated in the following example, taken from (Leconte, 1996).

Example 8 *Consider the following CSP*

$$\begin{array}{ll} x_i \in \{2i+1\} & \text{for } i = 0, 1, \ldots, n, \\ x_i \in \{0, 1, \ldots, 2n+2\} & \text{for } i = n+1, n+2, \ldots, 2n, \\ \texttt{alldifferent}(x_0, x_1, \ldots, x_{2n}). & \end{array}$$

In order to make this CSP range consistent, we have to remove the $n+1$ first odd integers from the domains of the n variables whose domain is not yet a singleton. This takes $O(n^2)$ time.

Observe that this algorithm has an opposite viewpoint from the algorithm for bounds consistency, although it looks similar. Where the algorithm for bounds consistency takes the domains (or intervals) as a starting point, the algorithm for range consistency takes the variables instead. But they both attempt to reach a situation in which the cardinality of a set of variables is equal to the cardinality of the union of the corresponding domains.

4.4 Hyper-arc Consistency

A hyper-arc consistency filtering algorithm for the `alldifferent` constraint was proposed by (Régin, 1994). The algorithm is based on matching theory (see Section 2.2.2 and Section 3.1). We first give a characterization in terms of Hall's Marriage Theorem.

Definition 12 (Tight set) *Let $x_1, x_2, \ldots, x_n$ be variables with respective finite domains $D_1, D_2, \ldots, D_n$. $K \subseteq \{x_1, \ldots, x_n\}$ is a* tight set *if $|K| = |D_K|$.*

Theorem 11 *The constraint $\texttt{alldifferent}(x_1, \ldots, x_n)$ is hyper-arc consistent if and only if $|D_i| \geq 1$ $(i = 1, \ldots, n)$ and $D_i \cap D_K = \varnothing$ for each tight set $K \subseteq \{x_1, \ldots x_n\}$ and each $x_i \notin K$.*

Proof. Let K be a tight set and $x_i \notin K$. If $\texttt{alldifferent}(x_1, \ldots, x_n)$ is hyper-arc consistent, it has a solution when $x_i = d$ for all $d \in D_i$, by Definition 3. From Theorem 5 immediately follows that $D_i \cap D_K = \varnothing$.

Conversely, suppose $\texttt{alldifferent}(x_1, \ldots, x_n)$ is not hyper-arc consistent. Thus, there exist a variable x_i and a value $d_i \in D_i$ for some $i \in \{1, \ldots, n\}$, such that $\texttt{alldifferent}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ has no solution, where $x_j \in D'_j = D_j \setminus \{d_i\}$ for all $j \neq i$. By Theorem 5, $|K| > |D'_K|$ for some $K \subseteq \{x_1, \ldots, x_n\} \setminus \{x_i\}$. If K is a tight set with respect to $\texttt{alldifferent}(x_1, \ldots, x_n)$, then $d_i \in D_K$. Otherwise, $|K| > |D_K|$. Then either some domain is empty, or K contains a tight set K', and $D_j \cap D_{K'} \neq \varnothing$ for some $x_j \in K \setminus K'$. □

Following Theorem 11, the `alldifferent` constraint can be made hyper-arc consistent by generating all tight sets K and updating $D_i = D_i \setminus D_K$ for all $x_i \notin K$. This approach is similar to the algorithms for achieving bounds consistency and range consistency. For bounds consistency and range consistency we could generate the respective Hall intervals and Hall sets rather easily because we were dealing with intervals containing the

domains. In order to generate tight sets similarly we should consider all possible subsets $K \subseteq \{x_1, \ldots, x_n\}$. As the number of subsets is exponential in n, this approach is not practical. A different, more constructive, approach makes use of matching theory to update the domains, and was introduced by (Régin, 1994).

Theorem 12 *Let G be the value graph of a sequence of variables $X = x_1, x_2, \ldots, x_n$ with respective finite domains $D_1, D_2, \ldots, D_n$. The constraint* $\texttt{alldifferent}(x_1, \ldots, x_n)$ *is hyper-arc consistent if and only if every edge in G belongs to a matching in G covering X.*

Proof. Immediate from Definition 3 and Theorem 4. □

The following Theorem identifies edges that belong to a maximum-size matching. The proof follows from (Petersen, 1891); see also (Schrijver, 2003, Theorem 16.1).

Theorem 13 *Let G be a graph and M a maximum-size matching in G. An edge belongs to a maximum-size matching in G if and only if it either belongs to M, or to an even M-alternating path starting at an M-free vertex, or to an even M-alternating circuit.*

Proof. Let M be a maximum-size matching in $G = (V, E)$. Suppose edge e belongs to a maximum-size matching N, and $e \notin M$. The graph $G' = (V, M \cup N)$ consists of even paths (possibly of length 0) and circuits with edges alternatingly in M and N. If the paths are not of even length, M or N can be made larger by interchanging edges in M and N along this path (a contradiction because they are of maximum size).

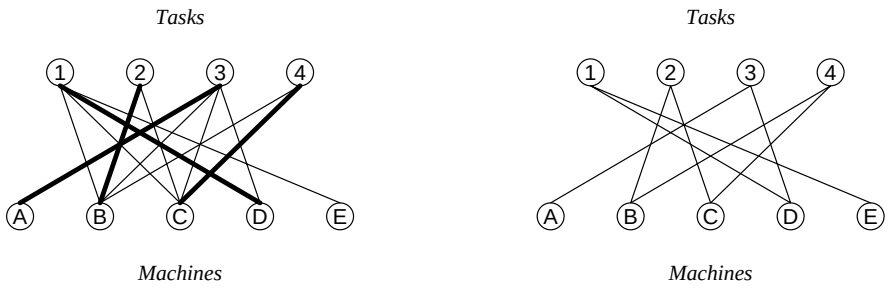
Conversely, let M be a maximum-size matching in G. By interchanging edges in M and not in M along even M-alternating paths starting at an M-free vertex and M-alternating circuits we obtain matchings of maximum size again. □

To establish a connection between Theorem 11 and Theorem 13, consider a tight set $K \subset \{x_1, \ldots, x_n\}$ of minimum size. The edges between vertices in K and in D_K form M-alternating circuits in the value graph. By applying Theorem 13 we remove those edges $x_i d$ with $x_i \notin K$ and $d \in D_K$, i.e. $D_i \cap D_K = \varnothing$. This corresponds to applying Theorem 11.

Example 9 *Consider again the task assignment problem of Example 3. A solution to this problem is given in Figure 2.a, where bold edges denote the corresponding matching M covering the tasks. Machine* E *is the only M-free vertex. All even M-alternating paths starting from* E *belong to a solution, by Theorem 13. Thus, all edges on the path* $\mathsf{E}, 1, \mathsf{D}, 3, \mathsf{A}$ *belong to a solution. The only M-alternating circuit in the graph is* $2, \mathsf{B}, 4, \mathsf{C}, 2$*, and by Theorem 13 all edges in this circuit belong to a solution. The other edges, i.e.* $(1, \mathsf{B})$*,* $(1, \mathsf{C})$*,* $(3, \mathsf{B})$ *and* $(3, \mathsf{C})$*, should be removed to make the* `alldifferent` *constraint hyper-arc consistent. The result is depicted in Figure 2.b. which corresponds to the CSP*

$$x_1 \in \{\mathsf{D}, \mathsf{E}\}, x_2 \in \{\mathsf{B}, \mathsf{C}\}, x_3 \in \{\mathsf{A}, \mathsf{D}\}, x_4 \in \{\mathsf{B}, \mathsf{C}\},$$
$$\texttt{alldifferent}(x_1, x_2, x_3, x_4).$$

Using Theorem 13, we construct a hyper-arc consistency filtering algorithm, presented as Algorithm 4. First we compute a maximum-size matching M in the value graph $G = (X \cup D_X, E)$. This can be done in $O(m\sqrt{n})$ time, using the algorithm by (Hopcroft & Karp,



a. A matching covering all tasks. b. The bipartite graph after filtering.

Figure 2: The bipartite graph for the task assignment problem of Example 9.

Algorithm 4: Hyper-arc consistency for `alldifferent`$(x_1, x_2, \ldots, x_n)$.

```
% identify X = x_1, x_2, ..., x_n
build value graph G = (X, D(X), E)
compute maximum matching M in G
if |M| < |X| then return false
mark all arcs in G_M that are not in M as unused
compute SCCs in G_M and mark all arcs in a SCC as used
perform breadth-first in G_M search starting from M-free vertices, and mark all
traversed arcs as used
for all arcs (x_i, d) in G_M marked as unused do
    D(x_i) := D(x_i) - d
    if D(x_i) = ∅ then return false
return true
```

1973), where $m = \sum_{i=1}^{n} |D_i|$. 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 graph $G_M = (X \cup D_X, A)$ with arc set $A = \{(v_1, v_2) \mid (v_1, v_2) \in M\} \cup \{(v_2, v_1) \mid (v_1, v_2) \notin M\}$. In other words, edges in M are being directed from X (the variables) to D_X (the domain values). Edges not in M are being directed reversely. We first compute the strongly connected components in G_M. This can be done in $O(n + m)$ time, following (Tarjan, 1972). Arcs between vertices in the same strongly connected component belong to an even M-alternating circuit in G, and are marked as "consistent". 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 "consistent". For all edges $x_i d$ that are not marked "consistent" and do not belong to M, we update $D_i = D_i \setminus \{d\}$. Then, by Theorem 13, the corresponding `alldifferent` constraint is hyper-arc consistent.

From the above follows that the `alldifferent` constraint is checked for consistency, i.e. checked to contain a solution, in $O(m\sqrt{n})$ time and made hyper-arc consistent in $O(m)$ time. We could also have applied a general algorithm to establish hyper-arc consistency on an n-ary constraint. Such algorithm was presented by (Mohr & Masini, 1988). For an

`alldifferent` constraint on n variables, the time complexity of their general algorithm is $O(\frac{d!}{(d-n)!})$, where d is the maximum domain size. Clearly, the above specialized algorithm is much more efficient.

During the whole solution process of the CSP, constraints other than `alldifferent` might also be used to remove values. In such cases, we must update our `alldifferent` constraint. As indicated by (Régin, 1994), this can be done incrementally, i.e. we can make use of our current value graph and our current maximum-size matching to compute a new maximum-size matching. This is less time consuming than restarting the algorithm from scratch. The same idea has been used by (Barták, 2003) to make the `alldifferent` constraint dynamic with respect to the addition of variables during the solution process.

4.5 Complexity Survey and Discussion

Table 2 present a time complexity survey of the algorithms that obtain the four notions of local consistency that have been applied to the `alldifferent` constraint in this section. Here n is again the number of variables inside the `alldifferent` constraint and m the sum of the cardinalities of the domains.

arc consistency	$O(n^2)$	(Van Hentenryck, 1989)
bounds consistency	$O(n \log n)$	(Puget, 1998), (Mehlhorn & Thiel, 2000), (Lopez-Ortiz et al., 2003)
	$O(n)$ (special cases)	(Mehlhorn & Thiel, 2000), (Lopez-Ortiz et al., 2003)
range consistency	$O(n^2)$	(Leconte, 1996)
hyper-arc consistency	$O(m\sqrt{n})$	(Régin, 1994)

Table 2: Survey of time complexity of four local consistency notions.

While increasing the strength of the local consistency notions from bounds consistency to range consistency and hyper-arc consistency, we have seen that the underlying principle remains the same. It simply boils down to the refinement of the sets to which we apply Hall's Marriage Theorem.

In practice, none of these local consistency notions always outperforms all others. It is strongly problem-dependent which local consistency is suited best. In general, however, bounds consistency is most suitable when domains are always closed intervals. As more holes may occur in domains, hyper-arc consistency becomes more useful. Another observation concerns the implementation of an algorithm that establishes bounds consistency. Although all three variants in Table 2 have the same worst-case complexity, the one proposed by (Lopez-Ortiz et al., 2003) appears to be most efficient in practice.

A general comparison of bounds consistency and hyper-arc consistency with respect to the search space has been made by (Schulte & Stuckey, 2001). In particular, attention is also paid to the `alldifferent` constraint.

5. Variants of the Alldifferent Constraint

This section presents three variants of the `alldifferent` constraint: the symmetric `alldifferent` constraint, the weighted `alldifferent` constraint and the soft `alldifferent` constraint. The first two variants exploit additional information in conjunction with the `alldifferent` constraint. The last variant allows the `alldifferent` constraint to be partially violated.

5.1 The Symmetric Alldifferent Constraint

A particular case of the `alldifferent` constraint, the symmetric `alldifferent` constraint, was introduced by (Régin, 1999b). We assume that the variables and their domain values represent the same set of elements. The symmetric `alldifferent` constraint states that all variables must take different values, and if the variable representing element i is assigned to the value representing element j, then the variable representing the element j must be assigned to the value representing element i. A more formal definition is presented below.

The `symm_alldifferent` constraint is particularly suitable for round-robin tournament problems. In such problems, for example a sports competition, each team has to be matched with another team. Typically, there are many more constraints involved than only the `symm_alldifferent` constraint, which makes the problems often very difficult to solve. The filtering of the `alldifferent` constraint and the `symm_alldifferent` constraint has been analyzed for practical problem instances by (Henz, Müller, & Thiel, 2004). They show that constraint programming, using the `symm_alldifferent` constraint, outperforms other approaches (such as Operations Research methods) with several orders of magnitude for these instances.

Definition 13 (Symmetric alldifferent constraint) *Let $x_1, x_2, \ldots, x_n$ be variables with respective finite domains $D_1, D_2, \ldots, D_n \subseteq \{1, 2, \ldots, n\}$. Then*

$$\texttt{symm\_alldifferent}(x_1, \ldots, x_n) = \\ \{(d_1, \ldots, d_n) \mid d_i \in D_i, d_i \neq d_j \text{ for } i \neq j, d_i = j \Leftrightarrow d_j = i \text{ for } i \neq j\}.$$

In a CSP, the `symm_alldifferent` constraint can also be expressed as an `alldifferent` constraint together with one or more constraints that preserve the symmetry. Another representation can be made that uses the so-called `cycle` constraint, where each cycle must contain two vertices; see also (Beldiceanu et al., 2005). In that case, a cycle on two vertices x and y indicates that x is assigned to y and vice versa. However, the `symm_alldifferent` constraint captures more global information than the common `alldifferent` constraint together with additional constraints. Hence the `symm_alldifferent` constraint can be used to obtain a stronger filtering algorithm. The following example, taken from (Régin, 1999b) shows exactly this.

Example 10 *Consider a set of three people that have to be grouped in pairs. Each person can only be paired to one other person. This problem can be represented as a CSP by introducing a set of people $S = \{p_1, p_2, p_3\}$ that are pairwise compatible. These people are represented both by a set of variables x_1, x_2 and x_3 and by a set of values v_1, v_2 and v_3,*

where x_i and v_i represent p_i. Then the CSP

$$x_1 \in \{v_2, v_3\}, x_2 \in \{v_1, v_3\}, x_3 \in \{v_1, v_2\},$$
$$x_1 = v_2 \Leftrightarrow x_2 = v_1,$$
$$x_1 = v_3 \Leftrightarrow x_3 = v_1,$$
$$x_2 = v_3 \Leftrightarrow x_3 = v_2,$$
$$\texttt{alldifferent}(x_1, x_2, x_3)$$

is hyper-arc consistent. However, the following CSP

$$x_1 \in \{v_2, v_3\}, x_2 \in \{v_1, v_3\}, x_3 \in \{v_1, v_2\},$$
$$\texttt{symm\_alldifferent}(x_1, x_2, x_3)$$

is inconsistent. Indeed, there exists no solution to this problem, as the number of variables is odd.

Suppose there exists a value $j \in D_i$, while $i \notin D_j$. Then we can immediately remove value j from D_i. Hence we assume that such situations do not occur in the following.

Similar to the common `alldifferent` constraint, the `symm_alldifferent` constraint can be expressed by a graph. Given a constraint $\texttt{symm\_alldifferent}(x_1, \ldots, x_n)$, construct the graph $G_{\text{symm}} = (X, E)$, with vertex set $X = \{x_1, x_2, \ldots, x_n\}$ and edge set $E = \{x_i x_j \mid x_i \in D_j, x_j \in D_i, i < j\}$. Note that we identify each variable x_i with value i for $i = 1, \ldots, n$. We denote the number of edges in G_{symm} by m, i.e. $m = \left(\sum_{i=1}^{n} |D_i|\right)/2$. An illustration of G_{symm} is given in the next example.

Example 11 *Consider the following CSP*

$$x_{\mathsf{a}} \in \{\mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}\}, \quad x_{\mathsf{b}} \in \{\mathsf{a}, \mathsf{c}, \mathsf{d}, \mathsf{e}\}, \quad x_{\mathsf{c}} \in \{\mathsf{a}, \mathsf{b}, \mathsf{d}, \mathsf{e}\}, \quad x_{\mathsf{d}} \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{e}\},$$
$$x_{\mathsf{e}} \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{i}, \mathsf{j}\}, \quad x_{\mathsf{f}} \in \{\mathsf{g}, \mathsf{h}\}, \quad x_{\mathsf{g}} \in \{\mathsf{f}, \mathsf{h}\}, \quad x_{\mathsf{h}} \in \{\mathsf{f}, \mathsf{g}, \mathsf{i}, \mathsf{j}\},$$
$$x_{\mathsf{i}} \in \{\mathsf{e}, \mathsf{h}, \mathsf{j}\}, \quad x_{\mathsf{j}} \in \{\mathsf{e}, \mathsf{h}, \mathsf{i}\},$$
$$\texttt{symm\_alldifferent}(x_{\mathsf{a}}, x_{\mathsf{b}}, \ldots, x_{\mathsf{j}}).$$

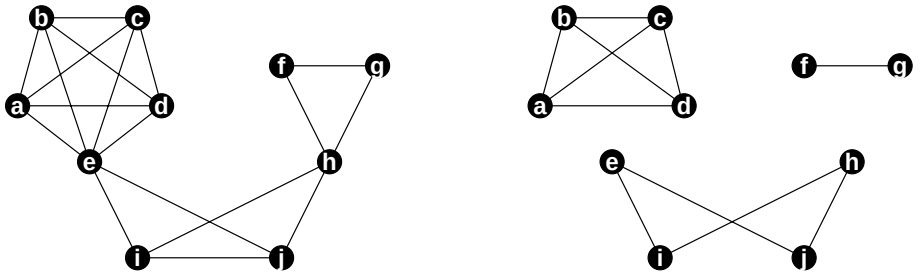
In Figure 3.a the corresponding graph G_{symm} is depicted, where $x_{\mathsf{a}}, x_{\mathsf{b}}, \ldots, x_{\mathsf{j}}$ are abbreviated to $\mathsf{a}, \mathsf{b}, \ldots, \mathsf{j}$.

Similar to Theorem 4, we have the following result.

Theorem 14 *Let $x_1, x_2, \ldots, x_n$ be a sequence of variables with respective finite domains $D_1, D_2, \ldots, D_n$. Then $(d_1, \ldots, d_n) \in \texttt{symm\_alldifferent}(x_1, \ldots, x_n)$ if and only if $M = \{x_i d_i \mid i < d_i\}$ is a matching in G_{symm}.*

Proof. An edge $x_i x_j$ in G_{symm} corresponds to the assignments $x_i = d_i$ and $x_{d_i} = i$, which are equivalent with $x_j = d_j$ and $x_{d_j} = j$, because $d_i = j$ and $d_j = i$. As no edges in a matching share a vertex, $x_i \neq x_j$ for all $i \neq j$. Finally, as M covers X, to all variables a value is assigned. □

We use Theorem 14 to make the `symm_alldifferent` constraint hyper-arc consistent.



a. The graph G_{symm}. b. The graph G_{symm} after filtering.

Figure 3: Filtering of the `symm_alldifferent` constraint of Example 11.

Theorem 15 *The constraint* `symm_alldifferent`$(x_1, \ldots, x_n)$ *is hyper-arc consistent if and only if every edge in the corresponding graph* G_{symm} *belongs to a matching that covers* $x_1, \ldots, x_n$.

Proof. By Definition 3 (hyper-arc consistency) and application of Theorem 14. □

Example 12 *Consider again the CSP of Example 11. Figure 3.b shows the graph of Figure 3.a after making the CSP hyper-arc consistent. Consider for example the subgraph induced by vertices f, g and h. Obviously vertices f and g have to be paired in order to satisfy the* `symm_alldifferent` *constraint. Hence, vertex h cannot be paired to f nor g, and the corresponding edges can be removed. The same holds for the subgraph induced by a, b, c, d and e, where the vertices a, b, c and d must form pairs.*

A matching in G_{symm} that covers $x_1, \ldots, x_n$ is also a maximum-size matching. From Theorem 13 we already know how to identify edges that belong to a maximum-size matching. Namely, given an arbitrary maximum-size matching M, they belong either to M, or to an even M-alternating path starting at a free vertex, or to an even M-alternating circuit. However, in the current case, G_{symm} does not need to be bipartite, so we cannot blindly apply the machinery from Section 4.4 to establish hyper-arc consistency for the `symm_alldifferent` constraint.

A hyper-arc consistency filtering algorithm for the `symm_alldifferent` constraint, presented as Algorithm 5, was proposed by (Régin, 1999b). First, we compute a maximum-size matching M in the (possibly non-bipartite) graph G_{symm}. This can be done in $O(m\sqrt{n})$ time by applying the algorithm by (Micali & Vazirani, 1980). If $|M| < n/2$, there is no solution. Otherwise, we need to detect all edges that can never belong to a maximum-size matching. Since there are no M-free vertices, we only need to check whether an edge that does not belong to M is part of an even M-alternating circuit. This can be done as follows. If for an edge $uv \in M$ we find an M-alternating path $u, \ldots, w, v$ (with $u \neq w$), we know that the edge wv is on an even M-alternating circuit. Moreover, we can compute all possible M-alternating paths from u to v, that avoid edge uv. All edges wv that are not on such a path cannot belong to an even M-alternating circuit. Namely, all M-alternating circuits through wv should also contain the matching edge uv. Hence, by Theorem 13 and Theorem 14 we can delete such edges wv. This procedure should be repeated for all vertices in G_{symm}.

Algorithm 5: Hyper-arc consistency for `symm_alldifferent`$(x_1, x_2, \ldots, x_n)$.

```
% identify X = x_1, x_2, ..., x_n
build graph G_symm = (X, E)
compute maximum matching M in G_symm
if |M| < |X|/2 then return false
mark all edges in G_symm that are not in M as unused
for all vertices x in G_symm do
    % let xd ∈ M
    compute all alternating x − d paths in G_symm − xd, and mark all edges on these
    paths as used
for all edges (x_i, d) in G_M marked as unused do
    D(x_i) := D(x_i) − d
    if D(x_i) = ∅ then return false
return true
```

The algorithm for computing the M-alternating paths can be implemented to run in $O(m)$ time, using results of (Edmonds, 1965) to take care of the nonbipartiteness, and (Tarjan, 1972) for the data structure, as was observed by (Régin, 1999b). Hence the total time complexity of the algorithm achieving hyper-arc consistency for the `symm_alldifferent` constraint is $O(nm)$.

Note that the above algorithm is not incremental, and may not be effective in all practical cases. For this reason, also an incremental algorithm was proposed by (Régin, 1999b), that does not ensure hyper-arc consistency, however. The algorithm computes once a maximum-size matching, and each incremental step has a time complexity of $O(m)$.

5.2 The Weighted Alldifferent Constraint

In this section we assume that for all variables in the `alldifferent` constraint each variable-value pair induces a cost. Eventually (the problem we want to solve involves also other constraints), the goal is to find a solution with minimum[8] total cost. In the literature, this combination is known as the constraint `MinWeightAllDiff` (Caseau & Laburthe, 1997), or `IlcAllDiffCost` (Focacci, Lodi, & Milano, 1999). This section shows how to exploit the `alldifferent` constraint and the minimization problem together as an "optimization constraint". First we give a definition of the weighted `alldifferent` constraint, and then we show how we to make it hyper-arc consistent. It should be noted that the weighted `alldifferent` constraint is a special case of the weighted global cardinality constraint (Régin, 1999a, 2002).

Definition 14 (Weighted alldifferent constraint) *Let $x_1, x_2, \ldots, x_n, z$ be variables with respective finite domains $D_1, D_2, \ldots, D_n, D_z$, and let $w_{ij} \in \mathbb{Q}_+$ for $i = 1, \ldots, n$ and all*

8. A maximization problem can be reformulated as a minimization problem by negating the objective function.

$j \in \bigcup_{i=1,\ldots,n} D_i$ *be constants. Then*

$$\texttt{minweight\_alldifferent}(x_1, \ldots, x_n, z, w) = \left\{(d_1, \ldots, d_n, \tilde{d}) \mid d_i \in D_i, \tilde{d} \in D_z, d_i \neq d_j \text{ for } i \neq j, \sum_{i,d_i=j} w_{ij} \leq \tilde{d}\right\}.$$

Note that for a pair (x_i, j) with $j \notin D_i$, we may define $w_{ij} = \infty$. The variable z in Definition 14 serves as a "cost" variable, which is minimized during the solution process. This means that admissible tuples are those instantiations of variables with total weight not more than the currently best found solution, represented by $\max D_z$. At the same time, $\min D_z$ should not be less than the currently lowest possible total weight.

In order to make the `minweight_alldifferent` constraint hyper-arc consistent, we introduce the directed graph $G_{\text{minweight}} = (V, A)$ with

$$V = \{s, t\} \cup X \cup D_X \text{ and } A = A_s \cup A_X \cup A_t$$

where $X = \{x_1, x_2, \ldots, x_n\}$ and

$$\begin{array}{rcl} A_s & = & \{(s, x_i) \mid i = 1, 2, \ldots, n\}, \\ A_X & = & \{(x_i, d) \mid d \in D_i\}, \\ A_t & = & \{(d, t) \mid d \in \bigcup_{i=1}^{n} D_i\}. \end{array}$$

To each arc $a \in A$, we assign a capacity $c(a) = 1$ and a weight $w(a)$. If $a = (x_i, j) \in A_X$, then $w(a) = w_{ij}$. If $a \in A_s \cup A_t$ then $w(a) = 0$.

Theorem 16 *The constraint* `minweight_alldifferent`$(x_1, \ldots, x_n, z, w)$ *is hyper-arc consistent if and only if*

i) for every arc $a \in A_X$ there exists an $s-t$ flow f under c in $G_{\text{minweight}}$ with $f(a) = 1$, $\text{value}(f) = n$ *and* $\text{weight}(f) \leq \max D_z$*, and*

ii) $\min D_z \geq \text{weight}(f)$ *for a minimum-weight $s-t$ flow f in $G_{\text{minweight}}$.*

Proof. We may assume that every feasible $s-t$ flow f of value n is integer, because c is integer. In fact, we may assume that f is $\{0,1\}$-valued, because $c(a) = 1$ for each arc $a \in A$. As $\text{value}(f) = n$, f uses exactly n arcs in A_X. An arc $a = (x_i, j) \in A_X$ with $f(a) = 1$ corresponds to assigning $x_i = j$. Because $c(a) = 1$ for all $a \in A_t$, each value is used at most once, which enforces the `alldifferent` constraint. Hence, if $\text{weight}(f) \leq \max D_z$, the corresponding assignment is a solution to the `minweight_alldifferent` constraint. □

Example 13 *Consider again the task assignment problem of Example 3. Suppose in addition that each task-machine combination induces a cost, as indicated in Table 3 below. The goal is to find an assignment with minimum cost. Denote the cost of assigning task i to machine j as w_{ij}. For the cost variable z we initially set* $\max D_z = 33$*, being the sum of the maximum assignment cost for each variable. Then we model the problem as the CSP*

$$\begin{array}{l} z \in \{0, 1, \ldots, 33\}, \\ x_1 \in \{\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}, x_2 \in \{\mathsf{B}, \mathsf{C}\}, x_3 \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}, x_4 \in \{\mathsf{B}, \mathsf{C}\}, \\ \texttt{minweight\_alldifferent}(x_1, x_2, x_3, x_4, z, w), \\ \texttt{minimize}\ z. \end{array}$$

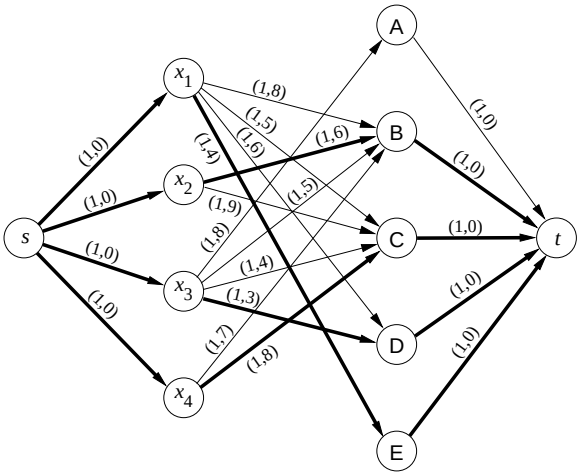


Figure 4: Graph $G_{\text{minweight}}$ for the `minweight_alldifferent` constraint of Example 13. For each arc a, $(c(a), w(a))$ is given. Bold arcs indicate a minimum-weight $s-t$ flow with weight 21.

	Machine				
Task	A	B	C	D	E
1	∞	8	5	6	4
2	∞	6	9	∞	∞
3	8	5	4	3	∞
4	∞	7	8	∞	∞

Table 3: Cost of task - machine combinations.

The corresponding graph $G_{\text{minweight}}$ is shown in Figure 4. The bold arcs in the graph denote a minimum-weight $s-t$ flow with weight 21. It corresponds to a solution to the COP, i.e. $x_1 = \mathsf{E}, x_2 = \mathsf{B}, x_3 = \mathsf{D}, x_4 = \mathsf{C}$ *and* $z = 21$.

Following Theorem 16, we can make the `minweight_alldifferent` constraint hyper-arc consistent by the following procedure:

> For all arcs $a = (x_i, j) \in A_X$, compute a minimum-weight $s-t$ flow f in $G_{\text{minweight}}$ with $f(a) = 1$. If $\text{value}(f) < n$ or $\text{weight}(f) > \max D_z$, update $D_i = D_i \setminus \{j\}$.
> For a minimum-weight $s-t$ flow f in $G_{\text{minweight}}$ with $\text{value}(f) = n$, update $\min D_z = \text{weight}(f)$ if $\min D_z < \text{weight}(f)$.

The drawback of the above procedure is that we have to compute a minimum-weight $s-t$ flow for each arc in A_X. It is more efficient however, to use the residual graph $(G_{\text{minweight}})_f$ and apply Theorem 3 as follows.

We first compute a minimum-weight flow f in $G_{\text{minweight}}$. For this we use the algorithm presented in Section 2.2.3. We need to compute n shortest $s-t$ paths in the residual graph, each of which takes $O(m + d \log d)$ time where $m = \sum_{i=1}^{n} |D_i|$ and $d = |\bigcup_{i=1}^{n} D_i|$. Thus the minimum-weight flow is computed in $O(n(m + d \log d))$ time. If $\text{weight}(f) > \max D_z$, the `minweight_alldifferent` constraint is inconsistent.

If $\text{weight}(f) \leq \max D_z$, we consider all arcs $a = (x_i, j) \in A_X$ with $f(a) = 0$ (see Theorem 3). We compute a minimum-weight $j - x_i$ path P in the residual graph $(G_{\text{minweight}})_f$, if it exists. If P does not exist, we update $D_i = D_i \setminus \{j\}$. Otherwise, P, a forms a directed circuit C of minimum weight, containing a. If $\text{weight}(f) + \text{weight}(C) > \max D_z$ we update $D_i = D_i \setminus \{j\}$. Finally, we update $\min D_z = \text{weight}(f)$ if $\min D_z < \text{weight}(f)$. Then, by Theorem 16 and Theorem 3 the `minweight_alldifferent` constraint is hyper-arc consistent. These last steps involve $m - n$ shortest path computations, each taking $O(m + d \log d)$ time. Hence the total time complexity for making the `minweight_alldifferent` constraint hyper-arc consistent is $O(m(m + d \log d))$. In fact, this can be improved to $O(n(m + d \log d))$, as was shown by (Régin, 1999a, 2002) and (Sellmann, 2002).

Other work concerning the `alldifferent` constraint in conjunction with an objective function has been done in (Lodi, Milano, & Rousseau, 2003). In that work, the so-called additive bounding procedure (Fischetti & Toth, 1989) and limited discrepancy search (Harvey & Ginsberg, 1995) are exploited in presence of an `alldifferent` constraint.

5.3 The Soft Alldifferent Constraint

Consider a CSP that is over-constrained, i.e. there exists no solution satisfying all constraints. In such occasions, it is natural to search for a solution that satisfies as many constraints as possible, and violates the other constraints. Constraints that are allowed to be violated are called *soft constraints*. The other constraints, which are not allowed to be violated, are called *hard constraints*. This section shows how we can handle the soft `alldifferent` constraint.

We follow the scheme proposed by (Régin, Petit, Bessière, & Puget, 2000) which is particularly useful for non-binary constraints. For each soft constraint $C(x_1, \ldots, x_n)$ we define a *violation measure* $\mu : D_1 \times \cdots \times D_n \to \mathbb{Q}$ and a "cost" variable z that represents the measure of violation. Then the CSP is transformed into a constraint optimization problem (COP), where all constraints are hard, and the (weighted) sum of cost variables is minimized. This approach allows one to use specialized filtering algorithms for soft constraints, as shown by (Petit, Régin, & Bessière, 2001). We first give a general definition of constraint softening, which we later apply to the `alldifferent` constraint.

Definition 15 (Constraint softening) *Let $x_1, x_2, \ldots, x_n, z$ be variables with respective finite domains $D_1, D_2, \ldots, D_n, D_z$. Let $C(x_1, \ldots, x_n)$ be a constraint with a violation measure $\mu(x_1, \ldots, x_n)$. Then*

$$\texttt{soft\_}C(x_1, \ldots, x_n, z, \mu) = \left\{ (d_1, \ldots, d_n, \tilde{d}) \mid d_i \in D_i, \tilde{d} \in D_z, \mu(d_1, \ldots, d_n) \leq \tilde{d} \right\}$$

is the soft version of C with respect to μ.

As in the previous section, 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, given the current state of the solution process.

We consider two measures of violation. 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 (see Section 4.1) that are violated. The latter measure is only defined for constraints for which a binary decomposition exists.

Let $X = x_1, \ldots, x_n$ be a sequence of variables with respective finite domains $D_1, \ldots, D_n$. For $\texttt{alldifferent}(x_1, \ldots, x_n)$ we have

$$\mu_{\text{var}}(x_1, \ldots, x_n) = \textstyle\sum_{d \in D_X} \max\left(|\{i \mid x_i = d\}| - 1, 0\right),$$
$$\mu_{\text{dec}}(x_1, \ldots, x_n) = \left|\{(i,j) \mid x_i = x_j, \text{ for } i < j\}\right|,$$

If we insert these measures into Definition 15 applied to the `alldifferent` constraint we obtain $\texttt{soft\_alldifferent}(x_1, \ldots, x_n, z, \mu_{\text{var}})$ and $\texttt{soft\_alldifferent}(x_1, \ldots, x_n, z, \mu_{\text{dec}})$.

Example 14 *Consider the following over-constrained CSP*

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

We have

$$\begin{array}{ll} \mu_{\text{var}}(a,a,b,c) = 1, & \mu_{\text{dec}}(a,a,b,c) = 1, \\ \mu_{\text{var}}(a,a,b,b) = 2, & \mu_{\text{dec}}(a,a,b,b) = 2, \\ \mu_{\text{var}}(a,a,a,b) = 2, & \mu_{\text{dec}}(a,a,a,b) = 3, \\ \mu_{\text{var}}(b,b,b,b) = 3, & \mu_{\text{dec}}(b,b,b,b) = 6. \end{array}$$

We transform the CSP into the following COP

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

This COP is not hyper-arc consistent, as there is no solution with $z < 1$. If we remove 0 from D_z, the COP is hyper-arc consistent, because there are at most 6 simultaneously violated not-equal constraints.

Suppose now that during the search for a solution, we have found the tuple $(x_1, x_2, x_3, x_4, z) = (a, a, b, c, 1)$, which has one violated not-equal constraint. Then $z \in \{1\}$ in the remaining search. As the assignment $x_4 = b$ always leads to a solution with $z \geq 2$, b can be removed from D_4. The resulting COP is hyper-arc consistent again.

One should take into account that a simplified CSP is considered in this example. In general, a CSP can consist of many more hard and soft constraints, and also more cost-variables that together with z form an objective function to be minimized.

Each of the violation measures μ_{var} and μ_{dec} gives rise to a hyper-arc consistency filtering algorithm.

5.3.1 Variable-Based Violation Measure

We first consider the variable-based violation measure. The results in this section are due to (Petit et al., 2001).

Theorem 17 *Let G be the value graph of a sequence of variables $X = x_1, x_2, \ldots, x_n$ and let M be a maximum-size matching in G. The constraint* `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{var}})$ *is hyper-arc consistent if and only if either*

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 of size $|M|$.*

Proof. We can assign $|M|$ variables to a different value. Thus we need to change the value of $n-|M|$ variables, i.e. $\mu_{\text{var}} = n-|M|$. 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. If $n - |M| = \max D_z$ only those edges that belong to a matching of size $|M|$ belong to a solution. □

The constraint `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{var}})$ can be made hyper-arc consistent similar to the approach in Section 4.4. First we compute a maximum-size matching M in the value graph G in $O(m\sqrt{n})$ time, where $m = \sum_{i=1}^{n} |D_i|$. 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-size matching. Here we apply Theorem 13, 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 4.4. Note that in this case also vertices in X may be M-free. Finally, we update $\min D_z = n - |M|$ if $\min D_z < n - |M|$. Then, by Theorem 17, the `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{var}})$ is hyper-arc consistent.

5.3.2 Decomposition-Based Violation Measure

Next we consider the decomposition-based violation measure. The results in this section are due to (van Hoeve, 2004).

Construct the directed graph $G_{\text{soft}} = (V, A)$ with

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

where $X = \{x_1, x_2, \ldots, x_n\}$ and

$$\begin{aligned} A_X &= \{(x_i, d) \mid d \in D_i\}, \\ A_s &= \{(s, x_i) \mid i = 1, \ldots, n\}, \\ A_t &= \{(d, t) \mid d \in D_i, i = 1, \ldots, n\}. \end{aligned}$$

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 way.

To each arc $a \in A$, we assign a capacity $c(a) = 1$ and a cost $w(a)$. If $a \in A_s \cup A_X$, then $w(a) = 0$. If $a \in A_t$, so $a = (d, t)_i$ for some $d \in D_X$ and integer i, the value of $w(a) = i$.

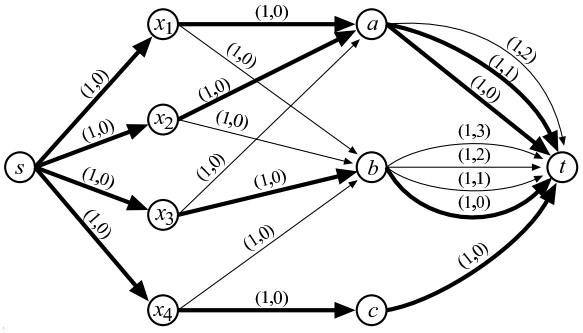


Figure 5: Graph G_{soft} for the soft `soft_alldifferent` constraint of Example 14. For each arc a, $(c(a), w(a))$ is given. Bold arcs indicate an optimal $s-t$ flow with cost 1.

Example 15 *Consider again the problem of Example 14. In Figure 5, the graph G_{soft} for the* `soft_alldifferent` *constraint is depicted. For each arc a, $(c(a), w(a))$ is given. The bold arcs in the graph denote a minimum-weight $s-t$ flow with weight 1. It corresponds to a solution to the COP, i.e. $x_1 = a, x_2 = a, x_3 = b, x_4 = c$ and $z = 1$.*

Theorem 18 *The constraint* `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{dec}})$ *is hyper-arc consistent if and only if*

i) for every arc $a \in A_X$ there exists an $s-t$ flow f under c in G_{soft} with $f(a) = 1$, $\text{value}(f) = n$ *and* $\text{weight}(f) \leq \max D_z$*, and*

ii) $\min D_z \geq \text{weight}(f)$ *for a minimum-weight $s-t$ flow f in G_{soft}.*

Proof. Similar to the proof of Theorem 16. The weights on the arcs in A_t are chosen such that the weight of a minimum-cost flow is exactly μ_{dec}. Namely, the first arc entering a value $d \in D_X$ causes no violation and chooses outgoing arc with weight 0. The k-th arc that enters d causes $k-1$ violations and chooses outgoing arc with weight $k-1$. □

The constraint `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{dec}})$ can be made hyper-arc consistent similar to the approach in Section 5.2. We first compute a minimum-cost flow f in G_{soft}. We do this by computing n shortest $s-t$ paths in the residual graph. Because there are only weights on arcs in A_t, each shortest path takes $O(m)$ time to compute; see (van Hoeve, 2004). Hence we can compute f in $O(nm)$ time. If $\text{weight}(f) > \max D_z$, the constraint is inconsistent.

To identify the arcs $a = (x_i, j) \in A_X$ that belong to a flow g with $\text{value}(g) = n$ and $\text{weight}(g) \leq \max D_z$, we apply Theorem 3. Thus, we search for a shortest $j - x_i$ path in $(G_{\text{soft}})_f$ that together with a forms a directed circuit C. We can compute all such shortest paths in $O(m)$ time because there are only weights on arcs in A_t. Then we update $D_i = D_i \setminus \{j\}$ if $\text{weight}(f) + \text{weight}(C) > \max D_z$. Finally, we update $\min D_z = \text{weight}(f)$ if $\min D_z < \text{weight}(f)$. Then, by Theorem 18, the `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{dec}})$ is hyper-arc consistent.

6. The Alldifferent Polytope

One of the cornerstones of Operations Research is the field of integer linear programming; see (Schrijver, 1986) and (Nemhauser & Wolsey, 1988). An integer linear programming model consists of integer variables, a set of linear constraints (inequalities or equations) and a linear objective function to be optimized. An integer linear program can be written as

$$\max \{cx \mid Ax \leq b, x \text{ integral}\} \tag{9}$$

for rational matrix A, rational vectors b and c and variable vector x of appropriate size. The continuous relaxation of (9) is

$$\max \{cx \mid Ax \leq b\}. \tag{10}$$

In case we are dealing only with continuous variables, like in program (10), the linear program can be solved to optimality in polynomial time; see (Schrijver, 1986). Otherwise, one needs to resort to a branch-and-bound framework, which is similar to the constraint programming search tree. In a branch-and-bound framework we solve at each node the relaxation (10). If all variables in the solution are integral, the original problem has been solved to optimality. Otherwise, we branch on a non-integral variable. One branch restricts the variable to be larger than its current solution value rounded up, the other branch restricts the variable to be smaller than its current solution value rounded down. Next we repeat the process on the generated subproblems. More sophisticated strategies exist, see for instance (Nemhauser & Wolsey, 1988), but are out of the scope of this paper.

A problem can usually be modelled as an integer linear program in more than one way. Each such model leads to a (possibly different) continuous relaxation. Ideally, one seeks for a model for which the relaxation is precisely the convex hull of all integer solutions. Namely, for such models the continuous relaxation has an integral, and hence optimal, solution. Formally, relaxation (10) defines a so-called polytope, i.e. the convex hull of a finite number of vectors, where the vectors are defined by the linear constraints $Ax \leq b$ of the problem. Our goal is to find a *tight* polytope, i.e. a polytope that describes the convex hull of all integer solutions to the problem.

Suppose the problem at hand contains the structure

$$\begin{array}{l} \texttt{alldifferent}(x_1, x_2, \ldots, x_n) \\ x_1, x_2, \ldots, x_n \in \{d_1, d_2, \ldots, d_n\} \end{array} \tag{11}$$

for which we would like to find a tight linear description. We assume that the domain values $d_1, d_2, \ldots, d_n$ are numerical values with $d_1 \leq d_2 \leq \cdots \leq d_n$. A convex hull representation of (11) was given by (Williams & Yan, 2001). We follow here the description and the proof of (Hooker, 2000, p. 232–233). The idea is that the sum of any k variables must be at least $d_1 + \cdots + d_k$.

Theorem 19 *Let $d_1, d_2, \ldots, d_n$ be any sequence of numerical values with $d_1 \leq d_2 \leq \cdots \leq d_n$. Then the convex hull of all vectors x that satisfy* `alldifferent`$(x_1, x_2, \ldots, x_n)$ *where*

$x_i \in \{d_1, d_2, \ldots, d_n\}$ *is described by*

$$\sum_{j=1}^{n} y_j = \sum_{j=1}^{n} d_j, \tag{12}$$

$$\sum_{j\in J} y_j \geq \sum_{j=1}^{|J|} d_j, \quad \textit{for all } J \subset \{1,\ldots,n\} \textit{ with } |J| < n. \tag{13}$$

for $y \in \mathbb{R}^n$.

Proof. It suffices to show that any valid inequality $a^{\mathsf{T}} y \geq \alpha$ can be obtained as a linear combination of (12) and (13) in which the coefficients of (13) are nonnegative. Assume, without loss of generality, that $a_{i_1} \geq a_{i_2} \geq \cdots \geq a_{i_n}$ for some permutation $i_1, \ldots, i_n$ of the indices. Then because $a^{\mathsf{T}} y \geq \alpha$ is valid, one can set $y_k = d_j$ if $i_j = k$, so that

$$\sum_{j=1}^{n} a_{i_j} d_j \geq \alpha. \tag{14}$$

From (12) and (13),

$$\sum_{j=1}^{n} y_{i_j} = d_1 + \cdots + d_n, \tag{15}$$

$$\sum_{j=1}^{k} y_{i_j} \geq d_1 + \cdots + d_k, \quad k = 1, \ldots, n-1. \tag{16}$$

Consider a linear combination in which each inequality of 16 has coefficient $a_{i_k} - a_{i_{k+1}}$, and (15) has coefficient a_{i_n}. The result is

$$\sum_{j=1}^{n} a_{i_j} y_{i_j} \geq \sum_{j=1}^{n} a_{i_j} d_j \overset{(14)}{\geq} \alpha$$

and the theorem follows. □

Example 16 *The convex hull of all vectors* x *that satisfy*

$$\begin{array}{l} \texttt{alldifferent}(x_1, x_2, x_3) \\ x_1, x_2, x_3 \in \{7, 11, 13\} \end{array}$$

can be given by

$$\begin{array}{l} y_1 + y_2 + y_3 = 31, \\ y_1 + y_2 \geq 18, \\ y_1 + y_3 \geq 18, \\ y_2 + y_3 \geq 18, \\ y_j \geq 7, \textit{ for } j = 1, 2, 3, \end{array}$$

and is depicted in Figure 6.

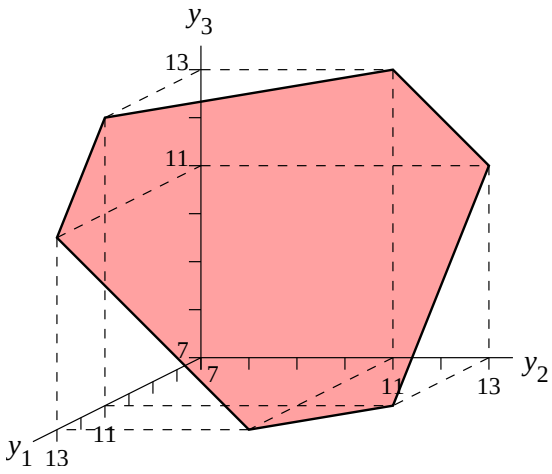


Figure 6: The polytope of Example 16.

Unfortunately, the number of inequalities to describe the convex hull grows exponentially with the number of variables. In practice it may therefore be profitable to generate these inequalities only when they are violated by the current relaxation.

Further work on the description of the `alldifferent` constraint in integer linear programming has been done by (Lee, 2002). That work proposes a representation that uses a binary encoding of the solution set. Further, (Appa, Magos, & Mourtos, 2004) present linear programming relaxations based upon multiple `alldifferent` constraints.

Acknowledgements

Many people have contributed helpful comments and suggestions to improve this paper. I wish to thank in particular Maarten van Emden, Krzysztof Apt, Sebastian Brand, Irit Katriel and Jean-Charles Régin.

References

Ahuja, R., Magnanti, T., & Orlin, J. (1993). *Network Flows*. Prentice Hall.

Appa, G., Magos, D., & Mourtos, I. (2004). LP Relaxations of Multiple all_different Predicates. In Régin, J.-C., & Rueher, M. (Eds.), *Proceedings of the First International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2004)*, Vol. 3011 of *LNCS*, pp. 364–369. Springer.

Apt, K. (1999). The essence of constraint propagation. *Theoretical Computer Science*, *221*(1–2), 179–210.

Apt, K. (2003). *Principles of Constraint Programming*. Cambridge University Press.

Barnier, N., & Brisset, P. (2002). Graph Coloring for Air Traffic Flow Management. In *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR 2002)*, pp. 133–147.

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

Beldiceanu, N., Carlsson, M., & Rampon, J.-X. (2005). Global Constraint Catalog. Tech. rep. T2005-08, SICS.

Beldiceanu, N., & Contejean, E. (1994). Introducing Global Constraints in CHIP. *Mathematical and Computer Modelling*, *20*(12), 97–123.

Berge, C. (1957). Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the United States of America*, *43*, 842–844.

Caseau, Y., & Laburthe, F. (1997). Solving Various Weighted Matching Problems with Constraints. In Smolka, G. (Ed.), *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'97)*, Vol. 1330 of *LNCS*, pp. 17–31. Springer.

Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

Dechter, R., & van Beek, P. (1997). Local and global relational consistency. *Theoretical Computer Science*, *173*, 283–308.

Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., & Berthier, F. (1988). The Constraint Logic Programming Language CHIP. In ICOT (Ed.), *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pp. 693–702. Springer.

Easterfield, T. (1946). A combinatorial algorithm. *Journal of the London Mathematical Society*, *21*, 219–226.

Edmonds, J. (1965). Paths, trees and flowers. *Canadian Journal of Mathematics*, *17*, 449–467.

Falkowski, B.-J., & Schmitz, L. (1986). A note on the queens' problem. *Information Processing Letters*, *23*, 39–46.

Fischetti, M., & Toth, P. (1989). An additive bounding procedure for combinatorial optimization problems. *Operations Research*, *37*, 319–328.

Focacci, F., Lodi, A., & Milano, M. (1999). Integration of CP and OR methods for matching problems. In *Proceedings of the First International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'99)*.

Gent, I., Stergiou, K., & Walsh, T. (2000). Decomposable Constraints. *Artificial Intelligence*, *123*(1–2), 133–156.

Gerards, A. (1995). Matching. In Ball, M., Magnanti, T., Monma, C., & Nemhauser, G. (Eds.), *Network Models*, Vol. 7 of *Handbooks in Operations Research and Management Science*, pp. 135–224. Elsevier Science.

Gomes, C., & Shmoys, D. (2002). Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. In *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*.

Grönkvist, M. (2004). A Constraint Programming Model for Tail Assignment. In Régin, J.-C., & Rueher, M. (Eds.), *Proceedings of the First International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2004)*, Vol. 3011 of *LNCS*, pp. 142–156. Springer.

Hall, P. (1935). On representatives of subsets. *Journal of the London Mathematical Society*, *10*, 26–30.

Harvey, W., & Ginsberg, M. (1995). Limited Discrepancy Search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, Vol. 1, pp. 607–615.

Henz, M., Müller, T., & Thiel, S. (2004). Global constraints for round robin tournament scheduling. *European Journal of Operational Research*, *153*(1), 92–101.

Hooker, J. (2000). *Logic-Based Methods for Optimization - Combining Optimization and Constraint Satisfaction*. Wiley.

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

König, D. (1931). Graphok és matrixok. *Matematikai és Fizikai Lapok*, *38*, 116–119.

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

Leconte, M. (1996). A bounds-based reduction scheme for constraints of difference. In *Proceedings of the Second International Workshop on Constraint-based Reasoning (Constraint-96), Key West, Florida*.

Lee, J. (2002). All-Different Polytopes. *Journal of Combinatorial Optimization*, *6*(3), 335–352.

Lodi, A., Milano, M., & Rousseau, L.-M. (2003). Discrepancy-Based Additive Bounding for the Alldifferent Constraint. In Rossi, F. (Ed.), *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)*, Vol. 2833 of *LNCS*, pp. 510–524. Springer.

Lopez-Ortiz, A., Quimper, C.-G., Tromp, J., & van Beek, P. (2003). A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pp. 245–250. Morgan Kaufmann.

Lovász, L., & Plummer, M. D. (1986). *Matching Theory*. North-Holland, Amsterdam.

Mehlhorn, K., & Thiel, S. (2000). Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In Dechter, R. (Ed.), *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP 2000)*, Vol. 1894 of *LNCS*, pp. 306–319. Springer.

Micali, S., & Vazirani, V. (1980). An $O(\sqrt{|v||E|})$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, pp. 17–27. IEEE.

Mohr, R., & Masini, G. (1988). Good Old Discrete Relaxation. In *European Conference on Artificial Intelligence (ECAI)*, pp. 651–656.

Mulder, H. (1992). Julius petersen's theory of regular graphs. *Discrete Mathematics*, *100*, 157–175.

Nemhauser, G., & Wolsey, L. (1988). *Integer and Combinatorial Optimization*. Wiley.

Older, W., Swinkels, G., & van Emden, M. (1995). Getting to the Real Problem: Experience with BNR Prolog in OR. In *Proceedings of the Third International Conference on the Practical Applications of Prolog (PAP'95)*. Alinmead Software Ltd.

Petersen, J. (1891). Die Theorie der regulären graphs. *Acta Mathematica*, *15*, 193–220.

Petit, T., Régin, J.-C., & Bessière, C. (2001). Specific Filtering Algorithms for Over-Constrained Problems. In Walsh, T. (Ed.), *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP 2001)*, Vol. 2239 of *LNCS*, pp. 451–463. Springer.

Puget, J.-F. (1998). 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)*, pp. 359–366. AAAI Press / The MIT Press.

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

Régin, J.-C. (1996). 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)*, Vol. 1, pp. 209–215. AAAI Press / The MIT Press.

Régin, J.-C. (1999a). Arc Consistency for Global Cardinality Constraints with Costs. In Jaffar, J. (Ed.), *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP'99)*, Vol. 1713 of *LNCS*, pp. 390–404. Springer.

Régin, J.-C. (1999b). The symmetric alldiff constraint. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pp. 420–425.

Régin, J.-C. (2002). Cost-Based Arc Consistency for Global Cardinality Constraints. *Constraints*, *7*, 387–405.

Régin, J.-C. (2003). Global Constraints and Filtering Algorithms. In Milano, M. (Ed.), *Constraint and Integer Programming - Toward a Unified Methodology*, Vol. 27 of *Operations Research/Computer Science Interfaces*, chap. 4. Kluwer Academic Publishers.

Régin, J.-C., Petit, T., Bessière, C., & Puget, J.-F. (2000). An Original Constraint Based Approach for Solving over Constrained Problems. In Dechter, R. (Ed.), *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP 2000)*, Vol. 1894 of *LNCS*, pp. 543–548. Springer.

Rossi, F., Petrie, C., & Dhar, V. (1990). On the equivalence of constraint satisfaction problems. In *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI)*, pp. 550–556.

Schrijver, A. (1986). *Theory of Linear and Integer Programming*. Wiley.

Schrijver, A. (2003). *Combinatorial Optimization - Polyhedra and Efficiency*. Springer.

Schulte, C., & Stuckey, P. (2001). When Do Bounds and Domain Propagation Lead to the Same Search Space. In Søndergaard, H. (Ed.), *Proceedings of the Third International Conference on Principles and Practice of Declarative Programming*, pp. 115–126. ACM Press.

Sellmann, M. (2002). An Arc-Consistency Algorithm for the Minimum Weight All Different Constraint. In Hentenryck, P. V. (Ed.), *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)*, Vol. 2470 of *LNCS*, pp. 744–749. Springer.

Stergiou, K., & Walsh, T. (1999). The difference all-difference makes. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pp. 414–419.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, *1*, 146–160.

Tsang, E., Ford, J., Mills, P., Bradwell, R., Williams, R., & Scott, P. (2004). ZDC-Rostering: A Personnel Scheduling System Based On Constraint Programming. Tech. rep. 406, University of Essex, Colchester, UK.

van der Waerden, B. (1927). Ein Satz über Klasseneinteilungen von endlichen Mengen. *Abhandlungen aus dem mathematischen Seminar der Hamburgischen Universität*, *5*, 185–188.

Van Hentenryck, P. (1989). *Constraint Satisfaction in Logic Programming*. Logic Programming Series. The MIT Press, Cambridge, MA.

van Hoeve, W.-J. (2001). The Alldifferent Constraint: A Survey. In *Proceedings of the Sixth Annual Workshop of the ERCIM Working Group on Constraints*. `http://www.arxiv.org/html/cs/0110012`.

van Hoeve, W.-J. (2004). A Hyper-Arc Consistency Algorithm for the Soft Alldifferent Constraint. In Wallace, M. (Ed.), *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, Vol. 3258 of *LNCS*, pp. 679–689. Springer.

Wallace, M., Novello, S., & Schimpf, J. (1997). ECLiPSe: A platform for constraint logic programming. Tech. rep., IC-Parc, Imperial College, London.

Williams, H., & Yan, H. (2001). Representations of the all_different Predicate of Constraint Satisfaction in Integer Programming. *INFORMS Journal on Computing*, *13*(2), 96–103.

Zhou, J. (1997). A permutation-based approach for solving the job-shop problem. *Constraints*, *2*(2), 185–213.