Multiconsistency and Robustness with Global Constraints

Khaled Elbassioni and Irit Katriel

Max-Planck-Institut für Informatik, Saarbrücken, Germany
{elbassio, irit}@mpi-sb.mpg.de

Abstract. We propose a natural generalization of arc-consistency, which we call multiconsistency: A value v in the domain of a variable x is k-multiconsistent with respect to a constraint C if there are at least k solutions to C in which x is assigned the value v. We present algorithms that determine which variable-value pairs are k-multiconsistent with respect to several well known global constraints. In addition, we show that finding super solutions is strictly harder than finding arbitrary solutions and suggest multiconsistency as an alternative way to search for robust solutions.

1 Introduction

A value v in the domain of a variable x is *consistent* with respect to the constraint C if there is at least one solution to the constraint in which x is assigned the value v. Identifying values which are not consistent is a fundamental task for a constraint solver; it is crucial for reducing the exponential-size search space that would otherwise need to be explored.

In this paper we generalize the notion of consistency: A value v in the domain of the variable x is k-multiconsistent with respect to a constraint C if there are at least k solutions to C in which x is assigned the value v. Intuitively, a value that appears in many solutions is a *"useful"* value. Knowing which values are useful can be helpful in several ways. For example, usefulness can be used as a heuristic while searching for a solution: While it is not guaranteed, it seems reasonable to assume that if the constraint program has a solution s, then the more useful a variable-value pair is with respect to individual constraints that are defined on it, the more likely it is to be used in s. This implies that it makes sense to regard the usefulness of the values as a heuristic that guides the search.

Another possible application is in the search for robust solutions, i.e., solutions that can be repaired if a small change occurs. In their recent paper on the topic, Hebrard et al. [5] give the example of a schedule: A robust schedule does not collapse if one job takes slightly longer to execute than planned. Rather, the schedule changes locally and the overall makespan changes little if at all. In the same paper, they define the notion of *super solutions*, which is a generalization of super models in propositional satisfiability. An (a, b)-super solution is a solution

R. Barták and M. Milano (Eds.): CPAIOR 2005, LNCS 3524, pp. 168–182, 2005.

such that if a variables lose their values, a new solution can be constructed by assigning new values to these a variables and changing the values of at most b other variables.

This is a very strong guarantee of robustness, and Hebrard et al. note that it is quite rare to have a solution for which all of the variables can be repaired. Therefore, they formulate the optimization problem of seeking the "most robust" solution, i.e., the solution that maximizes the number of repairable variables. They then study several approaches for finding super solutions, and the super MAC search algorithm that they have developed for this purpose emerged as the most promising. As for complexity, Hebrard et al. show that it is, in general, NP-hard to find an (a, b)-super solution for a constraint program, for any fixed a. They show this by proving that any constraint program P can be transformed in polynomial time into a second constraint program P' such that P has a solution iff P' has an (a, b)-super solution. Thus, finding a super solution is as hard as finding an arbitrary solution, which is NP-hard. We will show that, in fact, finding a super solution is strictly harder than finding an arbitrary solution. In particular, we will prove that it is NP-hard to determine whether an *AllDifferent* constraint has a $(1, 0)$-super solution. Finding an arbitrary solution to an *AllDifferent* constraint can be done in polynomial time [6, 11].

On the other hand, we will show that there are efficient algorithms to determine which values are k-multiconsistent with respect to *AllDifferent* and other global constraints. This information can easily be used to search for a k-multiconsistent solution, i.e., a solution that uses only k-multiconsistent assignments of values to the variables, or, if no such solution exists, a solution that maximizes the number of k-multiconsistent values. It is not guaranteed that a k-multiconsistent solution can be easily repaired if some of the variables lose their values. We are certainly not guaranteed that a local change will give a new solution, or even that another solution exists. However, with a k-multiconsistent solution, we do know that the remaining variables are assigned to values that were once considered "useful", and our purpose is to show that the computational price we need to pay for this knowledge is not very high in the case of the constraints that we consider. We therefore believe that it would be worthwhile to conduct experimental and theoretical research on the concept of multiconsistency and ways in which it can be applied.

Section 2 contains a formal definition of k-multiconsistency and other notions that will appear in the following sections. In Section 3 we show that it is NP-hard to determine whether an *AllDifferent* constraint has a $(1, 0)$-super solution. In Section 4 we describe an algorithm that computes k-multiconsistency for the *AllDifferent* constraint when the number of variables is equal to the number of values. In Section 5 we show that this basic algorithm can be generalized for the general *AllDifferent* constraint and for other global constraints. Finally, in Section 6 we list some open problems that arise from our work.

2 Multiconsistency and Preliminaries

2.1 Multiconsistency

The formal definition of multiconsistency appears below. It is a straightforward generalization of the definition of arc-consistency [1] as appears, e.g., in [1].

Definition 1. *Let C be a constraint on the variables $x_1, \ldots, x_\ell$ with respective domains $D(x_1), \ldots, D(x_\ell)$ and let $S \subseteq D(x_1) \times \ldots \times D(x_\ell)$ be the set of solutions to C. Then a variable-value pair (x_j, v_i) is* k-multiconsistent *with respect to C if there are at least k tuples in S in which the jth component is v_i.*

The rest of the paper deals with multiconsistency of individual values. However, for completeness, we include the definition of a multiconsistent solution.

Definition 2. *Let C be a constraint on the variables $x_1, \ldots, x_\ell$ with respective domains $D(x_1), \ldots, D(x_\ell)$. Let s be a solution to C and for all $1 \leq j \leq \ell$, let $s(x_j)$ be the value assigned by s to x_j. s is a* k-multiconsistent solution *if for every $1 \leq j \leq \ell$, $(x_j, s(x_j))$ is k-multiconsistent with respect to C.*

2.2 A Few Global Constraints

The global constraints that we will consider in this paper are:

- The *AllDifferent*$(x_1, \ldots, x_n)$ [8, 9, 10, 13, 15] constraint is specified on n assignment variables. A solution s assigns each variable x_i a value $s(x_i) \in D(x_i)$ such that for any $1 \leq i < j \leq n$, $s(x_i) \neq s(x_j)$.
- The *Global Cardinality Constraint* $GCC(x_1, \cdots, x_n, c_{v_1}, \cdots, c_{v_{n'}})$ [7, 11, 12, 14] is specified on n assignment variables $x_1, \ldots, x_n$ and n' count variables $c_{v_1}, \ldots, c_{v_{n'}}$. A solution s assigns each assignment variable x_j a value $s(x_j) \in D(x_j) \subseteq D = \{v_1, \cdots, v_{n'}\}$ and assigns each count variable c_{v_i} a value $s(c_{v_i}) \in D(c_{v_i})$ such that each value v_i is assigned to exactly $s(c_{v_i})$ assignment variables. We will assume that the domains of the count variables are intervals, each of which is specified by a lower and upper bound, i.e., $D(c_{v_i}) = [L_i, U_i]$.
- The *Same*$(X = \{x_1, \ldots, x_n\}, Z = \{z_1, \ldots, z_n\})$ [2] constraint is defined on two sets X and Z of distinct variables such that $|X| = |Z|$. A solution s assigns each variable $v \in X \cup Z$ a value $s(v) \in D(v)$ such that the multiset of values assigned to the variables of X is identical to the multiset of values assigned to the variables of Z.

2.3 Matchings

The solutions to the global constraints we consider in this paper will be represented as subsets of the edges of a graph that models the constraint. The following terms will be used:

[1] Some texts refer to *hyper-arc-consistency* when speaking of global constraints and reserve the term *arc-consistency* for the special case of binary constraints.

Definition 3. *Given a graph $G = (V, E)$, a subset M of E is a* matching *if every node is incident to at most one edge from M. It is a* perfect matching *if every node is incident to exactly one edge from M.*

In the case of a bipartite graph we have the following definition:

Definition 4. *Let $G = (V, E)$ be a bipartite graph with $V = X \cup Y$ such that X is the set of nodes on one side, Y is the set of nodes on the other side, and $|X| \leq |Y|$. Then a subset M of E is called an* X-perfect matching *if every node in X is incident to exactly one edge from M, and every node Y is incident to at most one edge from M.*

We turn to the more general case where each node of the graph has a capacity requirement that specifies how many of the edges incident to it should be included in the matching.

Definition 5. *Let $G = (V, E, C)$ be a capacitate graph, where C is a function that maps every node $v \in V$ to an interval $C(v) = [L_v, U_v]$. We call $C(v)$ the* capacity requirement *of v. A* generalized matching *[7] in G is a subset M of its edges such that each node $v \in V$ is incident to at least L_v and at most U_v edges in M.*

Alternating cycles and paths will appear as an important tool in our algorithms.

Definition 6. *Let G be a graph and let M be a subset of its edges. An* alternating path (cycle) *in G with respect to M is a simple path (cycle) in G where each edge belonging to M in the path (cycle) (except the last in the case of a path), is followed by an edge which is not in M, and vice versa.*

2.4 Flows

When the graph is directed and has capacities on the edges, and not on the nodes, we can view it as a flow network.

Definition 7. *Given a directed graph $\vec{G} = (V, \vec{E})$ with lower and upper capacities l_e, u_e for each arc $e \in \vec{E}$, a* feasible flow *in $\vec{G}$ is a function $f : E \to \mathbb{R}$ such that*

1. **Flow conservation:** *For each node $v \in V$,*

$$\sum_{\{u|(v,u)\in\vec{E}\}} f(v,u) = \sum_{\{w|(w,v)\in\vec{E}\}} f(w,v).$$

2. **Capacities:** *For each $e \in \vec{E}$, $l_e \leq f(e) \leq u_e$.*

An integral *feasible flow is a feasible flow such that for all $e \in \vec{E}$, $f(e)$ is an integer.*

The residual graph, defined below, appears in one of our algorithms:

Definition 8. *Given a directed graph $\vec{G} = (V, \vec{E})$ with lower and upper capacities l_e, u_e for each arc $e \in \vec{E}$ and a flow f in it, the* residual graph $\vec{G}_f$ *is defined as follows: For each $e = (u, v) \in \vec{E}$, (1) if $f(e) < u_e$ then the arc (u, v) appears in $\vec{G}_f$ with capacity $[0, u_e - f(e)]$. (2) if $f(e) > l_e$ then the arc (v, u) appears in $\vec{G}_f$ with capacity $[0, f(e) - l_e]$.*

It is not hard to show that for a directed graph $\vec{G}$, if f is a feasible flow in $\vec{G}$ and f' is a feasible flow in $\vec{G}_f$, then $f'' = f \oplus f'$ is a feasible flow in $\vec{G}$, where the operation $\oplus$ is defined as follows: If e has the same direction in $\vec{G}$ and $\vec{G}_f$ then $f(e) \oplus f'(e) = f(e) + f'(e)$ and otherwise, $f(e) \oplus f'(e) = f(e) - f'(e)$. Thus, the residual graph enables us to transform one feasible flow into another by finding a positive-weight cycle.

2.5 Enumeration Algorithms

Generally speaking, given a property $\pi : 2^U \mapsto \{0, 1\}$ defined over all subsets of a ground set U, an enumeration algorithm for π is a procedure that lists, one by one, all subsets Y of U satisfying π, i.e., for which $\pi(Y) = 1$. For example, assume that $U = E$ is the edge set of a bipartite graph $G = (V, E)$ and $\pi(Y)$ is the property that the edge set $Y \subseteq E$ is a perfect matching. Since, in general, the size of the output of an enumeration algorithm (in our case, the perfect matchings) is typically exponential in the size of the input (in our case, the size of the graph $|V| + |E|$), it is common to measure the efficiency of the algorithm in terms of the combined size of the input and output. Such an algorithm is said to be *incrementally polynomial* if, after generating a subset $\mathcal{X}$ of elements (satisfying π), the time to generate a new element is polynomial in both $|\mathcal{X}|$ and the size of the input. A stronger requirement on an enumeration algorithm is to run with *polynomial delay*, in which case, the time to generate a new element is polynomial only in the size of the input, i.e., does not depend on how many elements have been generated so far. As we shall see below, the enumeration algorithms we use are of the latter type.

3 NP-Hardness of Finding a $(1, 0)$-Super Solution to the *AllDifferent* constraint

In this section we show that it is NP-hard to determine whether an *AllDifferent* constraint has a $(1, 0)$-super solution. Since it takes $O(n^{3/2}n')$ time to determine whether it has an arbitrary solution [6, 11], this implies that finding super solutions is strictly harder than finding arbitrary solutions.

Theorem 1. *Given n variables, $x_1, \ldots, x_n$, with respective domains $D(x_1), \ldots, D(x_n)$, it is NP-hard to determine whether there exists a $(1, 0)$-super solution for the AllDifferent$(x_1, \ldots, x_n)$ constraint, even if $|D(x_i)| \leq 4$ for all $1 \leq i \leq n$.*

Proof. We use a polynomial-time transformation from the 3SAT problem: Given a conjunctive normal form formula $\phi(y_1, \ldots, y_N) = C_1 \wedge \ldots \wedge C_m$, where each C_j is a disjunction of 3 literals in $\{y_1, \bar{y}_1, \ldots, y_N, \bar{y}_N\}$, determine whether there exists a truth assignment to $y_1, \ldots, y_N$ which satisfies all clauses of $\phi(y_1, \ldots, y_N)$.

We show that we can construct an instance of *AllDifferent* that has a $(1, 0)$-super solution iff $\phi(y_1, \ldots, y_N)$ is satisfiable. Let $n = N + m$, and $x_1, \ldots, x_n$ be n variables, the union of whose domains is $D = \{a_1, b_1, \ldots, a_N, b_N, c_1, \ldots, c_m\}$. The domains of specific variables are defined as follows. For $i = 1, \ldots, N$, let $D(x_i) = \{a_i, b_i\}$. For $j = 1, \ldots, m$, let $D(x_{j+N}) = \{c_j\} \cup \{a_i : i = 1, \ldots, N, \text{and } y_i \in C_j\} \cup \{b_i : i = 1, \ldots, N, \text{and } \bar{y}_i \in C_j\}$ (see Figure 1).

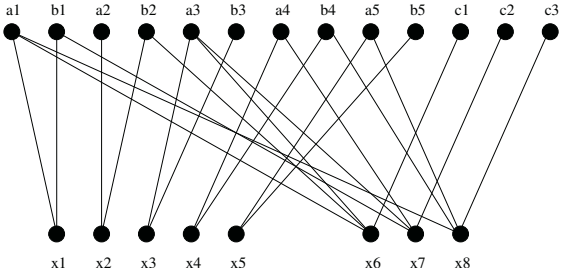

Fig. 1. The *AllDifferent* instance generated from the 3SAT formula $(y_1 \vee \bar{y}_2 \vee y_3) \wedge (\bar{y}_1 \vee y_3 \vee y_4) \wedge (y_1 \vee \bar{y}_4 \vee y_5)$

We claim that there is a $(1, 0)$-super solution to the *AllDifferent*$(x_1, \ldots, x_n)$ constraint with the specified variable domains if and only if the formula ϕ is satisfiable. Indeed, given a satisfying assignment σ for ϕ, we can construct a $(1, 0)$-super solution to our constraint as follows. For $i = 1, \ldots, N$, x_i is assigned either the value b_i or the value a_i, depending, respectively, on whether y_i is assigned $True$ or $False$ by σ. For $j = 1, \ldots, m$, the variable x_{j+N} is assigned the value c_j. Clearly, each variable was assigned a different value so the *AllDifferent* constraint is satisfied. Furthermore, for each variable x_i, there exists a value $v \in D(x_i)$ that has not been assigned to any other variable. This is obvious for $i = 1, \ldots, N$, and follows, for $i = N + 1, \ldots, n$ from the fact that σ is satisfying, i.e. for each clause C_j, $j = 1, \ldots, m$, there is a literal in C_j that is assigned $True$ by σ. Conversely, given a $(1, 0)$-super solution to the *AllDifferent* constraint, we define a truth assignment σ to the Boolean variables $y_1, \ldots, y_N$, by setting $y_i = True$ if x_i is assigned the value b_i, and setting $y_i = False$ if x_i is assigned the value a_i. Note that, for $j = 1, \ldots, m$, each variable x_{j+N} is assigned the value c_j by the super solution since otherwise some variable x_i, $i \in \{1, \ldots, N\}$ would have all its domain values assigned to variables, contradicting the requirement of a $(1, 0)$-super solution. In particular, for each $j = 1, \ldots, m$, variable x_{j+N} must have at least one value in its domain that is also an unassigned value in the domain of some variable x_i, $i \in \{1, \ldots, N\}$. This implies that each clause C_j is satisfied by σ. □

4 Multiconsistency for a Restricted Case of the *AllDifferent* Constraint

In this section we consider the following problem: Given an integer k and an *AllDifferent*$(x_1, \ldots, x_n)$ constraint where the domains of $x_1, \ldots, x_n$ are all contained in the set $\{1, \ldots, n\}$, determine which values are k-multiconsistent with respect to the constraint[2].

It is common to represent the *AllDifferent* constraint by a bipartite graph G with a node for each variable on one side, a node for each value on the other side, and an edge between the node representing the variable x_j and the node representing the value v_i iff v_i is in the domain of x_j. Then, there is a one-to-one correspondence between the solutions to the constraint and the matchings of cardinality n in G. In the restricted case that we consider in this section, a matching of cardinality n is also a perfect matching.

The k-multiconsistency problem for the restricted *AllDifferent* constraint, then, is the following: Given a bipartite graph G with n nodes on each side and m edges, determine which edges of G belong to at least k perfect matchings.

The algorithm in Figure 2 uses a recursive reformulation of an algorithm by Fukuda and Matsui that enumerates the perfect matchings in a bipartite graph [4]. For each edge e in G, the algorithm attempts to enumerate k perfect matchings that contain e. If it fails, it determines that e is not k-multiconsistent. The algorithm uses the following operations on graphs.

Definition 9. *Let $e = (u, v)$ be an edge in G. Then $G - e$ is the graph obtained by removing the edge e from G and $G \backslash e$ is the graph obtained by removing from G the nodes u and v and all edges incident to them.*

Clearly, there are k perfect matchings in $G \backslash e$ iff there are k perfect matchings in G that contain the edge e. Hence, checking whether e is k-multiconsistent is equivalent to checking whether $G \backslash e$ contains k perfect matchings.

Let $T(n, n', m)$ be the time required to find a maximum cardinality matching in a bipartite graph with n nodes on one side, n' nodes on the other, and m edges. In the $n' = n$ case, the enumeration algorithm by Fukada and Matsui needs $T(n, n, m)$ time to find the first perfect matching and then $O(n + m)$ time to generate each additional perfect matching. We get that given a single perfect matching that contains the edge e, we can check in $O(k(m + n))$ time whether e is k-multiconsistent. Note that once we have a perfect matching M, we can find, in linear time, a perfect matching M' that contains a specified edge e which is not in M: All we need is an alternating cycle that contains e. Thus, the total running time required to check k-multiconsistency for all edges is $T(n, n, m) + O(mk(m + n))$ time.

[2] This restriction of the *AllDifferent* constraint is also equivalent to a special case of the *Sortedness* constraint [3, 9].

Enumerating k Perfect Matchings

The basic idea of Fukuda and Matsui's enumeration algorithm is the following: First, it finds two perfect matchings M and M' in the graph. Then, it selects an edge e which belongs to one but not the other. e is used to partition the problem into two subproblems: The first is to generate all perfect matchings that contain e and the second is to generate all perfect matchings that do not contain e. Clearly, the outputs of the two subproblems are disjoint.

The procedure *NextPerfectMatchings* shown in Figure 2 implements this algorithm, with the additional upper bound k on the number of perfect matchings that should be generated. It receives a graph G, a perfect matching M in G and an integer k that indicates how many more perfect matchings should be generated. If $k > 0$, the procedure searches for an alternating cycle and generates a new perfect matching M'. Then it selects an edge $e \in M' \setminus M$ and makes two recursive calls to itself: The first receives the graph $G - e$ and the matching M. It generates all matchings in G that do not contain the edge e. The second recursive call receives the graph $G \backslash e$ and the matching $M' \setminus \{e\}$. It generates the matchings in G that contain the edge e. The procedure returns the number of matchings it has generated, which is k if it was successful and an integer smaller than k otherwise.

5 Generalizations for Other Constraints

In this section we show that the basic algorithm described in Section 4 can be generalized for the (unrestricted) *AllDifferent*, *GCC* and *Same* constraints.

5.1 *AllDifferent*

The bipartite graph representing the *AllDifferent* constraint is defined similarly to that of the restricted *AllDifferent* constraint, with one difference: Instead of n nodes on each side, there are n variable nodes and n' value nodes, with $n' \geq n$. Of course, when $n' > n$ the graph does not contain any perfect matching. A solution to the constraint now corresponds to a matching that matches all of the variable nodes, i.e., an X-perfect matching where the set of variable nodes is denoted by X.

The algorithm of Figure 2 can be modified for this case as follows: Replace all references to perfect matchings by X-perfect matchings. Algorithmically, this means that a new matching can be generated from an existing matching in one of two ways: By an alternating cycle as in the previous section, or by an alternating path from a matched value node to an unmatched value node.

5.2 *GCC*

The graph with which we represent the *GCC* constraint is a *capacitated* graph, i.e., a bipartite graph which topologically looks like the graph used for *AllDifferent*, but which has a capacity associated with each node. The capacity

```
Procedure kMultiConsistency(G,k)
    (* Initialization and tests for trivial inputs: *)
    foreach edge e in G do kCons[e] ← TRUE
    if k ≤ 0 then return
    if there is no perfect matching in G then
        foreach edge e in G do kCons[e] ← FALSE;
        return
    end if
    M ← a perfect matching in G
    k ← k − 1

    (* The main loop: *)
    foreach edge e in G do
        M' ← a perfect matching in G which contains e
        k' ← NextPerfectMatchings(G\e, M' \ {e}, k)
        if k' < k then kCons[e] ← FALSE
    end for
end

Procedure NextPerfectMatchings(G, M, k)
    if k ≤ 0 then return 0
    else if there is an alternating cycle C in G then
        M' ← M ⊕ C
        k ← k − 1

        e ← an edge from M' \ M
        (* First recursive call: perfect matchings without e *)
        k' ← NextPerfectMatchings(G − e, M, k)
        (* Second recursive call: perfect matchings containing e *)
        k'' ← NextPerfectMatchings(G\e, M' \ {e}, k − k')
        return k' + k'' + 1
    else
        return 0
    endif
end
```

Fig. 2. k-multiconsistency for the restricted *AllDifferent* constraint

of a node v, denoted $C_v = [L_v, U_u]$, is an interval. With capacity $[1, 1]$ for each variable node and $[L_i, U_i]$ for the value node that corresponds to the value v_i, we get that there is a one-to-one correspondence between the generalized matchings in G and the solutions of the GCC. Note that the different generalized matchings in G do not, in general, have the same cardinality.

To modify the algorithm of Figure 2 for the GCC constraint, we generalize the $G\backslash e$ operation that Fukuda and Matsui use with uncapacitated graphs, to the case of capacitated graphs. For a capacitated graph G and an edge $e = (u, v)$ in G, $G\backslash e$ is the graph obtained by subtracting 1 from the lower and upper

capacities of each of u and v. Note that reducing the upper capacity of a node to 0 is equivalent to removing the node and all edges incident to it from the graph.

We also need to generalize the manner in which the algorithm searches for M'. Given the capacitated graph $G = (V, E)$ and a generalized matching M in it, another generalized matching can be found by searching for a directed cycle in the directed graph $\vec{G} = (\vec{V}, \vec{E})$ defined as follows: $\vec{V} = V \cup \{s\}$. For each edge $e = (x, v) \in E$ between a variable node x and a value node v, $\{x, v\} \in \vec{E}$ if $e \in M$ and $\{v, x\} \in \vec{E}$ otherwise. Finally, for each value node v, $\{v, s\} \in \vec{E}$ if v is incident to more than L_v edges in M and $\{s, v\} \in \vec{E}$ if v is incident to less than U_v edges in M [7, 14].

5.3 *Same*

The basic algorithm can also be modified to support the *Same* constraint, but in this case the changes are more substantial.

The $Same(X = \{x_1, \ldots, x_n\}, Z = \{z_1, \ldots, z_n\})$ constraint [2] is modelled by a graph with three sets of nodes: One set for the variables of X (called x-nodes), a second set for the variables of Z (called z-nodes) and a third set for the values (called y-nodes). For each variable $u \in X \cup Z$ and for each value v in the domain of u, there is an edge in the graph between the node that represents u and the node that represents v. Let M be a subset of the edges and let y be a y-node. We denote by $M_X(y)$ ($M_Z(y)$) the set of x-nodes (z-nodes) adjacent to y by edges in M. An edge between an x-node (z-node) and a y-node is called *an xy-edge* (*a yz-edge*). A *parity matching* in such a graph is a subset M of the edges such that every x-node or z-node is incident to exactly one edge from M and for every y-node y, $|M_X(y)| = |M_Z(y)|$ [2]. There is a one-to-one correspondence between the parity matchings and the solutions to the *Same* constraint.

In the previous cases, after removing an edge from the graph we remained with a subproblem of the same type as the original problem. However, with the *Same* constraint the situation is slightly different. Suppose that we wish to enumerate all parity matchings that contain the xy-edge $e = (x, y)$. Then the algorithm will explore the graph $G \backslash e$ for sets of edges which are *almost* parity matchings. More precisely, we are interested in subsets M such that (1) $|M_Z(y)| = |M_X(y)| + 1$, (2) $|M_X(y')| = |M_Z(y')|$ for all $y' \neq y$ and (3) every variable in $X \cup Z$ except for x is matched. Then, $M \cup \{e\}$ is a parity matching which contains e. Since the algorithm recursively removes edges from the graph, the desired difference between $|M_X(y)|$ and $|M_Z(y)|$ can change in each recursive step, and not necessarily for the same y-node every time.

To support such demands, the algorithm of Figure 3 associates an imbalance requirement $I(y)$ to each y-node y, which is equal to the desired value of $|M_Z(y)| - |M_X(y)|$. Initially, $I(y) = 0$ for all y. When the algorithm makes a recursive call, there are three cases:

1. The recursive call needs to enumerate all solutions in which an xy-edge $e = (x, y)$ is contained. Then e is removed from the graph along with all other edges incident to x, and $I(y)$ is incremented.

Table 1. Domains of the variables for our example

j	$D(x_j)$	$D(z_j)$
1	{1,2}	{2,3}
2	{3,4}	{4,5}
3	{4,5,6}	{4,5}

2. The symmetric case for a yz-edge $e = (y, z)$. e is removed from the graph along with all other edges incident to z, and $I(y)$ is decremented.
3. The recursive call needs to enumerate all solutions in which an edge $e = (v, y)$ is not contained, for some x-node (or z-node) v and y-node y. Then e is removed from the graph and $I(y)$ remains unchanged.

Definition 10. *Let $G = (X \cup Z, Y, E)$ be a bipartite graph with $|X| = |Z|$ and an integer $I(y)$ associated with every $y \in Y$. A* generalized parity matching *is a subset $M \subseteq E$ such that for all y, $|M_Z(y)| - |M_X(y)| = I(y)$ and each $v \in X \cup Z$, is incident to exactly one edge in M.*

The algorithm needs to find a generalized parity matching from scratch only once, when $I(y) = 0$ for all y. Since in this case a generalized parity matching is just a parity matching, there already exists an algorithm for this task [2], which is based on finding a flow in the following network: We direct the arcs from x-nodes to y-nodes and from y-nodes to z-nodes, and place a capacity of $[0, 1]$ on each of them. In addition, we add two nodes s and t to the graph, add an arc with capacity $[1, 1]$ from s to each x-node, an arc with capacity $[1, 1]$ from each z-node to t and an arc with capacity $[n, n]$ from t to s, where $n = |X|$. There is a one-to-one correspondence between the integral feasible flows in this network and the parity matchings in the graph. Figure 4 shows the network constructed for the following example: $|X| = |Z| = 3$, $|Y| = 6$ and the domains of the variables of $X \cup Z$ are as in Table 1. Figure 5 shows an integral feasible flow in this network.

It remains to show how, given a generalized parity matching M, we can determine in linear time whether another generalized parity matching M' exists in the graph. To do this, we show how to generalize the graph described above such that there is a one-to-one correspondence between integral feasible flows and *generalized* parity matchings. Then, we can use the standard flow theory technique of finding another integral feasible flow by searching for a cycle in the residual graph (see Section 2).

As shown in Figure 6, we add an additional node $\mathcal{I}$ and connect it to the y-nodes with arcs that enforce the imbalances: For each y such that $I(y) > 0$, we add the arc $\{\mathcal{I}, y\}$ with capacity $[I(y), I(y)]$ and for each y such that $I(y) < 0$, we add the arc $\{y, \mathcal{I}\}$ with capacity $[-I(y), -I(y)]$. Since in a feasible flow, the flow into each y-node is equal to the flow out of this y-node, the required imbalances must be respected.

```
Procedure kMultiConsistencySame(G,k)
    (* Initialization and tests for trivial inputs: *)
    foreach edge e in G do kCons[e] ← TRUE
    foreach value node y do I(y) ← 0
    if k ≤ 0 then return
    if there is no parity matching in G then
        foreach edge e in G do kCons[e] ← FALSE;
        return
    end if
    M ← a parity matching in G
    k ← k − 1

    (* The main loop: *)
    foreach edge e = (v, y) in G do
        M' ← a parity matching in G which contains e
        if e is an xy-edge then I(y) ← 1 else I(y) ← −1
        k' ← NextParityMatchings(G\e, I, M' \ {e}, k)
        if k' < k then kCons[e] ← FALSE
        I(y) ← 0
    end for
end

Procedure NextParityMatchings(G, I, M, k)
    if k ≤ 0 then return 0
    else if there is another generalized parity matching M' in G then
        k ← k − 1

        e = (v, y) ← an edge from M' \ M
        (* First recursive call: matchings without e *)
        k' ← NextParityMatchings(G − e, I, M, k)

        (* Second recursive call: matchings containing e *)
        (* Update I(y) *)
        if e is an xy-edge then I(y) ← I(y) + 1
        else I(y) ← I(y) − 1
        k'' ← NextParityMatchings(G\e, I, M' \ {e}, k − k')
        (* Restore the previous I(y) *)
        if e is an xy-edge then I(y) ← I(y) − 1
        else I(y) ← I(y) + 1

        return k' + k'' + 1
    else
        return 0
    endif
end
```

Fig. 3. k-multiconsistency for the *Same* constraint

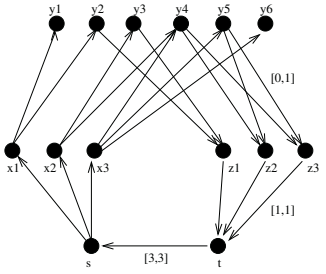

Fig. 4. The directed network for the example in Table 1

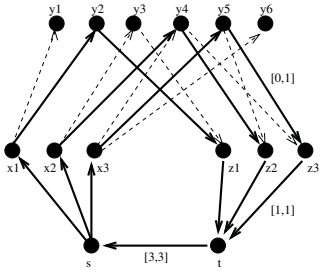

Fig. 5. A feasible flow in the graph of Figure 4

6 Discussion and Future Directions

In this paper, we have defined multiconsistency in the natural way and argued that the term corresponds to the intuitive notion of a "useful" value. Therefore, we believe that determining which values are k-multiconsistent with respect to a global constraint can be a component of reasonable heuristics for finding a solution to a constraint program, or for preferring solutions that can be expected to be more robust. In the realm of the search for robust solutions, we noted that the super solutions as defined by Hebrard et al. [5] seem to offer a better guarantee of robustness than k-multiconsistent solutions. However, we show that while it is NP-hard to determine whether an *AllDifferent* constraint has a $(1, 0)$-super solution, computing k-multiconsistency for the *AllDifferent*, *GCC* and *Same* constraints can be performed in time $T(n, n', m)+O(mk(m+n))$, where $T(n, n', m)$ is the time required to find a single solution, and is upper bounded by $O(n^{3/2}n')$ for *AllDifferent* and *GCC* [6, 11] and to $O(n^2n')$ for *Same* [2]. The complexity of computing arc-consistency (which we can now call 1-multiconsistency) for these constraints is $O(T(n, n', m))$. Thus, while there is a computational cost for k-multiconsistency, for constant k the algorithms can still be considered useful. We are currently working to further reduce the complexity of these algorithms.

There are many questions that remain to be explored in the context of multiconsistency. On the theoretical level, one would hope that efficient specialized algorithms can be found for many global constraints, whether exact algorithms

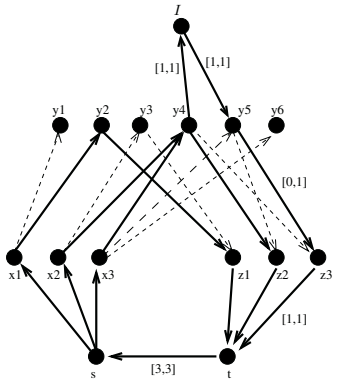

Fig. 6. Example: $I(y_1) = I(y_2) = I(y_3) = I(y_6) = 0$, $I(y_4) = -1$ and $I(y_5) = 1$. A flow in the augmented graph corresponds to a generalized parity matching

such as the ones in this paper, or faster approximation algorithms (i.e., algorithms that determine k-multiconsistency correctly for edges that participate in much fewer or much more than k solutions, but might make errors regarding edges that participate in approximately k solutions). In addition, it would be good to have a theoretical analysis that will better clarify the meaning of multiconsistency. An example of such a result could be a probabilistic analysis that correlates the robustness of the solution to a random *AllDifferent* constraint with the level of consistency of this solution, i.e., the maximal k for which this solution is k-multiconsistent. A third type of theoretical result could be the following: Given a constraint, efficiently compute a "reasonable" value of k for this constraint, where "reasonable" could mean a k such that at least $1/4$ and at most $3/4$ of the edges are k-multiconsistent. It seems desirable to determine such values, because we do not gain much information by computing k-multiconsistency with an "unreasonable" k: Most of the edges fall into the same set (consistent or inconsistent), so we cannot conclude any preferences among them.

Finally, there are questions that will need to be explored experimentally. What is the practical value of the heuristics we propose? Does a constraint solver really find a solution faster if it prefers to assign the "useful" values? Does the robustness increase in practice (even if there does not exist a theoretical guarantee?) Are there other heuristics that can be conceived and which use multiconsistency?

References

1. K.R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.
2. N. Beldiceanu, I. Katriel, and S. Thiel. Filtering algorithms for the Same constraint. In *CP-AI-OR 2004*, volume 3011 of *LNCS*, pages 65–79, 2004.
3. N. Bleuzen-Guernalec and A. Colmerauer. Optimal narrowing of a block of sortings in optimal time. *Constraints*, 5(1–2):85–118, 2000.

4. K. Fukuda and T. Matsui. Finding all the perfect matchings in bipartite graphs. *Appl. Math. Lett.*, 7(1):15–18, 1994.
5. E. Hebrard, B. Hnich, and T. Walsh. Super solutions in constraint programming. In *CP-AI-OR 2004*, volume 3011 of *LNCS*, pages 157–172. Springer-Verlag, 2004.
6. J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Computing*, 2(4):225–231, 1973.
7. I. Katriel and S. Thiel. Fast bound consistency for the global cardinality constraint. In *CP 2003*, volume 2833 of *LNCS*, pages 437–451, 2003.
8. A. Lopez-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 2003.
9. K. Mehlhorn and S. Thiel. Faster Algorithms for Bound-Consistency of the Sortedness and the Alldifferent Constraint. In *CP 2000*, volume 1894 of *LNCS*, pages 306–319, 2000.
10. J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 359–366, July 1998.
11. C.-G. Quimper, A. López-Ortiz, P. van Beek, and A. Golynski. Improved algorithms for the *global cardinality* constraint. In *CP 2004*, volume 3258 of *LNCS*, pages 542–556. Springer-Verlag, 2004.
12. C.-G. Quimper, P. van Beek, A. Lopez-Ortiz, A. Golynski, and S. B. Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *CP 2003*, pages 600–614, 2003.
13. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI-94*, pages 362–367, 1994.
14. J.-C. Régin. Generalized Arc-Consistency for Global Cardinality Constraint. In *AAAI 1996*, pages 209–215, 1996.
15. W.J. van Hoeve. The alldifferent constraint: A survey. In *Manuscript*, 2001.