I N F O R M A T I K

Filtering Algorithms for the *Same* and *UsedBy* Constraints

Nicolas Beldiceanu, Irit Katriel and Sven Thiel

MPI–I–2004–1–001 February 2004

FORSCHUNGSBERICHT RESEARCH REPORT

M A X - P L A N C K - I N S T I T U T
F Ü R
I N F O R M A T I K

Stuhlsatzenhausweg 85 66123 Saarbrücken Germany

Authors' Addresses

Nicolas Beldiceanu
LINA FRE CNRS 2729
Ecole des Mines de Nantes
La Chantrerie
4, rue Alfred Kastler. B.P. 20722
F-44307 NANTES Cedex 3, France
email: `Nicolas.Beldiceanu@emn.fr`

Irit Katriel, Sven Thiel
Stuhlsatzenhausweg 85
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany
email: `{irit,sthiel}@mpi-sb.mpg.de`

Abstract

We define the *Same* and *UsedBy* constraints. *UsedBy* takes two sets of variables X and Z such that $|X| \geq |Z|$ and assigns values to them such that the multiset of values assigned to the variables in Z is contained in the multiset of values assigned to the variables in X. *Same* is the special case of *UsedBy* in which $|X| = |Z|$. We show algorithms that achieve arc consistency and bound consistency for these constraints.

This work is partially supported by DFG grant SA 933/1-1 and by the EU IST Programme, IST-1999-14186 (ALCOM-FT).

Keywords

Constraint programming, Global Constraints, Bound Consistency, Strongly Connected Component.

1 Introduction

As a motivating example, we consider simple scheduling problems of the following type. The organization Doctors Without Borders [8] has a list of doctors and a list of nurses, each of whom volunteered to go on one rescue mission in the next year. Each volunteer specifies a list of possible dates and each mission should include one doctor and one nurse. The task is to produce a list of pairs such that each pair includes a doctor and a nurse who are available on the same date and each volunteer appears in exactly one pair. Since the list of potential rescue missions at any given date is infinite, it does not matter how the doctor-nurse pairs are distributed among the different dates.

We model this problem by the $\mathit{Same}(X = \{x_1, \ldots, x_n\}, Z = \{z_1, \ldots, z_n\})$ constraint which is defined on two sets X and Z of distinct variables such that $|X| = |Z|$ and each $v \in X \cup Z$ has a domain $D(v)$. A solution is an assignment of values to the variables such that the value assigned to each variable belongs to its domain and the multiset of values assigned to the variables of X is identical to the multiset of values assigned to the variables of Z.

This problem can be generalized to the case in which there are more nurses than doctors and the task is to create a list of pairs as above, with the requirement that every doctor appears in exactly one pair and every nurse in at most one pair (naturally, not all of the nurses will be paired). For this version we use the general case of the $\mathit{UsedBy}(X = \{x_1, \ldots, x_n\}, Z = \{z_1, \ldots, z_m\})$ constraint where $|X| = n \geq m = |Z|$ and a solution is an assignment of values to the variables such that the multiset of values assigned to the variables of Z is contained in the multiset of values assigned to the variables of X.

Given a constraint with domains for the variables, the first question is whether there exists an assignment of values to the variables which satisfies the constraint. The second question is whether we can efficiently identify elements in the domains of the variables that cannot participate in any solution to the constraint. An algorithm for this task is called a *filtering* algorithm. Filtering algorithms are classified according to the level of consistency they achieve. The *arc consistency* problem is to reduce the domains of the variables such that for all $v \in X \cup Z$ and $i \in D(v)$, there is a solution to the constraint in which v is assigned the value i. In the *bound consistency* problem we assume that there is a total order on the values in the domains of the variables and that for each $v \in X \cup Z$, $D(v)$ is a contiguous interval of values, i.e., $D(v) = [\underline{D}(v), \overline{D}(v)]$. The problem is to shrink these intervals to the minimum possible sizes without losing any solutions. This implies that if the domains are bound consistent, then for each $v \in X \cup Z$, there is at least one solution to the constraint in which v is assigned the value $\underline{D}(v)$ and there is at least one solution in which it is assigned the value $\overline{D}(v)$.

1.1 *Same* $= 2\times$ *GCC*?

The *Same* constraint can be modeled by two *Global Cardinality* constraints [4, 6], one on the set X and the other on the set Z, where count variables which are associated with the same value are not duplicated. We show here that consistency for all of the variables of the *GCC* constraints (including assignment and count variables) does not imply consistency for the *Same* constraint.

In our example, $|X| = |Z| = 2$ and $|Y| = 4$. The domains of the assignment variables are: $D(x_1) = \{1, 2\}$, $D(x_2) = \{3, 4\}$, $D(z_1) = \{1, 2, 3, 4\}$ and $D(z_2) = \{3, 4\}$. By examining the variable-value graphs[1] shown in Figure 1, one can see that all values are consistent w.r.t. the *GCC* constraints $GCC(\{x_1, x_2\}, \{v_1, v_2, v_3, v_4\})$ and $GCC(\{z_1, z_2\}, \{v_1, v_2, v_3, v_4\})$, while an arc consistency or bound consistency computation for $Same(\{x_1, x_2\}, \{z_1, z_2\})$ would remove 3 and 4 from the domain of z_1 – if z_1 is assigned 3 or 4, then 1 and 2 cannot both be assigned to the same number of variables from X and Z because one of them must be assigned to x_1 and neither can be assigned to z_2.

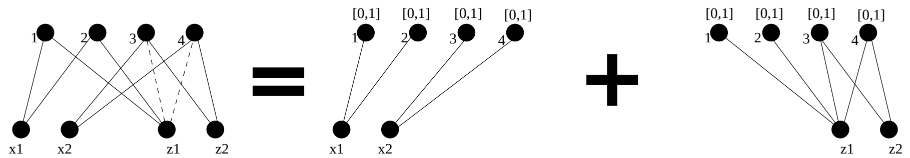

Figure 1: Example showing that consistency of the two *GCC*'s does not imply consistency of the *Same* constraint.

1.2 Our contribution

In this paper we show filtering algorithms for the *Same* and *UsedBy* constraints. Let Y be the union of the domains of the variables in $X \cup Z$ and let $n' = |Y|$. The arc consistency algorithms run in time $O(n^2n')$ and the bound consistency algorithms in time $O(n'\alpha(n', n') + n \log n)$, where α is the inverse of Ackermann's function[2].

[1]This construction will be formally defined in Section 2.

[2]For all practical purposes, $\alpha(n', n')$ can be regarded as a small constant.

We begin with *Same* and then generalize to *UsedBy*. The general approach resembles the flow-based filtering algorithms for the *AllDifferent* [5, 6] and *Global Cardinality* (*GCC*) [4, 7] constraints: We construct a bipartite variable-value graph, find a single solution in it and compute the strongly connected components (SCCs) of the residual graph. We show that an edge is consistent iff both of its endpoints are in the same SCC.

The main difference compared to the previous constraints that were solved by the flow-based approach is that we now have three sets of nodes. One set for each set of variables and a third set for the values. This difference significantly complicates the bound consistency algorithms, in particular the SCC computation compared to the corresponding stage in the *AllDifferent* and *GCC* cases. Our contribution is therefore not only in providing a solution to these constraints but also in showing that the ideas that appear in the previous algorithms can be extended to much more complex variable-value graphs.

In Section 2 we define the variable-value graph for the *Same* constraint and characterize the solutions to the constraint in terms of subsets of the edges in this graph. In Section 3 we show the arc consistency algorithm for the *Same* constraint and in Section 4 we show the bound consistency algorithm. Finally, in Section 5 we deal with the generalization to *UsedBy*. Source code for the bound consistency algorithms is available by request from the authors. A preliminary version of Sections 2-4 appeared as [1].

2 The *Same* constraint

We represent the *Same* constraint as a bipartite graph $B = (X \cup Z, Y, E)$, which we call the *variable-value graph*, where $E = \{\{v, y\} | v \in X \cup Z \wedge y \in Y \wedge y \in Dom(v)\}$. That is, the nodes on one side represent the variables and the nodes on the other represent the values and every variable is connected by an edge to all values in its domain.

The following definition and lemma characterize the set of all solutions to the constraint in terms of subsets of edges of B.

Definition 1 *Let $M \subseteq E$ be a set of edges of B. For any node $v \in X \cup Y \cup Z$, let $N_M(v)$ be the set of nodes which are neighbors of v in $B' = (X \cup Z, Y, M)$. We say that M is a* parity matching *in B iff $\forall_{v \in X \cup Z} |N_M(v)| = 1$ and $\forall_{y \in Y} |N_M(y) \cap X| = |N_M(y) \cap Z|$.*

Lemma 1 *There is a one to one correspondence between the solutions to the Same constraint and the parity matchings in B.*

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

Table 1: Domains of the variables for our example.

Proof Given a parity matching M in B, we can construct the solution

$$Same(\{N_M(x_1), \ldots, N_M(x_n)\}, \{N_M(z_1), \ldots, N_M(z_n)\}).$$

Since $|N_M(v)| = 1$ for all $v \in X \cup Z$, all of the assignments are well defined. In addition, for each edge (v, y) in B, and in particular in M, $y \in Dom(v)$. Finally, since $|N_M(y) \cap X| = |N_M(y) \cap Z|$ for all $y \in Y$, each value is assigned the same number of times to variables of X and Z. Hence, the constraint is satisfied.

On the other hand, given a solution

$$Same(\{y(x_1), \ldots, y(x_n)\}, \{y(z_1), \ldots, y(z_n)\})$$

where $y(v)$ is the value assigned to the variable v, we can obtain the set of edges $M = \{\{v, y\} | v \in X \cup Z \wedge y = y(v)\}$. Since $y(v) \in Dom(v)$ for all v, we have $M \subseteq E$. In addition, since $y(v)$ is determined for all variables, we have that $|N_M(v)| = 1$ for all $v \in X \cup Z$ and since each $y \in Y$ appears the same number of times in $\{y(x_1), \ldots, y(x_n)\}$ and in $\{y(z_1), \ldots, y(z_n)\}$), we have that $|N_M(y) \cap X| = |N_M(y) \cap Z|$, so M is a parity matching. ∎

In the next sections we show the filtering algorithms for the *Same* constraint, first the arc consistency algorithm and then bound consistency. We will illustrate the algorithm with the aid of the following example. $|X| = |Z| = 3$, $|Y| = 6$ and the domains of the variables of $X \cup Z$ are as in Table 1.

3 Arc consistency

In this section we show an algorithm that achieves arc consistency for the *Same* constraint. Inspired by Régin [7], we convert the graph B into a capacitated and directed graph $\vec{B} = (\vec{V}, \vec{E})$, as follows. We direct the edges from X to Y and from Y to Z and assign capacity requirements of $[0, 1]$ to each of these edges. We add two nodes s and t, an edge with capacity $[1, 1]$ from s to each $v \in X$ and from each $v \in Z$ to t and an edge with capacity $[n, n]$ from t to s. Figure 2 shows this graph for the constraint on the variables in Table 1. A flow in $\vec{B}$ is feasible iff there is a flow of value n on the arc from t to s. This implies that one unit of flow goes through every node in $X \cup Z$. By flow conservation, every node in Y is connected by edges that carry flow to the same number of nodes from X and from Z.

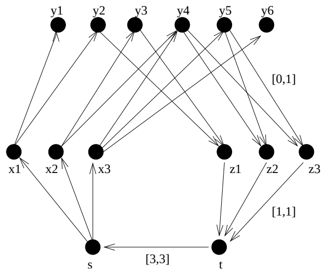

Figure 2: The capacitated graph for the example in Table 1.

Observation 1 *There is a one to one correspondence between parity matchings in B and feasible flows in $\vec{B}$.*

The algorithm uses Ford and Fulkerson's augmenting paths method to find a feasible flow f in $\vec{B}$ (e.g., the flow shown by the bold edges in Figure 3). If there is no such flow it reports that the constraint is not satisfiable. Otherwise, it removes the nodes s and t and builds the *residual graph* $\vec{B}_f = (\vec{V}_f, \vec{E}_f)$ where $\vec{V}_f = X \cup Y \cup Z$ and $\vec{E}_f = \vec{E} \cup \{(v, u) | u, v \in \vec{V} \wedge (u, v) \in \vec{E} \wedge f(e) = 1\}$. That is, all edges appear in their original orientation and the edges that carry flow appear also in reverse direction (see Figure 4). The following lemma shows that $\vec{B}_f$ can be used to determine which of the edges of the graph are consistent.

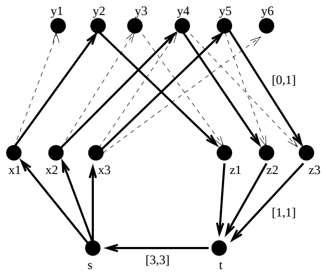

Figure 3: A feasible flow in the graph of Figure 2.

Lemma 2 *An edge $e = (u, v) \in \vec{B}_f$ is consistent iff u and v belong to the same SCC.*

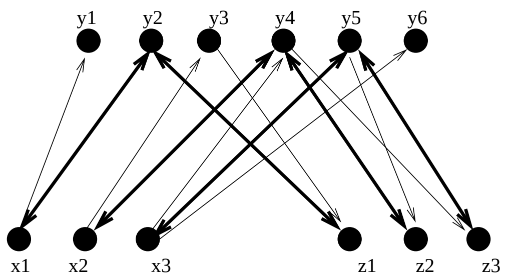

Figure 4: The residual graph with respect to the flow of Figure 3.

Proof If $f(e) = 1$ then e participates in the solution that corresponds to the flow f and is therefore consistent. u and v are connected by two antiparallel edges so they are in the same SCC. Let $M = \{e' | f(e') = 1\}$ be the parity matching that corresponds to the flow f. Assume that $e \notin M$ and that u and v are in the same SCC. Then there is a cycle P which uses e in $\vec{B}_f$. Starting at any y node on the cycle, number its nodes: $y_0, v_0, y_1, v_1, \ldots, y_{p-1}, v_{p-1}$ where $y_i \in Y$ and $v_i \in X \cup Z$ for all $0 \leq i < p$. Let $M' = M \oplus P$ be the symmetric difference between the parity matching M and the cycle P. That is, $M' = \{e' | e' \in M \wedge e' \in P \wedge e' \notin M \cap P\}$. If we look at $y_i \in P$ and its two neighbors, we have four possibilities: If $v_{(i-1) \bmod p}, v_i \in X$ or $v_{(i-1) \bmod p}, v_i \in Z$ then in M' each of $v_{(i-1) \bmod p}, v_i$ remains matched and y_i is matched with the same number of nodes from each of X and Z as it was in M'.
The two other options are that $v_{(i-1) \bmod p} \in X$ and $v_i \in Z$ or $v_{(i-1) \bmod p} \in Z$ and $v_i \in X$. Then the edges $(v_{(i-1) \bmod p}, y_i)$ and (y_i, v_i) are either both in M or both not in M. To see that this is true, note that the cycle must leave each $v \in X \cup Z$ that it enters. It can only enter a node $v \in X$ by an edge which is in M so it must leave it by an edge which is not in M. On the other hand, it can only enter a node $v \in Z$ by an edge which is not in M and leave it by an edge in M. This implies that either $v_{(i-1) \bmod p} \in Z$ and $v_i \in X$ and both edges are in M or $v_{(i-1) \bmod p} \in X$ and $v_i \in Z$ and both edges are in M'. In either case, when comparing M with M', y_i either lost or gained a neighbor from each of X and Z, so the number of neighbors of y_i in each of those sets remains equal.

Each of $v_{(i-1) \bmod p}, v_i$ is adjacent on P to one edge from M and one edge which is not in M. Hence, in M' it is still matched with exactly one y-node. We get that M' is a parity matching that contains e, so e is consistent.

It remains to show that an edge e which is not in an SCC of $\vec{B}_f$ (which implies $f(e) = 0$) is not consistent. Let $C_1, \ldots, C_k$ be the SCCs of $\vec{B}_f$. Since all edges in M appear in both directions, any edge between two SCCs is not in M. For any node v, let $C(v)$ be the SCC that v belongs to and let $E_{in} = \{(u,v)|C(u) = C(v)\}$ be the edges for which both endpoints are in the same SCC. Assume that an edge $e \in E \setminus E_{in}$ is consistent and let M' be a parity matching such that $e \in M'$. Consider the graph $\vec{B}'_f = (\vec{V}_f, \vec{E}'_f)$ where $\vec{E}'_f = E_{in} \cup M'$. That is, $\vec{B}'_f$ contains all the edges within SCCs plus the edges between SCCs which are in M'. If we shrink each SCC of $\vec{B}'_f$ into a single node, we get a DAG (directed acyclic graph). Let D_e be the connected component of this DAG which contains e and let C be a root of D_e, i.e., there are only outgoing edges from C. Let E_{xy} be the $x \to y$ edges and E_{yz} the $y \to z$ edges out of C. Since C is not an isolated node, $|E_{xy}| + |E_{yz}| \geq 1$. Let Y_C be the set of y nodes in the SCC represented by C. Then $|M'(Y_C) \cap X| = |M(Y_C) \cap X| - |E_{xy}|$ and $|M'(Y_C) \cap Z| = |M(Y_C) \cap Z| + |E_{yz}|$. Hence, M and M' cannot both be parity matchings, contradicting our assumption. ∎

Figure 5 shows the SCCs of $\vec{B}_f$ for the example in Table 1. The nodes of each SCC have a distinct shape and the inconsistent edges are dashed.

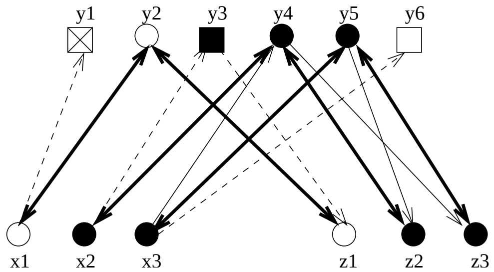

Figure 5: The SCCs of the residual graph of Figure 4.

Let $|E|$ denote the number of edges in $\vec{B}$ and recall that $n = |X| = |Z|$ and $n' = |Y|$. Clearly, $|E| = O(nn')$. The running time of the algorithm is dominated by the time it takes to find a feasible flow, which is $O(n|E|) = O(n^2n')$ [3, 7].

4 Bound consistency

In this section we show the bound consistency algorithm. It does the same as the arc consistency algorithm of the previous section, but achieves faster running time by exploiting the simpler structure of the variable-value graph: Since the domain of every variable node is an interval, the variable-value graph is *convex*, which means that the neighborhood of every variable node

```
(* Assumption: X and Z are sorted according to D̲. *)
P_x ← [] (* priority queue containing x nodes sorted by D̄ *)
P_z ← [] (* priority queue containing z nodes sorted by D̄ *)
j ← 0
for i = 1 to n' do
    forall x_h with D̲(x_h) = i do P_x.Insert x_h
    forall z_h with D̲(z_h) = i do P_z.Insert z_h
    (* Assume that MinPriority of an empty queue is ∞ *)
    while P_x.MinPriority = i or P_z.MinPriority = i do
        if P_x.IsEmpty or P_z.IsEmpty then report failure
        j ← j + 1
        x ← P_x.ExtractMin; match x with y_i
        z ← P_z.ExtractMin; match z with y_i
    end while
endfor
if P_x and P_z are both empty then report success
else report failure
```

Figure 6: Algorithm to find a parity matching in a convex graph.

is a consecutive sequence of value nodes. In the next subsections we show how each step of the arc consistency algorithm can be performed efficiently on a convex graph. In particular, we show how to find a parity matching in time $O(n' + n \log n)$ and how to compute the SCCs of the residual graph in time $O(n'\alpha(n', n'))$.

4.1 Finding a parity matching

Figure 6 shows the algorithm for finding a parity matching in the graph B. It uses two priority queues, P_x for the nodes in X and P_z for the nodes in Z. In both queues the nodes are sorted by the upper endpoints of their domains.

For any $v \in X \cup Z$, let $\underline{D}(v)$ and $\overline{D}(v)$ denote the lower and upper endpoints of $D(v)$, respectively. The algorithm traverses the value nodes from y_1 to $y_{n'}$ and for each y_i inserts to the respective queue all variable nodes $v \in X \cup Z$ with $\underline{D}(v) = i$. It then checks whether there is a node in one of the queues (the node with minimum priority) whose domain ends at i. If so, it tries to match this node and a node from the other queue with y_i. If the other queue is empty, it declares that there does not exist a parity matching in the graph.

Figure 7 shows the parity matching obtained by the algorithm for the example in Table 1. It corresponds to the flow shown in Figure 3. Note that while the flow algorithm could have produced any one of several different flows, this matching is one of the only two possible outputs of the algorithm of Figure 6. The other option would be to match z_2 with y_5 and z_3 with y_4 instead of the other way around.

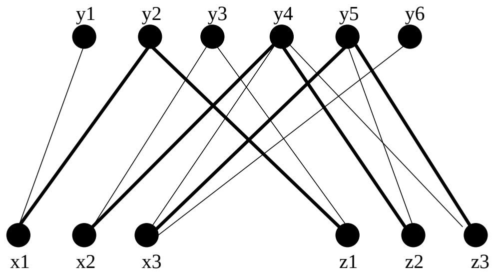

Figure 7: The parity matching obtained by the algorithm in Figure 6 for our running example.

Lemma 3 *If there is a PM in B then the algorithm in Figure 6 finds one.*

Proof We show by induction on i that if there is a PM M then for all $0 \leq i \leq n'$, there is a PM M_i in B which matches $\{y_1, \ldots, y_i\}$ with the same matching mates as the algorithm.

For $i = 0$, the claim holds with $M_0 = M$. For larger i, given a PM M, we can assume by the induction hypothesis that there is a PM M_{i-1} that matches the nodes in $\{y_1, \ldots, y_{i-1}\}$ with the same matching mates as the algorithm. We show how to construct M_i from M_{i-1}. As long as the matching mates of y_i are not the same as the ones determined by the algorithm, perform one of the following transformations:

If y_i is matched with a pair x_j, z_k such that neither one of x_j and z_k was matched with y_i by the algorithm, then since x_j and z_k were not matched by the algorithm with any of $\{y_1, \ldots, y_{i-1}\}$, they both remained in the queues after iteration i, which implies $\overline{D}(x_j) > i$ and $\overline{D}(z_k) > i$. In M_i we match both of them with y_{i+1}.

The other option is that the algorithm matched y_i with a pair x_j, z_k which are not both matched with y_i in M_i. Since the algorithm extracted them from the queues, we know that at least one of them has upper domain endpoint equal to i. Assume w.l.o.g. that $\overline{D}(x_j) = i$. Since M_{i-1} agrees with the algorithm on the matching mates of $\{y_1, \ldots, y_{i-1}\}$, we get that in M_{i-1}, x_j is matched with y_i and z_k is matched with $y_{i'}$ for some $i' > i$. Hence, there is some other node $z_{k'} \in Z$ which is matched with y_i in M_{i-1} and which was matched by the algorithm with $y_{i''}$ for some $i'' > i$. When the algorithm extracted z_k from the queue, $z_{k'}$ was in the queue because it is a neighbor of y_i. Hence, $\overline{D}(z_{k'}) \geq \overline{D}(z_k)$ so we can replace z_k and $z_{k'}$. That is, we can match z_k with y_i and $z_{k'}$ with $y_{i'}$.

Each time we apply one of the transformations above to M_{i-1}, we decrease the number of differences between the set of matching mates of y_i in M_{i-1} and in the matching generated by the first i iterations of the algorithm. So we can continue until we obtain a matching M_i that agrees with the algorithm on the matching mates of $\{y_1, \ldots, y_i\}$. ∎

Lemma 4 *If the algorithm in Figure 6 reports success then it constructs a parity matching in B.*

Proof If the algorithm reports success then P_x and P_z are empty at the end, which means that all nodes in $X \cup Z$ were extracted and matched with y nodes. In addition, since the algorithm did not report failure during the extractions, whenever $v \in X \cup Z$ was matched with y_i, we have that $\underline{D}(v) \leq i \leq \overline{D}(v)$. For all $1 \leq i \leq n'$, whenever y_i is matched with some node from X it is also matched with a node from Z, and vice versa. Hence, we get that the matching that was constructed is a parity matching. ∎

4.2 Finding strongly connected components

Having found a parity matching in B, which we can interpret as a flow in $\vec{B}$, we next wish to find the SCCs of $\vec{B}_f$ (cf. Figure 5). Mehlhorn and Thiel [5] gave an algorithm that does this in the residual graph of the *AllDifferent* constraint in time $O(n)$ plus the time required for sorting the variables according to the lower endpoints of their domains. Katriel and Thiel [4] enhanced this algorithm for the *GCC* constraint, in which a value node can be matched with more than one variable node. For our graph, we need to construct a new algorithm that can handle the distinction between the nodes in X and in Z and the more involved structure of the graph.

As in [4, 5], the algorithm in Figure 10 begins with n' initial components, each containing a node $y_i \in Y$ and its matching mates (if any). It then merges these components into larger ones. While the algorithm used for the *AllDifferent* graph can do this in one pass over the y nodes from $y_1, \ldots, y_{n'}$, our algorithm makes two such passes for reasons that will be explained in the following. The first pass resembles the SCC algorithm for the *AllDifferent* graph. It traverses the y nodes from y_1 to $y_{n'}$ and uses a stack to merge components which are strongly connected and are adjacent to each other. It maintains a list *Comp* of completed components and a stack CS of temporary components. The components in both *Comp* and CS are not guaranteed to be SCCs of $\vec{B}_f$. They are strongly connected but may not be maximal. However, a component in *Comp* is completed with respect to the first pass, while the components in CS may still be merged with unexplored components and with other components in CS.

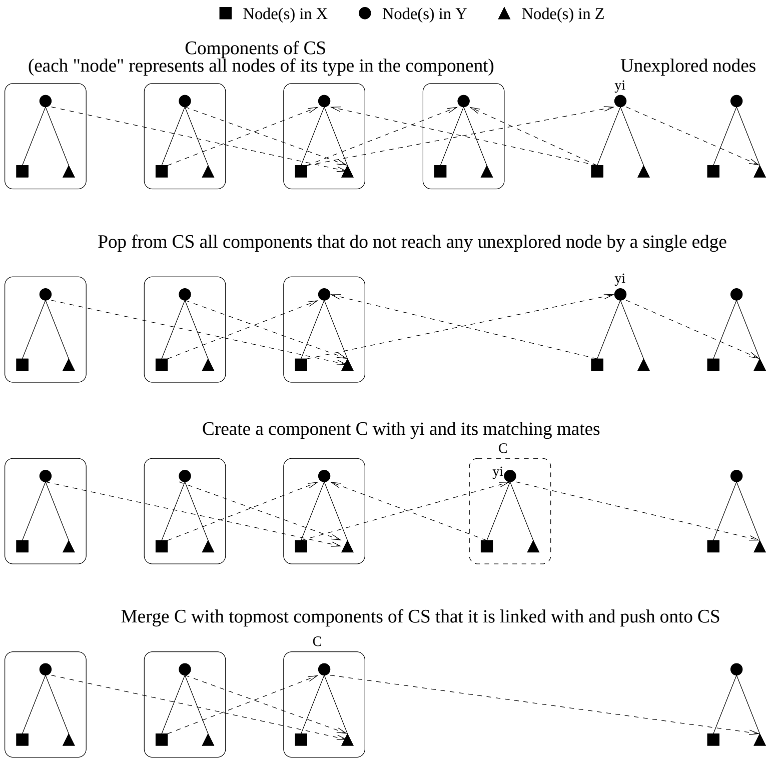

Figure 8: Iteration i of the first pass of the algorithm in Figure 10.

Let $\vec{B}_f^i$ be the graph induced by $\{y_1, \ldots, y_i\}$ and their matching mates. The first pass begins with the empty graph $\vec{B}_f^0$ and in iteration i moves from the graph $\vec{B}_f^{i-1}$ to $\vec{B}_f^i$, as follows. As long as the topmost component in CS does not reach any $y_{i'}$ with $i' > i$ by a single edge, this component is popped from CS and appended to *Comp*. Then the algorithm creates a new component C with y_i and its matching mates. It repeatedly checks if C and the topmost component in CS reach each other by a single edge in each direction. If so, it pops the component from CS and merges it into C. Finally, it pushes C onto CS and proceeds to the next iteration (see Figure 8).

The reason that this pass is enough for the *AllDifferent* graph but not for ours is that in our case the outgoing edges of a y-node do not fulfil the convexity criteria: It could be that there is an edge from y_i to a z node which is matched with $y_{i'}$ for some $i' > i+1$ while there is no edge from y_i to any of y_{i+1}'s matching mates. In the *AllDifferent* case, this could not happen: If a matching mate of y_i can reach any $y_{i'}$ by a single node then convexity implies that it can reach all y nodes between y_i and $y_{i'}$. This means that in our graph, there could be two components C, C' in CS such that C reaches C' by a single edge and C' reaches C by a single edge, but this is not detected when the second of them was inserted into CS because the first was not the topmost in the stack (see, e.g., Figure 9). The second pass, which merges such components, will be described later.

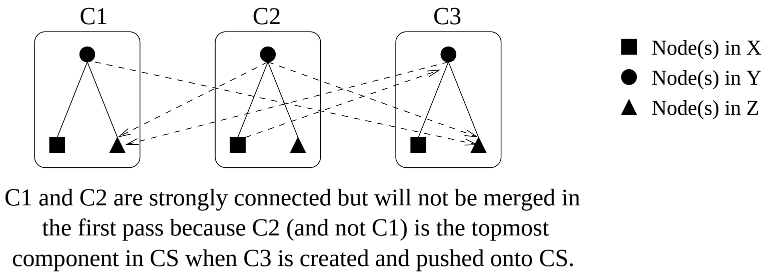

Figure 9: Example of components that will not be merged by the first pass of the SCC algorithm.

In the following, whenever we speak of a component C of $\vec{B}_f$, we refer to a set of nodes such that for every node in C, all of its matching mates are also in C. This means that a component C is strongly connected, but may not be maximal. We say that a component C *reaches* a component C' if there is a path in $\vec{B}_f$ from a node in C to a node in C'. In addition, two components C and C' are *linked* if there is an edge from C to C' and there is an edge from C' to C. They are *linked by* $x \to y$ *(linked by* $y \to z$*) edges* if the edges in both directions are $x \to y$ ($y \to z$) edges.

For clarity, the pseudo-code in Figure 10 uses the following shortcuts. We will show how to implement them with linear-time preprocessing in Subsection 4.4. Let C, C' be two components.

- $MinY[C]$ ($MaxY[C]$) is the minimum (maximum) index of a y node in C.
- $ReachesRight[C]$ is the largest index i such that y_i or one of its matching mates can be reached by a single edge from a node in C.

- *xyLeftLinks*$[C]$ (*yzLeftLinks*$[C]$) is true iff C is linked with some component to its left by $x \to y$ ($y \to z$) edges.
- *Linked*$[C, C']$ is true iff C and C' are linked.
- *xyLinks*$[C, C']$ (*yzLinks*$[C, C']$) is true iff C and C' are linked by $x \to y$ ($y \to z$) edges.

The following lemmas examine the components that are generated by the first pass of the algorithm, first with respect to their order and connectivity in CS and then with respect to the SCCs of $\vec{B}_f$ that they compose. They will help us to show that the SCCs of $\vec{B}_f$, when viewed as combinations of components that are generated by the first pass of the algorithm, have a relatively simple structure which enables to identify them in the second pass.

Lemma 5 *Let $CS = < C_1, C_2, \cdots >$ be the components in CS at the end of iteration i of the first pass (ordered from bottom to top). Then for all κ, $MaxY[C_\kappa] < MinY[C_{\kappa+1}]$. In other words, no component is nested in another and the components appear in CS in increasing order of the indices of their y nodes.*

Proof At the beginning CS is empty and the claim clearly holds. Assume that the claim holds after iteration $i-1$ and consider the changes made to the stack during iteration i. First some of the topmost components are popped from the stack; this does not affect the correctness of the claim. Then some of the topmost components are merged with each other and with the new component C and the result is pushed to the top of the stack. By the induction hypothesis, all components that were popped and merged contain y nodes with larger indices than the components that remained in the stack. In addition, all y nodes in CS after iteration $i-1$ have indices smaller than i. We get that the claim holds after iteration i. ∎

Lemma 6 *Let $CS = < C_1, C_2, \cdots >$ be the components in CS at the end of iteration i of the first pass (ordered from bottom to top). Then for all κ, C_κ and $C_{\kappa+1}$ are not linked.*

Proof Again, the claim clearly holds for the empty stack. Assume that it is true after iteration $i-1$. By the induction hypothesis, the claim holds for every adjacent pair of components that remained in CS after popping the completed components. If the new component C that is pushed onto CS is linked with the component C' which is immediately below it, then the

(* Pass 1: Start with singleton components and merge adjacent ones ***)**
$Comp \leftarrow$ empty list
$CS \leftarrow$ empty stack
for $i = 1$ **to** n' **do**
    while CS not empty $\wedge \mathit{ReachesRight}(\mathrm{Top}(CS)) < i$ **do**
        (* $\mathrm{Top}(CS)$ cannot reach $y_{i'}$ with $i' \geq i$. ***)**
        $C' \leftarrow \mathrm{Pop}(CS)$; append C' to $Comp$
    end while
    $C \leftarrow \{y_i\} \cup N_M(y_i)$
    while CS not empty $\wedge \mathit{Linked}(TOP(CS), C)$ **do**
        $C' \leftarrow \mathrm{Pop}(CS)$; $C \leftarrow C' \cup C$
    end while
    push C onto CS
endfor
while CS not empty **do** $C' \leftarrow \mathrm{Pop}(CS)$; append C' to $Comp$
(* Pass 2: merge non-adjacent components ***)**
$SCCs \leftarrow$ empty list
$CS \leftarrow$ empty stack
for $i = 1$ **to** $|Comp|$ **do** **(*** Traverse the components of $Comp$ by $MinY[C]$ order ***)**
    while CS not empty $\wedge \mathit{ReachesRight}(\mathrm{Top}(CS)) < MinY[C_i]$ **do**
        (* $\mathrm{Top}(CS)$ cannot reach $C_{i'}$ with $i' \geq i$. ***)**
        $C \leftarrow \mathrm{Pop}(CS)$
        if $\mathit{xyLeftLinks}[C]$ **then** push C onto $CSxy$
        else if $\mathit{yzLeftLinks}[C]$ **then** push C onto $CSyz$
        else append C to $SCCs$
        if $\mathit{xyLinks}[\mathrm{Top}(CS), \mathrm{Top}(CSxy)]$ **then**
            $C \leftarrow \mathrm{Pop}(CS)$; $C' \leftarrow \mathrm{Pop}(CSxy)$
            $C \leftarrow C' \cup C$
            while $\mathit{Linked}[C, \mathrm{Top}(CS)]$ **do** $C' \leftarrow \mathrm{Pop}(CS)$; $C \leftarrow C' \cup C$
            push C onto CS
        endif
        if $\mathit{yzLinks}[\mathrm{Top}(CS), \mathrm{Top}(CSyz)]$ **then**
            $C \leftarrow \mathrm{Pop}(CS)$; $C' \leftarrow \mathrm{Pop}(CSyz)$
            $C \leftarrow C' \cup C$
            while $\mathit{Linked}[C, \mathrm{Top}(CS)]$ **do** $C' \leftarrow \mathrm{Pop}(CS)$; $C \leftarrow C' \cup C$
            push C onto CS
        endif
    end while
    push C_i onto CS
endfor
return $SCCs$

Figure 10: Algorithm to find the SCCs of the residual graph.

algorithm would have popped C' and merged it with C. Hence, the claim holds at the end of iteration i. ∎

Lemma 7 *If $CS = < C_1, C_2, \cdots >$ is as in Lemma 5 and $\{C_{i_1}, \ldots, C_{i_\kappa}\}$ is a maximal set of components in CS which belong to the same SCC of $\vec{B}_f$ such that $i_1 \leq \cdots \leq i_\kappa$ then $C_{i_{\kappa-1}}$ and C_{i_κ} are linked.*

Proof Assume that there is such a set of components $\{C_{i_1}, \ldots, C_{i_\kappa}\}$ in CS where $C_{i_\kappa - 1}$, C_{i_κ} are not linked.
Case 1: C_{i_κ} does not reach $C_{i_{\kappa-1}}$ by a single edge. Then it must reach a component C_{i_j} with $i_j < i_{\kappa-1}$ by a single edge. This edge must be a $y \rightarrow z$ edge because otherwise convexity and Lemma 5 would imply that it also reaches $C_{i_{\kappa-1}}$ by an $x \rightarrow y$ edge, in contradiction to our assumption. Let z_γ be the target of this edge and y_β be its matching mate. Assume that β is maximal among β' such that $y_{\beta'}$ is in one of the components $C_{i_1}, \ldots, C_{i_{\kappa-2}}$ and it has a matching mate $z_{\gamma'}$ which is reachable from C_{i_κ} by a $y \rightarrow z$ edge. There is a path from z_γ to $C_{i_{\kappa-1}}$. If the path includes an $x \rightarrow y$ edge from a matching mate x_α of y_β to $y_{\beta'}$ with $\beta' > \beta$, then $\overline{D}(x_\alpha) > \beta$ and $\overline{D}(z_\gamma) > \beta$ so the algorithm in Figure 6 could not have matched x_α and z_γ with y_β, a contradiction. If the path includes a $y \rightarrow z$ edge from $y_{\beta'}$ with $\beta' \leq \beta$ to a node $z_{\gamma''}$ which is matched with $y_{\beta''}$ where $\beta'' > \beta$, then $z_{\gamma''}$ was in P_z when z_γ was extracted, so by convexity and Lemma 5 it is reachable from C_{i_κ} by a $y \rightarrow z$ edge, in contradiction to the maximality of β. We get that the path must go from z_γ to the left and then bypass y_β by an $x \rightarrow y$ edge $(x_{\alpha'}, y_{\beta''})$ such that $x_{\alpha'}$ is matched with $y_{\beta'}$ and $\beta' < \beta < \beta''$. We can assume w.l.o.g. that the path from z_γ to $x_{\alpha'}$ does not go to the left of $x_{\alpha'}$; if it does then one can show that either Lemma 6 is violated or there also exists a path that shortcuts the part that goes to the left of $x_{\alpha'}$. If the path ends with a $y \rightarrow z$ edge into a matching mate $z_{\gamma'}$ of $y_{\beta'}$ followed by the matching edges $(z_{\gamma'}, y_{\beta'}) \circ (y_{\beta'}, x_{\alpha'})$ then $\overline{D}(x_{\alpha'}) > \beta'$ and $\overline{D}(z_{\gamma'}) > \beta'$ so the algorithm in Figure 6 could not have matched $x_{\alpha'}$ and $z_{\gamma'}$ with $y_{\beta'}$, a contradiction. So the path ends with an $x \rightarrow y$ edge $(x_{\alpha'''}, y_{\beta'})$ followed by the edge $(y_{\beta'}, x_{\alpha'})$, where $x_{\alpha'''}$ is matched with $y_{\beta'''}$ for some $\beta' < \beta''' \leq \beta$. $x_{\alpha'''}$ was in P_x when $x_{\alpha'}$ was extracted, so it also reaches $y_{\beta''}$ by an $x \rightarrow y$ edge. By continuing backwards along the path and applying the same considerations, we get that there is an $x \rightarrow y$ edge from x_α to $y_{\beta''}$, hence again $\overline{D}(x_\alpha) > \beta$ and $\overline{D}(z_\gamma) > \beta$, so the algorithm in Figure 6 could not have matched x_α and z_γ with y_β, a contradiction.
Case 2: $C_{i_{\kappa-1}}$ does not reach C_{i_κ} by a single edge. Then C_{i_κ} is reached from another component C_{i_j} by an $x \rightarrow y$ edge $(x_\alpha, y_{\beta'})$. Assume that x_α is matched with y_β and that β is maximal among β'' such that $y_{\beta''}$ is in one of

the components $\{C_{i_1},\ldots,C_{i_{\kappa-2}}\}$ and has a matching mate $x_{\alpha''}$ which reaches C_{i_κ} by an $x \to y$ edge. There is a path from $C_{i_{\kappa-1}}$ to y_β. If there is an edge $(x_{\alpha''}, y_\beta)$ from $x_{\alpha''}$ which is matched with $y_{\beta''}$ for some $\beta < \beta'' < \beta'$, then by convexity and Lemma 5, $x_{\alpha''}$ reaches C_{i_κ} by an $x \to y$ edge, in contradiction to the maximality of β. If there is an edge $(y_{\beta''}, z_\gamma)$ such that z_γ is matched with y_β and $\beta'' > \beta$ then $\overline{D}(x_\alpha) > \beta$ and $\overline{D}(z_\gamma) > \beta$ so the algorithm in Figure 6 could not have matched x_α and z_γ with y_β, a contradiction. We get that the path must bypass y_β by a $y \to z$ edge $(y_{\beta''}, z_{\gamma'''})$ such that $z_{\gamma'''}$ is matched with $y_{\beta'''}$ for some $\beta''' < \beta < \beta''$, and then return from $z_{\gamma'''}$ to y_β. With arguments similar to the ones used in case 1, we get that the path from $z_{\gamma'''}$ to y_β must consist of $y \to z$ edges, which implies that there is a $y \to z$ edge from $y_{\beta''}$ to z_γ, hence again $\overline{D}(x_\alpha) > \beta$ and $\overline{D}(z_\gamma) > \beta$ so the algorithm in Figure 6 could not have matched x_α and z_γ with y_β, a contradiction. ∎

Corollary 1 *Let $CS = < C_1, C_2, \cdots >$ and $\{C_{i_1},\ldots,C_{i_\kappa}\}$ be as in Lemma 7. Then $C_{i_{\kappa-1}}$ and C_{i_κ} are either linked by $x \to y$ edges or linked by $y \to z$ edges.*

Proof Lemma 7 guarantees that $C_{i_{\kappa-1}}$ and C_{i_κ} are linked. Assume that these edges are not of the same type. That is, one is an $x \to y$ edge and the other is a $y \to z$ edge. Then by convexity and Lemma 5 we get that either $C_{i_{\kappa-1}}$ and $C_{i_{\kappa-1}+1}$ or $C_{i_\kappa-1}$ and C_{i_κ} are linked, and this contradicts Lemma 6. ∎

Lemma 8 *Let $CS = < C_1, C_2, \cdots >$ and $\{C_{i_1},\ldots,C_{i_\kappa}\}$ be as in Lemma 7 and assume that there is at least one unexplored node which is in the same SCC of $\vec{B}_f$ as $\{C_{i_1},\ldots,C_{i_\kappa}\}$. Then there is an edge from C_{i_κ} to an unexplored node.*

Proof Assume the converse. Then there is a component in $\{C_{i_1},\ldots,C_{i_{\kappa-1}}\}$ which is connected by a single edge to an unexplored node. Let j be the maximal index such that $C_j \in \{C_{i_1},\ldots,C_{i_{\kappa-1}}\}$ has an edge to an unexplored node. If this is a $y \to z$ edge then by convexity and Lemma 5 C_{i_κ} is also connected by a $y \to z$ edge to an unexplored node. So it must be an $x \to y$ edge. Let $x_j \in C_j$ be its source. With arguments similar to the ones used in the proof of Lemma 7 we can show that there cannot be a path from C_{i_κ} back to x_j, in contradiction to the assumption that C_{i_κ} and C_j are in the same SCC of $\vec{B}_f$. ∎

Corollary 2 *If $\{C_1, C_2, \ldots, C_{p_1}\}$ are the components found by the first pass of the algorithm and $\{C_{i_1},\ldots,C_{i_\kappa}\}$ is a maximal subset of these components*

which are strongly connected between them such that $MinY[C_{i_1}] \leq \cdots \leq MinY[C_{i_\kappa}]$. Then
(1) No component in $\{C_{i_1}, \ldots, C_{i_\kappa}\}$ is nested in another. That is, for all $j, j' \in \{i_1, \ldots, i_\kappa\}$ such that $j < j'$, $MaxY[C_j] < MinY[C_{j'}]$.
(2) $C_{i_\kappa - 1}$ and C_{i_κ} are linked. Furthermore, they are either linked by $x \to y$ edges or linked by $y \to z$ edges.

Proof Assume that there is a component C_j in the set which is nested in another component $C_{j'}$. Then $C_{j'}$ consists of nodes which are both to the left and to the right of C_j. This means that at some point, the algorithm merged these nodes into one component. At this point in time, the topmost component in CS consisted only of nodes which are to the left of C_j and it was merged with a component that contains only nodes to the right of C_j. By Lemma 5, this means that C_j was popped before this iteration, but this contradicts Lemma 8, so we have shown that (1) holds.
By Lemma 8, we know that none of $\{C_{i_1}, \ldots, C_{i_{\kappa-1}}\}$ were popped from CS before C_{i_κ} was pushed onto it. At a certain iteration, C_{i_κ} was pushed onto CS and stayed there at least until the next iteration. This, together with Corollary 1, implies (2). ∎

To sum up, the first pass partitions the nodes into components such that the components that compose an SCC of $\vec{B}_f$ are not nested within one another, and the two rightmost components of each SCC are linked by edges of the same type. The second pass of the algorithm merges components that belong to the same SCC. It starts with an empty stack CS and an empty list $SCCs$ and traverses the components found in the first pass by increasing order of $MinY[C]$. When considering a new component C, it first pops from CS all topmost components that cannot reach C or beyond it. For each such component C', it first checks if C' is linked with a component to its left by $x \to y$ edges. If so, there are components in CS that it needs to be merged with. So the algorithm pushes C' to a second stack $CSxy$. Otherwise, it checks if C' is linked with a component to its left by $y \to z$ edges and if so, pushes it to a third stack $CSyz$. Otherwise, it appends C' to $SCCs$ because C' is not linked with any component in CS and it does not reach unexplored components.

Before popping the next component, it checks whether the topmost component in CS and the topmost component in $CSxy$ are linked by $x \to y$ edges. If so, it pops each from its stack and merges them. It then repeatedly checks if the merged component is linked with the topmost component on CS. If so, the two are merged. Finally, the component which is the result of the merges is pushed back onto CS. The algorithm then checks whether the topmost component in CS and the topmost component in $CSyz$ are linked by $y \to z$ edges and if so, handles this in a similar way.

In the remaining part of this section we show that this algorithm finds the SCCs of $\vec{B}_f$. Denote the set of components found in the first pass of the algorithm by $Comp = \{C_1, C_2, \ldots, C_{p_1}\}$, such that for all $1 \leq i < p_1$, $MaxY[C_i] < MinY[C_{i+1}]$. Since each of these components is strongly connected but is not necessarily an SCC of $\vec{B}_f$, the components in $Comp$ are partitioned into sets of components such that the components of each set compose an SCC of $\vec{B}_f$.

Definition 2 *A subset $S = \{C_{i_1}, \ldots, C_{i_\kappa}\}$ of $Comp$ is an* SCC set *if the union of the components in S is an SCC of $\vec{B}_f$. In the following we will use this notation while assuming that $MinY[C_{i_1}] < \cdots < MinY[C_{i_\kappa}]$.*

Definition 3 *Let $S_1 = \{C_{i_1}, \ldots, C_{i_\kappa}\}$ and $S_2 = \{C_{j_1}, \ldots, C_{j_{\kappa'}}\}$ be distinct SCC sets. Then S_1 is* nested *in S_2 if there exists $j \in \{j_1, \ldots, j_{\kappa'-1}\}$ such that for all $C_i \in S_1$, $MinY[C_j] < MinY[C_i] < MinY[C_{j+1}]$. S_1 and S_2 are* interleaved *if there exist $i, i' \in \{i_1, \ldots, i_{\kappa-1}\}$ and $j, j' \in \{j_1, \ldots, j_{\kappa'-1}\}$ such that $MinY[C_i] < MinY[C_j] < MinY[C_{i'}] < MinY[C_{j'}]$.*

The SCC sets of $Comp$ can be interleaved in one another, but in the following lemma we show that this can only occur in a restricted form. This will help us to show that the second pass of Algorithm 10 identifies all SCC sets in $Comp$.

Lemma 9 *Let $S_1 = \{C_{i_1}, \ldots, C_{i_\kappa}\}$ and $S_2 = \{C_{j_1}, \ldots, C_{j_{\kappa'}}\}$ be interleaved SCC sets. If $MinY[C_i] < MinY[C_j] < MinY[C_{i+1}] < MinY[C_{j+1}]$ then there are links between $S_i^l = \{C_1, \ldots, C_i\}$ and $S_i^h = \{C_{i+1}, \ldots, C_{i_\kappa}\}$ and there are links between $S_j^l = \{C_1, \ldots, C_j\}$ and $S_j^h = \{C_{j+1}, \ldots, C_{j_{\kappa'}}\}$. These links can be of one of two forms: (1) S_i^l and S_i^h are linked by $x \to y$ edges; S_j^l and S_j^h are linked by $y \to z$ edges. (2) S_i^l and S_i^h are linked by $y \to z$ edges; S_j^l and S_j^h are linked by $x \to y$ edges.*

Proof We show that any other option is not possible. If S_i^l and S_i^h are linked by an $x \to y$ edge from S_i^l to S_i^h and a $y \to z$ edge from S_i^h to S_i^l then by convexity and Lemma 5, S_i^l and S_j^l are also linked, which means that the components of S_1 and S_2 belong to the same SCC of $\vec{B}_f$, contradicting the assumption that S_1 and S_2 are distinct SCC sets. The same follows if S_i^l and S_i^h are linked by a $y \to z$ edge from S_i^l to S_i^h and an $x \to y$ edge from S_i^h to S_i^l and if S_j^l and S_j^h are linked by an $x \to y$ edge in one direction and a $y \to z$ edge in the other.

Assume that each of the pairs S_i^l, S_i^h and S_j^l, S_j^h is linked by $x \to y$ edges in both directions. Then by convexity and Lemma 5, S_j^l and S_i^h are also linked by $x \to y$ edges and again the components of S_1 and S_2 are in the same SCC of $\vec{B}_f$. The same holds if the pairs S_i^l, S_i^h and S_j^l, S_j^h are both linked by $y \to z$ edges. ∎

Figure 11 shows an example of two interleaved SCC sets $S_1 = \{C_1, C_3, C_5\}$ and $S_2 = \{C_2, C_4\}$. For clarity, some of the edges were not drawn but the reader should assume that all edges that are implied by convexity exist in the graph. The SCCs of S_1 are linked by $x \to y$ edges and the SCCs of S_2 are linked by $y \to z$ edges.

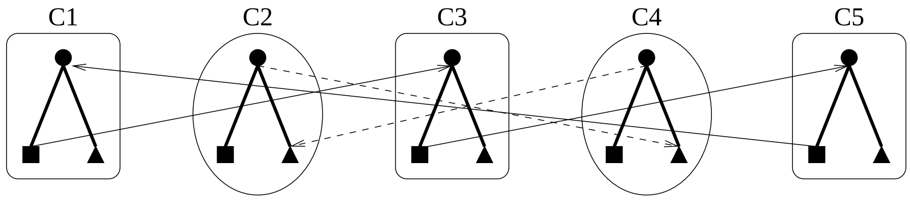

Figure 11: Two interleaved SCC sets.

The following resemble Lemmas 5 and 8 but refer to the second pass.

Lemma 10 *Let $CS =< C_1, C_2, \cdots >$ be the components in CS at the end of iteration i of the second pass (ordered from bottom to top). Then for all κ, $MaxY[C_\kappa] < MinY[C_{\kappa+1}]$.*

Proof At the beginning CS is empty and the claim clearly holds. Assume that the claim holds after iteration $i-1$ and consider the changes made to the stack during iteration i. When a topmost component of the stack is popped it does not affect the correctness of the claim. When a new component is pushed onto the stack it is the result of merging some of the topmost components in the stack with components from the temporary stacks $CSxy$ and $CSyz$. Since the temporary stacks contain only components with higher y nodes than CS, the merged component contains indices which are all higher than what is in components which are below it in CS. ∎

Lemma 11 *Let $S = \{C_{i_1}, \ldots, C_{i_\kappa}\}$ be an SCC set. Then for all $1 \leq \ell \leq \kappa - 1\}$, C_{i_ℓ} reaches at least one of $\{C_{i_{\ell+1}}, \ldots, C_{i_\kappa}\}$ by a single edge.*

Proof By Lemmas 5 and 10, the components of S appear in CS in the second pass in the same order in which they appear in the first pass. This implies that after a certain iteration of the second pass, $\{C_{i_1}, \ldots, C_{i_\ell}\}$ are in CS and $\{C_{i_{\ell+1}}, \ldots, C_{i_\kappa}\}$ are unexplored. Since there is an edge from one of $\{C_{i_1}, \ldots, C_{i_\ell}\}$ to one of $\{C_{i_{\ell+1}}, \ldots, C_{i_\kappa}\}$, we can show the claim by arguments which are similar to the ones used in the proof of Lemma 8. ∎

We can now show that the second pass of Algorithm 10 identifies the SCCs of $\vec{B}_f$.

Lemma 12 *Let $S = \{C_{i_1}, \ldots, C_{i_\kappa}\}$ be an SCC set. Then in the second pass of the algorithm, the components of S will be merged.*

Proof Let S be an SCC set such that all SCC sets which are nested in S were merged. We show by induction that all components of S are merged in the second pass. That is, we show that for all i from $i_{\kappa-1}$ to i_1, $C_i, \ldots, C_{i_\kappa}$ will be merged. For $i = i_{\kappa-1}$, we know by Corollary 2 that $C_{i_{\kappa-1}}$ and C_{i_κ} are linked by either $x \rightarrow y$ edges in both directions (case 1) or $y \rightarrow z$ edges in both directions (case 2). By Lemma 11, we know that $C_{i_{\kappa-1}}$ is not popped from CS before C_{i_κ} is pushed onto it. When C_{i_κ} is popped from CS for the first time, it is pushed onto the stack CS' where CS' is $CSxy$ (case 1) or $CSyz$ (case 2). Assume that when $C_{i_{\kappa-1}}$ became the topmost component in CS, C_{i_κ} was not the topmost component in CS'. If it was popped from CS' before that time, this is because a component above $C_{i_{\kappa-1}}$ in CS is linked with it. Since this component is not in S, this contradicts the assumption that S is an SCC set. On the other hand, if there was another component C' above C_{i_κ} in CS', then by Lemma 9 this is because C' is linked by $x \rightarrow y$ edges (case 1) or $y \rightarrow z$ edges (case 2) with another component C'' which was above $C_{i_{\kappa-1}}$ in CS. If C' and C'' were not merged, we get that there must have been a component above C' in CS' when C'' was the topmost component on CS. Applying the same argument recursively, we get that the number of components is infinite. Hence, C' and C'' were merged before $C_{i_{\kappa-1}}$ was the topmost component in CS, so C' could not have been above C_{i_κ} in CS'.

Assume that $C_{i_j+1}, \ldots, C_{i_\kappa}$ were merged by the algorithm into a larger component C. By Lemma 11 we know that C_{i_j} reaches C by a single edge and this implies that it was not popped from CS before any of $C_{i_j+1}, \ldots, C_{i_\kappa}$. In addition, since C and C_{i_j} are in the same SCC, there is a path from C to C_{i_j}. This path does not go through components that are between C and C_{i_j} in the stack because that would place these components in the same SCC as C and C_{i_j}, contradicting the assumption that S is an SCC set. Assume that the path begins with an $x \to y$ edge from C to C_{i_j} or a component below it in CS. Then by convexity there is also an $x \to y$ edge from C to C_{i_j}. If the edge from C_{i_j} to C is a $y \to z$ edge then by convexity and Lemma 10, C is linked with C_{i_j} and all components which are above it in the stack and will be merged with them by the algorithm. If this edge is an $x \to y$ edge then C will be pushed onto $CSxy$ and as in the base case, it will be merged with C_{i_j} when the later will become the topmost component in CS.

If the path from C to C_{i_j} begins with a $y \to z$ edge from C to a component below C_{i_j} then arguments similar to the ones used in the proof of Lemma 7 imply that C also reaches C_{i_j} by a $y \to z$ edge. If the edge from C_{i_j} to C is an $x \to y$ edge then again by convexity and Lemma 10 C_{i_j} and all components above it in CS are linked with C and will be merged with it. If it is a $y \to z$ edge then C will be pushed onto $CSyz$ and will be merged with C_{i_j} when the later will become the topmost component in CS. ∎

4.3 Narrowing the bounds

After the SCC computation we have for each node in $X \cup Z$ the index of its y mate and for each $y_i \in Y$ the component it belongs to. As in [4], we can in time $O(n + n')$ obtain for every component a list of its y nodes sorted by index and two lists of its x and z nodes, one sorted by the lower endpoints and one by the upper endpoints of their domains. Let C be a component, let $I_C = (i_1, \ldots, i_k)$ be the sorted list of the indices of its y nodes and let $\overline{D}_C = (\overline{D}_{j_1}, \ldots, \overline{D}_{j_\ell})$ be the sorted list of the upper domain endpoints of its x nodes. Then the upper endpoint of the narrowed domain of $x_j \in C$ is i_k where $i_k \leq \overline{D}_j < i_{k+1}$ (with $i_{k+1} = n' + 1$) and this value can be computed for all x nodes in C with one simultaneous scan of I_C and $\overline{D}_C$. Narrowing the lower endpoints of the domains of the x nodes and both endpoints of the domains of the z nodes can be done in a similar manner with the relevant lists.

4.4 Implementation details

Figure 12 shows how the shortcuts that are used in the pseudo-code in Figure 10 can be implemented with linear-time preprocessing. After finding the parity matching and before beginning the SCC algorithm, we compute for each $y_i \in Y$ several values:

$D^x[i] = [\underline{D^x}[i], \overline{D^x}[i]]$ is the union of the domains of the x nodes which are matched with y_i. Similarly, $D^z[i] = [\underline{D^z}[i], \overline{D^z}[i]]$ is the union of the domains of the z nodes which are matched with y_i. Both can be computed in linear time. For the $D^x[i]$'s we traverse the x nodes and for each x_j, set $\underline{D^x}[i] = \min\{\underline{D}(x_j), \underline{D^x}[i]\}$ and $\overline{D^x}[i] = \max\{\overline{D}(x_j), \overline{D^x}[i]\}$ where y_i is the value node that x_j is matched with. The computation of the $D^z[i]$'s is similar.

MinYZLink$[i]$ (*MaxYZLink*$[i]$) is the minimum (maximum) index i' of a y node which is matched with a node $z \in Z$ such that $i \in Dom(z)$, i.e., there is an edge from y_i to a matching mate of $y_{i'}$. The *MinYZLink*$[i]$'s can be computed in linear time as follows: Traverse the y nodes from left to right. For each y_i and for each $i' \in D^z(y_i)$, *MinYZLink*$[i']$ should be set to i unless it was already set to a smaller value. Since the $D^z(y_i)$'s are intervals and each $i \in D^z(y_i)$, it suffices to maintain the index of the largest i' for which *MinYZLink*$[i']$ was set and to traverse each $D^z(y_i)$ beginning at this index. The computation of *MaxYZLink*$[i]$ is symmetric.

Finally, *MaxLeftXYLink*$[i]$ is the maximum index $i' < i$ such that $y_{i'}$ is matched with some $x_j \in X$ with $i \in Dom(x_j)$ (if no such $y_{i'}$ exists then *MaxLeftXYLink*$[i] = n' + 1$, i.e., infinity. Symmetrically, *MinRightXYLink*$[i]$ is the minimum index $i' > i$ with the same property, and if none exists then *MaxLeftXYLink*$[i] = -1$, i.e., minus infinity. We begin by initializing *MaxLeftXYLink*$[i] = n' + 1$ for all i. For $i = 1$, this value remains and for higher i's *MaxLeftXYLink*$[i]$ can be computed as follows. Push y_1 onto a stack and then traverse the y nodes from y_2 to $y_{n'}$. When processing y_i, first pop from the stack all topmost nodes $y_{i'}$ such that $\overline{D^x}[i'] < i$. If the stack is not empty afterwards, then *MaxLeftXYLink*$[i]$ is the topmost node on the stack; this is the latest node that was pushed and has a matching mate $x \in X$ whose domain contains i. Last, y_i is pushed onto the stack. A symmetric algorithm can compute *MinRightXYLink*$[i]$ in linear time.

Once these values are known for each individual y node, the SCC algorithm can maintain them for every component: When a new component is created it contains a single y node and its matching mates, and when two components are merged, each value can be updated in constant time for the merged component. The shortcuts in the pseudocode can therefore be implemented as follows:
ReachesRight$[C]$ is $\max\{$*MaxYZLink*$[C], \overline{D^x}[C]\}$.
xyLeftLinks$[C]$ is true iff $\underline{D^x}[C] \leq$ *MaxLeftXYLink*$[C] <$ *MinY*$[C]$, which means that C is reached by an $x \to y$ edge from a component C' to its left and it also reaches C' by an $x \to y$ edge. *yzLeftLinks*$[C]$ is true iff (1) $\underline{D^z}[C] <$ *MinY*$[C]$ and (2) $\underline{D^z}[C] \leq$ *MinYZLink*$[C]$, which means that (1) C is reached by a $y \to z$ edge from at least one component to its left and (2) there is a component C' for which (1) holds and which is also reached from C by a $y \to z$ edge.
xyLinks$[C, C']$, where C is to the left of C', is true iff (1) $\overline{D^x}[C] \geq$ *MinY*$[C']$ and (2) $\underline{D^x}[C'] \leq$ *MaxY*$[C]$, which means that (1) C reaches C' by an $x \to y$

edge and (2) C' reaches C by an $x \to y$ edge. Similarly, $yzLinks[C, C']$ is true iff $\overline{D^z}[C] \geq MinY[C']$ and $\underline{D^z}[C'] \leq MaxY[C]$.
$Linked[C, C']$ is true iff $Reaches[C, C'] \wedge Reaches[C', C]$, where $Reaches[C_1, C_2]$ is true iff (1) $\overline{D^x}[C_1] \geq MinY[C_2]$ or (2) $\underline{D^z}[C_2] \leq MaxY[C_1]$, which means that (1) there is an $x \to y$ edge from C_1 to C_2 or (2) there is a $y \to z$ edge from C_1 to C_2.

4.5 Complexity analysis

The parity matching algorithm traverses the y nodes and performs $2n$ Insert and ExtractMin operations on the priority queues. This takes time $O(n' + n \log n)$. In the SCC computation everything takes linear time except one thing: Maintaining the component list. We wish to be able to merge components and to find which component a node belongs to. For this we use a Union-Find[2] data structure over the y nodes. We perform $O(n')$ operations on this structure, which in total take time $O(n'\alpha(n', n'))$. We get that the total running time of the algorithm is $O(n'\alpha(n', n') + n \log n)$.

5 The *UsedBy* constraint

In this section we show algorithms that achieve arc consistency and bound consistency for the general case of the *UsedBy* constraint. The simplest solution is to add $n - m$ dummy variables to Z with domains that include all values: $\{y_1, \ldots, y_{n'}\}$ in the arc consistency case and $[y_1, y_{n'}]$ in the bound consistency case. Let $\overline{Z}$ be the set obtained by padding Z in this way. Then we have that $|X| = |\overline{Z}| = n$ and we can apply the algorithms for the *Same* constraint. The correspondence between solutions to $Same(X, \overline{Z})$ and solutions to $UsedBy(X, Z)$ should be obvious.

In the rest of this section we show how the *UsedBy* constraint can be solved directly. While these algorithms are not asymptotically faster than what we get with the padding method described above, they should be faster in practice.

As in the case of the *Same* constraint, we construct the bipartite graph B and look for a solution in it. This time, a solution is a *z-parity matching*, defined as follows.

Definition 4 *Let $M \subseteq E$ be a set of edges of B. For any node $v \in X \cup Y \cup Z$, let $N_M(v)$ be the set of nodes which are neighbors of v in $B' = (X \cup Z, Y, M)$. We say that M is a z-*parity matching *in B iff $\forall_{v \in X \cup Z} |N_M(v)| = 1$ and $\forall_{y \in Y} |N_M(y) \cap X| \geq |N_M(y) \cap Z|$.*

for $i = 1$ **to** n' **do**
  $\underline{D^x}[i] \leftarrow i$; $\overline{D^x}[i] \leftarrow i$
  $\underline{D^z}[i] \leftarrow i$; $\overline{D^z}[i] \leftarrow i$
  $\mathit{MaxYZLink}[i] \leftarrow i$; $\mathit{MinYZLink}[i] \leftarrow i$
  $\mathit{MinRightXYLink}[i] \leftarrow -1$; $\mathit{MaxLeftXYLink}[i] \leftarrow n' + 1$
endfor
for $j = 1$ **to** n **do**
  $i \leftarrow \mathrm{YMate}(x_j)$; $\underline{D^x}[i] \leftarrow \min\{\underline{D^x}[i], \underline{D}(x_j)\}$; $\overline{D^x}[i] \leftarrow \max\{\overline{D^x}[i], \overline{D}(x_j)\}$
  $i \leftarrow \mathrm{YMate}(z_j)$; $\underline{D^z}[i] \leftarrow \min\{\underline{D^z}[i], \underline{D}(z_j)\}$; $\overline{D^z}[i] \leftarrow \max\{\overline{D^z}[i], \overline{D}(z_j)\}$
endfor

$\mathit{MinYZLink}[1] \leftarrow 1$; $i \leftarrow 2$
for Yindex $= 1$ **to** n' **do**
  while $i \geq \overline{D^z}[\mathrm{Yindex}]$ **do** $\mathit{MinYZLink}[i] \leftarrow \mathrm{Yindex}$; $i \leftarrow i + 1$
endfor
$\mathit{MaxYZLink}[n'] \leftarrow n'$; $i \leftarrow n' - 1$
for Yindex $= n'$ **to** 1 **do**
  while $i \geq \underline{D^z}[\mathrm{Yindex}]$ **do** $\mathit{MaxYZLink}[i] \leftarrow \mathrm{Yindex}$; $i \leftarrow i - 1$
endfor

$S \leftarrow []$ **(*** Empty stack ***)**
S.Push 1
for $i = 2$ **to** n' **do**
  while S.NotEmpty $\wedge$ $\overline{D^x}[S.\mathrm{Top}] < i$ **do** S.Pop
  if S.NotEmpty **then** $\mathit{MaxLeftXYLink}[i] \leftarrow S$.Top
  S.Push i
endfor
$S \leftarrow []$ **(*** Empty stack ***)**
S.Push n'
for $i = n' - 1$ **to** 1 **do**
  while S.NotEmpty $\wedge$ $\underline{D^x}[S.\mathrm{Top}] > i$ **do** S.Pop
  if S.NotEmpty **then** $\mathit{MinRightXYLink}[i] \leftarrow S$.Top
  S.Push i
endfor

Figure 12: Preprocessing stage for the algorithm in Figure 10

The following is a simple extention of Lemma 1.

Lemma 13 *There is a one to one correspondence between the solutions to the UsedBy constraint and the z-parity matchings in B.*

5.1 Arc consistency

As with the *Same* constraint, the arc consistency algorithm builds the directed graph $\vec{B}$ and finds an s-t flow of value m in it. The edges participating in this flow match all nodes of Z and a similar number of nodes from X, such that every z node corresponds to an x node which is assigned the same value. To get a z-parity matching M, we take the set of edges that participate in the flow and an edge connecting each of the remaining x nodes with an arbitrary value in its domain. We then construct the residual graph $\vec{B}_f$. As in the case of the *Same* constraint, $\vec{B}_f$ contains all edges in their original orientation and the matching edges also in reverse direction. Finally, we add a node s with an edge (y_i, s) for each value node $y_i \in Y$ and an edge (s, y_i) for each $y_i \in Y$ such that $|N_M(y_i) \cap X| > |N_M(y_i) \cap Z|$ (see Figure 13).

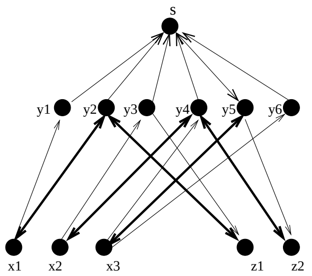

Figure 13: Example of a residual graph with the added node s.

The purpose of adding the node s is as follows. Let $y_i \in Y$ be such that $|N_M(y_i) \cap X| > |N_M(y_i) \cap Z|$ and let $x_j \in N_M(y_i) \cap X$. By matching x_j with any of its neighbors $y_{i'}$ with $i' \neq i$, we get an alternative z-parity-matching. More generally, let M be a z-parity matching obtained as described above and let P be an alternating path that begins at a node y_i with $|N_M(y_i) \cap X| > |N_M(y_i) \cap Z|$ and ends at another node $y_{i'} \in Y$. Then $M \oplus P$ is also a z-parity matching. By construction, there is an edge from s to y_i and an edge from $y_{i'}$ to s, so all edges on this path belong to the same SCC of $\vec{B}_f$. We prove this in the following lemma.

Lemma 14 *An edge $e = (u, v) \in \vec{B}_f$ is consistent iff u and v belong to the same SCC.*

Proof Let M be the z-parity matching that was obtained from the flow f as described above. If $e \in M$ then e participates in the solution that corresponds to the matching M and is therefore consistent. u and v are connected by two antiparallel edges so they are in the same SCC. Assume that $e \notin M$ and that u and v are in the same SCC. Then there is a cycle P which uses e in $\vec{B}_f$. If the cycle does not contain the special node s then as in Lemma 2, $M' = M \oplus P$ is a z-parity matching. If the cycle does contain the node s, consider the path $P' = P \setminus s$ which is the cycle P without s and the edges adjacent to it. P starts with an edge (y_i, v) from a node $y_i \in Y$ such that $|N_M(y_i) \cap X| > |N_M(y_i) \cap Z|$ and ends with an edge (u, y_j) into another node $y_j \in Y$. We show that $M' = M \oplus P'$ is a z-parity-matching. For an internal node (a node which is different from y_i and y_j), the situation is as in Lemma 2 so all nodes preserve the properties required of a z-parity-matching.

If $v \in X$ then the edge (y_i, v) is in M because otherwise it cannot be traversed in this direction. y_i is matched in M' with one X node less than in M, but this is fine because in M it is matched with more X than Z nodes. If $v \in Z$ then the edge may or may not be in M. In both cases we get that in M' y_i is matched with at least as many X nodes as Z nodes.

If $u \in Z$ then the edge (u, y_j) is in M and then y_j is matched in M' with less Z nodes and the same number of X nodes compared to M. If $u \in X$ then it is not in M, because there must be another edge incoming into u on the path and this must be the single edge of M which is adjacent to u. We get once again that y_j is matched in M' with one more X node than in M. So M' is a z-parity-matching.

The proof that an edge e which is not in an SCC of $\vec{B}_f$ is not consistent is identical to the corresponding part of Lemma 2. ∎

5.2 Bound consistency

The algorithm in Figure 14 finds a z-parity matching in a convex graph. As the parity matching algorithm of Figure 6, it traverses the nodes of Y from left to right. In iteration i it first inserts into the queue P_x (P_z) all nodes of X (Z) whose domains start at i, since they become matching candidates. It then pops from P_z all nodes whose domains end at i and matches each of them with y_i along with a node from P_x. If P_x becomes empty during this process, then not all of these z nodes can be matched so the algorithm reports failure. Next, it pops from P_x all nodes whose domains end at i and matches them with y_i. For each such x node, if P_z is not empty then a node is popped from it and is matched with y_i as well. If all n' iterations complete without reporting failure, the algorithm reports success.

Lemma 15 *If there is a z-parity matching in B then the algorithm in Figure 14 finds one.*

Proof The proof of this lemma is similar to the proof of Lemma 3. There are two cases that need to be added. The first is that y_i was matched by the algorithm with a node x_j and without a corresponding z node. This implies that $\overline{D}(x_j) = i$ and P_z was empty when x_j was extracted. Hence, in M_{i-1} x_j must also be matched with y_i and it does not have a corresponding z node. The second case is that y_i was matched in M_{i-1} with x_j and without a corresponding z node, and y_i was not matched with x_j by the algorithm. Then $\overline{D}(x_j) > i$ and we change M_{i-1} by matching x_j with y_{i+1} instead of y_i. ∎

Lemma 16 *If the algorithm in Figure 14 reports success then it constructs a z-parity matching in B.*

Proof If the algorithm reports success then P_x and P_z are empty at the end, which means that all nodes in $X \cup Z$ were extracted and matched with y nodes. In addition, since the algorithm did not report failure during the extractions, whenever $v \in X \cup Z$ was matched with y_i, we have that $\underline{D}(v) \leq i \leq \overline{D}(v)$. For all $1 \leq i \leq n'$, whenever y_i is matched with some node from Z it is also matched with a node from X. Hence, we get that the matching that was constructed is a z-parity matching. ∎

After finding a z-parity matching in B, we wish to find the SCCs of $\vec{B}_f$. We first find the SCCs of $\vec{B}_f \setminus \{s\}$, which is the graph $\vec{B}_f$ without the node s and the edges adjacent to it. For this we can use the algorithm in Figure 10; the fact that a y node may be matched with more x nodes than z nodes does not affect the algorithm or the proofs.

We then merge the SCCs of $\vec{B}_f \setminus \{s\}$ into the SCCs of $\vec{B}_f$. The general idea follows the one used in [4]: For each SCC of $\vec{B}_f \setminus \{s\}$ we determine whether it reaches s and whether it is reached from s. We then merge all components which reach s and are reached from s into one component. Unlike in the *GCC* case, all y nodes in the *UsedBy* graph reach s by a single edge so it only remains to determine which components are reached from s.

Definition 5 *The* index $I(v)$ *of a node $v \in X \cup Y \cup Z$ is defined as follows. For $y_i \in Y$, $I(y_i) = i$. For $v \in X \cup Z$, $I(v) = i$ where y_i is the matching mate of v.*

```
(* Assumption: X and Z are sorted according to D̲. *)
P_x ← [] (* priority queue containing x nodes sorted by D̄ *)
P_z ← [] (* priority queue containing z nodes sorted by D̄ *)
j ← 0
for i = 1 to n' do
    forall x_h with D̲(x_h) = i do P_x.Insert x_h
    forall z_h with D̲(z_h) = i do P_z.Insert z_h
    (* Assume that MinPriority of an empty queue is ∞ *)
    while P_z.MinPriority = i do
        if P_x.IsEmpty then report failure
        x ← P_x.ExtractMin; match x with y_i
        z ← P_z.ExtractMin; match z with y_i
    end while
    while P_x.MinPriority = i do
        x ← P_x.ExtractMin; match x with y_i
        if P_z.NotEmpty then
            z ← P_z.ExtractMin; match z with y_i
        endif
    end while
endfor
report success
```

Figure 14: Algorithm to find a z-parity matching in a convex graph.

Definition 6 *Let $P =< v_1, v_2, \ldots, v_p >$ be a simple path in $\vec{B}_f \setminus \{s\}$ and for each v_i let $Comp[v_i]$ be the SCC of $\vec{B}_f \setminus \{s\}$ that v_i belongs to. The* signature *$S(P)$ of P is the sequence of components visited by P. That is, $S(P) =< Comp[v_1], Comp[v_2], \ldots, Comp[v_p] >$.*

Assume that a path visits a node v, proceeds to a node u with $Comp[u] \neq Comp[v]$ and later visits a node w with $Comp[w] = Comp[v]$. Since v and w are in the same SCC, there is a path from v to w which visits nodes only inside this SCC. Hence, we will assume w.l.o.g. that a path does not return to a component that it has already visited and left. This means that in the signature of a path, $Comp[v_i] = Comp[v_j]$ implies $Comp[v_\kappa] = Comp[v_i]$ for all $i < \kappa < j$.

Definition 7 *Let $P =< v_1, v_2, \ldots, v_p >$ be a simple path in $\vec{B}_f \setminus \{s\}$ and for each v_i let v'_i be the first node on P such that $Comp[v_i] = Comp[v'_i]$. Then P is* left-monotonous *if $I(v'_\kappa) \geq I(v'_{\kappa+1})$ for all $1 \leq \kappa < p$. It is* right-monotonous *if $I(v'_\kappa) \leq I(v'_{\kappa+1})$ for all $1 \leq \kappa < p$.*

We next show that if there is a path from a component C to another component C' then there is also a path from C to C' which is either left-monotonous or right-monotonous.

Lemma 17 *If there is a path $P =< v_1, v_2, \ldots, v_p >$ then there is also a path $P' =< u_1, u_2, \ldots, u_{p'} >$ such that $u_1 = v_1$, $u_{p'} = v_p$ and P' is either left-monotonous or right-monotonous.*

Proof By induction on p. For $p = 1$, the claim clearly holds with $P' = P$. Assume that it holds for all $\bar{p} < p$ but not for p. Let $P =< v_1, v_2, \ldots, v_p >$ be a path. By the induction hypothesis, there is a path $P'' =< u_2, \ldots, u_{p'} >$ such that $u_2 = v_2$, $u_{p'} = v_p$ and P'' is monotonous. Assume w.l.o.g. that it is left-monotonous (the right-monotonous case is symmetric). If P'' goes through a node u_i with $Comp[u_i] = Comp[v_1]$, then since there is a path Q from v_1 to u_i which lies in the same SCC, the claim holds with $P' = Q \circ < u_i, \ldots, u_{p'} >$. Otherwise, if $I(v_1) > I(v_2)$ then since $v_2 = u_2$ the claim holds with $P' =< v_1, u_2, \ldots, u_{p'} >$, which is left-monotonous.

If, on the other hand, $I(v_1) < I(v_2)$ then there are two cases. The first is that $I(v_1) < I(u_{p'})$. If the edge from v_1 to v_2 is an $x \to y$ edge then by convexity there is also an $x \to y$ edge from v_1 to all components on the path, and in particular to $Comp[u_{p'}]$. Then the claim holds with $P' =< v_1, y_{I(u_{p'})} >$, which is right-monotonous. If the edge from v_1 to v_2 is a $y \to z$ edge then by convexity there is also a $y \to z$ edge from $y_{I(u_{p'})}$ to v_2, which means that

$v_2, \ldots, u_{p'}$ are on a cycle and hence belong to one SCC. Again, the claim holds with the right-monotonous path $P' =< v_1, y_{I(u_{p'})} >$. The second case is that $I(u_{i+1}) < I(v_1) < I(u_i)$ for some $1 < i < p$. Since there is a path from v_1 to u_i and we assume that $Comp[v_1] \neq Comp[u_i]$, the edge from u_i to u_{i+1} is a $y \to z$ edge, because otherwise convexity would imply that there is an $x \to y$ edge from u_{i+1} to v_1, placing them in the same SCC. Hence, by convexity there is also a $y \to z$ edge from $y_{I(v_1)}$ to u_{i+1}, so the claim holds with $P' =< v_1, \ldots, y_{I(v_1)}, u_{i+1}, \ldots, u_{p'} >$. ∎

As a corollary, we get that we can find all components that are reached from s with three passes over the y nodes as shown in Figure 15. The first pass identifies the components that are reached from s by a single edge. The second pass traverses the y nodes from right to left and identifies the components that are reached from s by a left-monotonous path originating in a component that was marked in the first pass. The third pass is symmetric to the second. It traverses the y nodes from left to right and finds the components that are reached from s by a right-monotonous path originating at a component that was marked in one of the previous passes. In the following, when we speak of iteration i' of any pass of the algorithm, we mean the iteration in which $i = i'$, and not the ith iteration.

Lemma 18 *The algorithm in Figure 15 correctly marks the components that are reached from s.*

Proof Let C be a component that is reached from s. By Lemma 17, we know that C is also reached from s by a left-monotonous or right-monotonous path. Let p be the length of this path and $v \in C$ be the first node on it. If $p = 1$ then there is an edge (s, y_i) to some $y_i \in C$ and C will be marked in iteration i of the first pass.

For $p > 1$, if the last edge on the path is within the same component, then this component is reached by the path of length $p - 1$ that is obtained by discarding this edge. Then our claim follows by the induction hypothesis. We can therefore assume that the last edge on the path is between two distinct components, and in particular that it is not in M.

There are two cases to consider. The first is that the path is left-monotonous. We show by induction on p that it will be marked not later than in iteration $I(v)$ of the second pass. If the last edge on the path is a $y \to z$ edge (y_i, z_j) then $Comp[i]$ is reached from s by a left-monotonous path of length $p-1$ and by the induction hypothesis, it will be marked not later than during iteration i of the second pass. Then, *min_y_reached_from_s* will be set to $\underline{D^x}[Comp[i]] \leq i$. In iteration $I(z_j)$, $\overline{D^z}[C] \geq i \geq$ *min_y_reached_from_s* so C will be marked.

If the last edge on the path is an $x \to y$ edge (x_j, y_i), then by the induction hypothesis $Comp[I(x_j)]$ will be marked by the end of iteration $I(x_j)$ of the second pass and then *min_y_reached_from_s* will be set to at least $\underline{D^x}(x_j) \leq i$. In iteration i, we have *min_y_reached_from_s* $\leq i$ so C will be marked.

The second case is that C is reached from s by a right-monotonous path. We can symmetrically show by induction that it will be marked in the third pass of the algorithm. ∎

6 Future directions

We have shown that fast flow-based bound consistency algorithms can be designed when complex variable-value graphs are involved. We are working on extending these ideas to solve variations on the *Same* and *UsedBy* constraints. For example, a *GCC*-like restriction on the *Same* constraint: We still require that the multiset of values assigned to $\{x_1, \dots, x_n\}$ is the same as the multiset of values assigned to $\{z_1, \dots, z_n\}$, and in addition we have for each value i a lower bound and an upper bound on the number of variables in $\{x_1, \dots, x_n\}$ (and hence also $\{z_1, \dots, z_n\}$) that can be assigned the value i.

References

[1] N. Beldiceanu, I. Katriel, and S. Thiel. Filtering algorithms for the Same constraint. In *Proceedings of International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP'AI'OR 2004)*, LNCS. Springer, 2004. to appear.

[2] H.N. Gabow and R.E. Tarjan. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[3] L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[4] I. Katriel and S. Thiel. Fast bound consistency for the global cardinality constraint. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *LNCS*, pages 437–451, 2003.

```
(* Pass 1: Mark components that are reached from s by a single edge *)
for i = 1 to n' do
    if Reached_from_s[i] then
        Reached_from_s[Comp[i]] ← true
    endif
endfor
(* Pass 2: Mark components that are reached from s by a left-monotonous path *)
min_y_reached_from_s ← n' + 1
for i = n' to 1 do
    C ← Comp[i]
    if min_y_reached_from_s ≤ i then
        Reached_from_s[C] ← true
    endif
    if D̄^z[C] ≥ min_y_reached_from_s then
        Reached_from_s[C] ← true
    endif
    if Reached_from_s[C] then
        min_y_reached_from_s ← D̲^x[C]
    endif
endfor
(* Pass 3: Mark components that reach s by a right-monotonous path *)
max_y_reached_from_s ← 0
for i = 1 to n' do
    C ← Comp[i]
    if max_y_reached_from_s ≥ i then
        Reached_from_s[C] ← true
    endif
    if D̲^z[C] ≤ max_y_reached_from_s then
        Reached_from_s[C] ← true
    endif
    if Reached_from_s[C] then
        max_y_reached_from_s ← D̄^x[C]
    endif
endfor
```

Figure 15: Algorithm to mark the components that reach s.

[5] K. Mehlhorn and S. Thiel. Faster Algorithms for Bound-Consistency of the Sortedness and the Alldifferent Constraint. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2000)*, volume 1894 of *LNCS*, pages 306–319, 2000.

[6] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, 1994.

[7] J.-C. Régin. Generalized Arc-Consistency for Global Cardinality Constraint. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 209–215, 1996.

[8] Médecins sans Frontières. http://www.doctorswithoutborders.org/.

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2004-4-001	J. Haber	?
MPI-I-2004-1-003	I. Katriel	On Algorithms for Online Topological Ordering and Sorting
MPI-I-2004-1-002	P. Sanders, S. Pettie	A Simpler Linear Time 2/3 - epsilon Approximation for Maximum Weight Matching
MPI-I-2004-1-001	N. Beldiceanu, I. Katriel, S. Thiel	Filtering algorithms for the Same and UsedBy constraints
MPI-I-2003-NWG2-002	F. Eisenbrand	Fast integer programming in fixed dimension
MPI-I-2003-NWG2-001	L.S. Chandran, C.R. Subramanian	Girth and Treewidth
MPI-I-2003-4-009	N. Zakaria	FaceSketch: An Interface for Sketching and Coloring Cartoon Faces
MPI-I-2003-4-008	C. Roessl, I. Ivrissimtzis, H. Seidel	Tree-based triangle mesh connectivity encoding
MPI-I-2003-4-007	I. Ivrissimtzis, W. Jeong, H. Seidel	Neural Meshes: Statistical Learning Methods in Surface Reconstruction
MPI-I-2003-4-006	C. Roessl, F. Zeilfelder, G. Nürnberger, H. Seidel	Visualization of Volume Data with Quadratic Super Splines
MPI-I-2003-4-005	T. Hangelbroek, G. Nürnberger, C. Roessl, H.S. Seidel, F. Zeilfelder	The Dimension of C^1 Splines of Arbitrary Degree on a Tetrahedral Partition
MPI-I-2003-4-004	P. Bekaert, P. Slusallek, R. Cools, V. Havran, H. Seidel	A custom designed density estimation method for light transport
MPI-I-2003-4-003	R. Zayer, C. Roessl, H. Seidel	Convex Boundary Angle Based Flattening
MPI-I-2003-4-002	C. Theobalt, M. Li, M. Magnor, H. Seidel	A Flexible and Versatile Studio for Synchronized Multi-view Video Recording
MPI-I-2003-4-001	M. Tarini, H.P.A. Lensch, M. Goesele, H. Seidel	3D Acquisition of Mirroring Objects
MPI-I-2003-2-004	A. Podelski, A. Rybalchenko	Software Model Checking of Liveness Properties via Transition Invariants
MPI-I-2003-2-003	Y. Kazakov, H. Nivelle	Subsumption of concepts in $DL\ \mathcal{FL}_0$ for (cyclic) terminologies with respect to descriptive semantics is PSPACE-complete
MPI-I-2003-2-002	M. Jaeger	A Representation Theorem and Applications to Measure Selection and Noninformative Priors
MPI-I-2003-2-001	P. Maier	Compositional Circular Assume-Guarantee Rules Cannot Be Sound And Complete
MPI-I-2003-1-018	G. Schaefer	A Note on the Smoothed Complexity of the Single-Source Shortest Path Problem

MPI-I-2003-1-017	G. Schäfer, S. Leonardi	Cross-Monotonic Cost Sharing Methods for Connected Facility Location Games
MPI-I-2003-1-016	G. Schäfer, N. Sivadasan	Topology Matters: Smoothed Competitive Analysis of Metrical Task Systems
MPI-I-2003-1-015	A. Kovács	Sum-Multicoloring on Paths
MPI-I-2003-1-014	G. Schäfer, L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, T. Vredeveld	Average Case and Smoothed Competitive Analysis of the Multi-Level Feedback Algorithm
MPI-I-2003-1-013	I. Katriel, S. Thiel	Fast Bound Consistency for the Global Cardinality Constraint
MPI-I-2003-1-012		- not published -
MPI-I-2003-1-011	P. Krysta, A. Czumaj, B. Voecking	Selfish Traffic Allocation for Server Farms
MPI-I-2003-1-010	H. Tamaki	A linear time heuristic for the branch-decomposition of planar graphs
MPI-I-2003-1-009	B. Csaba	On the Bollobás – Eldridge conjecture for bipartite graphs
MPI-I-2003-1-008	P. Sanders	Polynomial Time Algorithms for Network Information Flow
MPI-I-2003-1-007	H. Tamaki	Alternating cycles contribution: a strategy of tour-merging for the traveling salesman problem
MPI-I-2003-1-006	M. Dietzfelbinger, H. Tamaki	On the probability of rendezvous in graphs
MPI-I-2003-1-005	M. Dietzfelbinger, P. Woelfel	Almost Random Graphs with Simple Hash Functions
MPI-I-2003-1-004	E. Althaus, T. Polzin, S.V. Daneshmand	Improving Linear Programming Approaches for the Steiner Tree Problem
MPI-I-2003-1-003	R. Beier, B. Vöcking	Random Knapsack in Expected Polynomial Time
MPI-I-2003-1-002	P. Krysta, P. Sanders, B. Vöcking	Scheduling and Traffic Allocation for Tasks with Bounded Splittability
MPI-I-2003-1-001	P. Sanders, R. Dementiev	Asynchronous Parallel Disk Sorting
MPI-I-2002-4-002	F. Drago, W. Martens, K. Myszkowski, H. Seidel	Perceptual Evaluation of Tone Mapping Operators with Regard to Similarity and Preference
MPI-I-2002-4-001	M. Goesele, J. Kautz, J. Lang, H.P.A. Lensch, H. Seidel	Tutorial Notes ACM SM 02 A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2002-2-008	W. Charatonik, J. Talbot	Atomic Set Constraints with Projection
MPI-I-2002-2-007	W. Charatonik, H. Ganzinger	Symposium on the Effectiveness of Logic in Computer Science in Honour of Moshe Vardi
MPI-I-2002-1-008	P. Sanders, J.L. Träff	The Factor Algorithm for All-to-all Communication on Clusters of SMP Nodes
MPI-I-2002-1-005	M. Hoefer	Performance of heuristic and approximation algorithms for the uncapacitated facility location problem
MPI-I-2002-1-004	S. Hert, T. Polzin, L. Kettner, G. Schäfer	Exp Lab A Tool Set for Computational Experiments
MPI-I-2002-1-003	I. Katriel, P. Sanders, J.L. Träff	A Practical Minimum Scanning Tree Algorithm Using the Cycle Property
MPI-I-2002-1-002	F. Grandoni	Incrementally maintaining the number of l-cliques
MPI-I-2002-1-001	T. Polzin, S. Vahdati	Using (sub)graphs of small width for solving the Steiner problem
MPI-I-2001-4-005	H.P.A. Lensch, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2001-4-004	S.W. Choi, H. Seidel	Linear One-sided Stability of MAT for Weakly Injective Domain
MPI-I-2001-4-003	K. Daubert, W. Heidrich, J. Kautz, J. Dischler, H. Seidel	Efficient Light Transport Using Precomputed Visibility
MPI-I-2001-4-002	H.P.A. Lensch, J. Kautz, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing, Transmission, and Interactive Display of High Quality 3D Models on the Web

MPI-I-2001-4-001	H.P.A. Lensch, J. Kautz, M. Goesele, W. Heidrich, H. Seidel	Image-Based Reconstruction of Spatially Varying Materials
MPI-I-2001-2-006	H. Nivelle, S. Schulz	Proceeding of the Second International Workshop of the Implementation of Logics
MPI-I-2001-2-005	V. Sofronie-Stokkermans	Resolution-based decision procedures for the universal theory of some classes of distributive lattices with operators
MPI-I-2001-2-004	H. de Nivelle	Translation of Resolution Proofs into Higher Order Natural Deduction using Type Theory