I N F O R M A T I K

Fast Bound Consistency for the Global Cardinality Constraint

Irit Katriel and Sven Thiel

MPI–I–2003–1–013 June 2003

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

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

Abstract

We show an algorithm for bound consistency of *global cardinality constraints*, which runs in time $O(n + n')$ plus the time required to sort the assignment variables by range endpoints, where n is the number of assignment variables and n' is the number of values in the union of their ranges. We thus offer a fast alternative to Régin's arc consistency algorithm [5] which runs in time $O(n^{3/2}n')$ and space $O(n \cdot n')$. Our algorithm also achieves bound consistency for the number of occurrences of each value, which has not been done before.

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, GCC, Bound Consistency.

1 Introduction

The *Global Cardinality Constraint* $GCC(x_1, \cdots, x_n, c_{v_1}, \cdots, c_{v_{n'}})$ is specified on n assignment variables $x_1, \ldots, x_n$ and n' count variables $c_{v_1}, \ldots, c_{v_{n'}}$. The idea is that each assignment variable x_j takes a value in $D = \{v_1, \cdots, v_{n'}\}$ and each value v_i is used exactly c_{v_i} times. With each assignment variable x_j we associate a domain $D_j \subseteq D$, and the domain of a count variable c_{v_i} is an interval $E_i = [L_i, U_i]$. For a tuple $t \in D^n$ and $v \in D$ denote by $occ(v, t)$ the number of occurrences of the value v in t. Then the set S containing all solutions of the constraint is defined as follows:

$$S = \{(w_1, \ldots, w_n; o_1, \ldots, o_{n'}) | \forall_j w_j \in D_j \wedge \forall_i occ(v_i, (w_1, \ldots, w_n)) = o_i \in E_i\}$$

Given a constraint with ranges for the variables, the first question is whether $S \neq \emptyset$, which means that there is an assignment of values to the variables which satisfies the constraint. The *arc consistency* problem for the assignment variables is to reduce the domains of these variables such that D_j is the projection of S onto its jth component. In the *bound consistency* problem we assume $v_1 < \cdots < v_{n'}$ and each D_j is a contiguous interval of values, i.e. $D_j = [\underline{D}_j, \overline{D}_j]$. The problem is to shrink the intervals of both the assignment and the count variables to the minimum sizes such that $S \subseteq D_1 \times \cdots \times D_n \times E_1 \times \cdots \times E_{n'}$. This means that for each $1 \leq j \leq n$, there is at least one tuple in S whose jth component is the smallest (largest) value in D_j. And for $i = 1, \ldots, n'$, there is a tuple in S whose $(n+i)$th component is the smallest (largest) value in E_i.

Régin [5] has provided an algorithm that achieves arc consistency for the assignment variables of a GCC, but does not reduce the domains of the count variables. He constructs a bipartite graph and computes a maximum flow in it. He then defines the residual graph with respect to this flow, finds the strongly connected components (SCCs) in the residual graph and discards the edges between SCCs that carry no flow. The algorithm runs in time $O(n^{3/2}n')$ and space $O(n \cdot n')$ and is dominated by the complexity of finding a maximum flow using Ford and Fulkerson's algorithm [3].

Mehlhorn and Thiel [4] considered the *AllDifferent* constraint, which requires that each variable is assigned a different value and is the special case of GCC in which $[L_i, U_i] = [0, 1]$ for all $1 \leq i \leq n'$. They propose an algorithm for bound consistency of this restricted case. While the algorithm is phrased in terms of bipartite matchings, it can also be interpreted as a simplified version of Régin's flow-based algorithm; the maximum flow in this case is a maximum matching in the graph. Their algorithm runs in linear time plus the time required to sort the variables according to the endpoints of their ranges. The improved running time compared to Régin's algorithm is achieved due to the fact that the algorithm exploits the simpler structure of the graph: the assumption that the range of each variable is an interval implies that the bipartite graph is *convex*, which means that the neighborhood of each variable node is a contiguous sequence of value nodes.

In this paper we show an algorithm of the same flavor for bound consistency of GCC. A significant difference between the problems is that in an AllDifferent constraint, only variable nodes must be matched but a value node may or may not be matched. Now, we also have lower bound requirements on the value nodes. We will show, again using matching terminology, that it is still possible to find a maximum flow in the bipartite graph and compute the SCCs in the residual graph in time which is linear in the number of nodes plus the time required for sorting the range endpoints of the assignment variables. In addition, we show two linear-time algorithms that narrow the domains of the count variables. The first is very simple and fast in practice but does not always achieve bound consistency. The second does achieve bound consistency but requires more elaborate data structures. Source code is available by request from the authors.

2 Preliminaries

2.1 Normalization of the x-ranges

The x-ranges are called *normalized* if $D = \{1, \cdots, n'\}$. This has the advantage that we can use the values as array indices. Normalization can always be achieved by identifying each value v_i with its index i in the sorted order $v_1 \leq \cdots \leq v_n$. To do this we need to compute for every original domain $D_j = [v_{l_j}, v_{h_j}]$ the corresponding normalized domain $D'_j = [l_j, h_j]$. After sorting the endpoints of $D_1, \cdots, D_n$ in ascending order, this can be done in time $O(n+n')$. We want to point out that sorting can be done in time $O(n)$, if the v_i's are integers drawn from the range $[1, n^k]$ for some fixed k. After achieving bound consistency for the normalized domains $D'_1, \ldots, D'_n$, we can easily narrow the original domains to bound consistency.

2.2 A generalization of matching

A *matching* in a graph is a subset M of its edges such that each node is adjacent to at most one edge in M. We generalize this notion. We consider capacitated graphs $G = (V, E, C)$ where C is a function that maps every node $\nu \in V$ to an interval $C(\nu) = [L_\nu, U_\nu]$. We call $C(\nu)$ the *capacity requirement* of ν. For a set M of edges and a node ν we denote by $M(\nu)$ the set of all nodes that are adjacent to ν by an edge in M. A *generalized matching* in G is a subset M of its edges such that for each node $\nu \in V$ we have $|M(\nu)| \in [L_\nu, U_\nu]$. We call $M(\nu)$ the set of *matching mates of* ν.

As in [4], we define the *intersection graph* of a GCC, which is in this case a capacitated bipartite graph with n nodes $\{x_1, \cdots, x_n\}$ representing the variables on one side and n' nodes $\{y_1, \cdots, y_{n'}\}$ representing the values in the ranges of the variables on the other side. The capacity ranges of the variable nodes are all $[1,1]$, indicating that each variable must be assigned exactly one value. For a value node, the capacities are according to the count requirements of the value represented by the node:

$$[L_\nu, U_\nu] = \begin{cases} [L_i, U_i] & \nu = y_i \\ [1,1] & \nu = x_j \end{cases}$$

The edge (x_j, y_i) exists in the graph if and only if $i \in [\underline{D_j}, \overline{D_j}]$.

Running example: Throughout the paper, we will illustrate the algorithms with the following example (see Figure 1): $GCC(x_1, \cdots, x_6, c_{v_1}, \cdots, c_{v_4})$ where the assignment variable ranges D_j and the count variable ranges E_i are

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

D_1	D_2	D_3	D_4	D_5	D_6
[1,1]	[1,2]	[1,2]	[2,2]	[3,4]	[3,4]

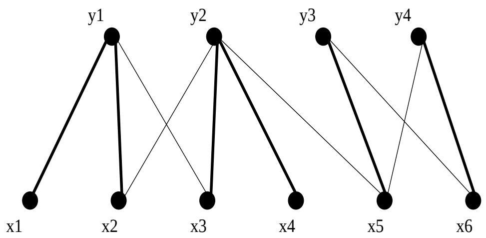

Figure 1: The intersection graph of the example GCC and a generalized matching in it, marked with bold edges.

Lemma 1 *Let T denote the projection of S onto its first n components. Every generalized matching $M = \{\{x_i, y_{g(i)}\} | 1 \le i \le n\}$ corresponds to the tuple $(g(1), \cdots, g(n))$ in T and vice versa.*

Proof Immediate from the definition of a generalized matching. ∎

2.3 The connection to Régin's algorithm

Régin's algorithm computes a maximum flow in a graph $\vec{H}$ which is similar to our intersection graph with a few differences: it is directed, contains two additional nodes s and t, and the capacity requirements are on the edges and not on the nodes. The edge set of $\vec{H}$ is defined as follows.[1] There is an edge (x_j, y_i) with capacity requirement $[0,1]$ iff $i \in D_j$. For every y_i there is an edge (y_i, s) with capacity bounds $[L_i, U_i]$, and for every x_j we have the edge (t, x_j) with capacity requirement $[0,1]$. Régin shows that the constraint has a solution iff the maximum flow from t to s has value n.

His algorithm searches for a flow F of value n. If none exists, it reports failure. Otherwise, it constructs the residual graph $\vec{R}_F$, which allows to determine for each variable the arc consistent values in its domain. $\vec{R}_F$ has the same nodes as $\vec{H}$ (except for t) and the following edges: an edge (x_j, y_i) iff $i \in D_j \wedge F(x_j, y_i) = 0$, an edge (y_i, x_j) iff $i \in D_j \wedge F(x_j, y_i) = 1$, an edge (y_i, s) iff $F(y_i, s) < U_i$, and finally an edge (s, y_i) iff $F(y_i, s) > L_i$. In Corollary 2 in [5], Régin proves that a value i in the domain D_j is consistent for x_j iff $F(x_j, y_i) = 1$ or x_j and y_i belong to the same SCC of $\vec{R}_F$.

We will now translate his construction into the language of generalized matchings in the intersection graph. Given an intersection graph G, a generalized matching M corresponds to the following flow F_M in $\vec{H}$. For an edge (x_j, y_i), $F_M(x_j, y_i) = 1$ if $\{x_j, y_i\} \in M$ and $F_M(x_j, y_i) = 0$ otherwise. $F_M(y_i, s) = |M(y_i)|$ for all i, and $F_M(t, x_j) = 1$ for all j. Observe that the total flow from t to s has value n.

We augment the residual graph $\vec{R}$ that corresponds to F_M such that we obtain a graph $\vec{G}$ with the following property: A value i in the domain of a variable x_j is arc consistent iff x_j and y_i are in the same SCC of $\vec{G}$. If x_j is matched with y_i in M, $\vec{R}$ contains the edge (y_i, x_j), and in $\vec{G}$ we add the edge (x_j, y_i) in the opposite direction. So if x_j and y_i are matched (equivalently, $F_M(x_j, y_i) = 1$) then they belong to the same SCC of $\vec{G}$. We call $\vec{G}$ the *oriented intersection graph.* Note that a similar augmentation appears in [4].

Lemma 2 follows easily from the discussion above and Régin's results cited in it:

Lemma 2 *An edge $\{x_j, y_i\}$ in G belongs to a generalized matching iff x_j and y_i are in the same SCC of $\vec{G}$.*

This lemma implies the correctness of the following bound consistency algorithm for the assignment variables:

1. Find a generalized matching in G.

[1] In fact, the edges in $\vec{H}$ have the opposite direction compared to Régin's graph. We prefer the reversed orientation because it is equivalent and simplifies the following presentation.

2. Construct $\vec{G}$ and compute its SCCs.

3. Narrow the ranges as much as possible such that they still represent all edges within the SCCs.

We want to point out that our algorithm uses $O(n + n')$ space and does not construct any of these graphs explicitely.

3 Finding a generalized matching in the intersection graph

The algorithm for finding a generalized matching in a convex graph is given in Figure 2. It makes three passes over the y nodes. In the first two passes it goes from left to right and uses a priority queue P to which x nodes are inserted when they become candidates for matching and in which they are sorted according to the upper endpoints of their domains. That is, the node x that is extracted from P by an *ExtractMin* operation is the one whose domain ends earliest. So any node that remains in P can match the same future y-nodes as x (by convexity), but maybe even more. And hence, it is reasonable to extract x and keep the others.

In the first pass the algorithm ignores the lower bound capacities and finds a generalized matching in the graph $G_{L=0} = (V, E, C_{L=0})$, where $C_{L=0}(y_i) = [0, U_i]$ for all i and $C_{L=0}(x_j) = [1, 1]$ for all j. $G_{L=0}$ is the same as G except that the lower bound capacities of all y nodes are zero. This generalized matching is constructed as follows: The y nodes are traversed from $y_1, \cdots, y_{n'}$. When y_i is reached, all of the x nodes that are connected to y_i, but not to any node with a smaller index, are inserted into the queue; they are now candidates for matching. Then y_i is matched with up to U_i nodes from P (less if P does not contain that many nodes). If, while processing y_i, the algorithm extracts a node from the queue which is not connected to y_i, it reports failure (cf. Lemma 3).

In the second pass, the algorithm makes another traversal of the y nodes in the same order, but this time it ignores the upper bounds and constructs a generalized matching in the graph $G^{U=\infty} = (V, E, C^{U=\infty})$, where $C^{U=\infty}(y_i) = [L_i, \infty]$ for all i and $C^{U=\infty}(x_j) = [1, 1]$ for all j. In this matching, each y node is matched with the minimal number of x nodes such that its lower capacity bound is respected and all x nodes which are not connected to y nodes with higher indices are matched. If the queue becomes empty while the algorithm tries to fulfil the lower capacity bound L_i for node y_i, the algorithm reports failure (see Lemma 6).

The matching found in the first pass is used during the second pass to determine the order in which nodes are inserted into the queue. We will show that if a generalized matching exists, then there also exists a generalized matching such that each y_i is matched only with nodes that were matched in the first pass with nodes from $y_1, \cdots, y_i$ (cf. Lemma 4). Hence, when y_i is reached in the second pass, the nodes that were matched with it in the first pass are inserted into the queue as candidates for matching with it.

The matching found in the second pass may violate the upper capacity bounds in G. The third pass corrects this by traversing the y nodes from $y_{n'}$ to y_1 and shifting these excesses to y nodes with lower indices.

For our running example, we show in the table below the mate of x_j after the respective pass of the algorithm.

j	1	2	3	4	5	6
pass 1	1	1	1	2	2	3
pass 2	1	2	2	2	3	4
pass 3	1	1	2	2	3	4

Lemma 3 *If the algorithm reports failure in the first pass, then there is no generalized matching in $G_{L=0}$. And hence, there is also none in G.*

Proof Suppose the algorithm reports failure in iteration i after extracting a node x_j (observe that any extraction in iteration $n'+1$ causes failure.) Then in iteration $i-1$ it only extracts x nodes that are not connected to y_i, and P contains x_j afterwards. Let i' be the maximum iteration before $i-1$ such that an x node connected to y_i is extracted or P becomes empty during iteration i' (see Figure 3). If no such iteration exists, choose $i'=0$. Let X denote the x nodes extracted in iterations $i'+1, \cdots, i-1$. Since P never becomes empty during these iterations, we have $|X| = \sum_{k=i'+1}^{i-1} U_k$, i.e. X exhausts the capacities of the nodes in $Y = \{y_{i'+1}, \cdots, y_{i-1}\}$.

The nodes in X are not connected to y_i or a y node to its right. We show that there is also no connection to $y_{i'}$ or a y node to its left. Suppose otherwise, i.e. $x_\ell \in X$ is connected to $y_{i''}$ with $i'' \le i'$. As x_ℓ was inserted to P in iteration i' or earlier and is extracted in a later iteration, P was never empty in iteration i'. By the choice of i', this implies that a node $x_{j'}$ connected to y_i was extracted in iteration i'. As x_l was removed later, it must also be connected to y_i, a contradiction to the choice of i'. Thus all neighbors of the nodes in X are in Y. A similar argument proves that the node x_j, which caused the failure in iteration i, is only connected to nodes in Y.

(* Input: (1) Ranges of assignment variables $D_j = [\underline{D}_j, \overline{D}_j]$ for each x_j, $1 \leq j \leq n$ ***)**
(* (2) A capacity requirement $[L_i, U_i]$ for each $1 \leq i \leq n'$. ***)**
(* Output: A generalized matching if it exists, *failure* otherwise. ***)**

(* 1st pass: find a matching in $G_{L=0}$ (encoded in m_1) ***)**
$P \leftarrow []$ **(*** priority queue containing x nodes sorted according to $\overline{D}$ ***)**
$j \leftarrow 0$; $U_{n'+1} \leftarrow \infty$
for $i = 1$ to $n' + 1$ **do**
  forall x_h with $\underline{D}_h = i$ **do** $P.Insert\ x_h$
  $u \leftarrow 0$
  while P is not empty and $u < U_i$ **do**
    $j \leftarrow j + 1$; $x_{f(j)} \leftarrow P.ExtractMin$
    $m_1[f(j)] \leftarrow i$; $u \leftarrow u + 1$
    if $\overline{D}_{f(j)} < i$ **then** report failure
    (* No failure implies $i \in D_{f(j)}$ ***)**
  end
  $\beta_i \leftarrow j$
endfor

(* 2nd pass: find a matching in $G^{U=\infty}$ and ensure feasibility for G ***)**
$j \leftarrow 0$; $\alpha_0 \leftarrow 0$
for $i = 1$ to n' **do**
  forall x_h with $m_1[h] = i$ **do** $P.Insert\ x_h$
  for $\ell = 1$ to L_i **do**
    if P is empty **then** report failure
    $j \leftarrow j + 1$; $x_{g(j)} \leftarrow P.ExtractMin$
    match $y_i \leftrightarrow x_{g(j)}$ **(*** $m_1[g(j)] \leq i$ ***)**
  endfor
  while P is not empty and $P.MinPriority < i + 1$ **do**
    $j \leftarrow j + 1$; $x_{g(j)} \leftarrow P.ExtractMin$
    match $y_i \leftrightarrow x_{g(j)}$ **(*** $m_1[g(j)] \leq i$ ***)**
  end
  $\alpha_i \leftarrow j$
endfor

(* 3rd pass: transform the matching from 2nd pass into a matching in G ***)**
for $i = n'$ to 1 **do**
  if y_i has currently h mates s.th. $e := h - U_i > 0$ **then**
    for $k = 1$ to e **do**
      choose a current mate x_{j_k} of y_i with $i' := m_1[j_k] < i$
      match $y_{i'} \leftrightarrow x_{j_k}$ **(*** y_i looses x_{j_k} ***)**
    endfor
  endif
endfor

Figure 2: Algorithm for generalized matching in a convex capacitated bipartite graph

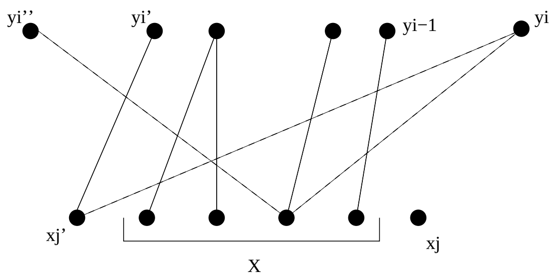

Figure 3: Illustration of the proof of Lemma 3

Since a generalized matching in $G^{U=\infty}$ would have to match the nodes in Y with at least $|X|+1$ nodes, but can only match them with $|X|$ different nodes, such a matching cannot exist. ∎

For any matching M and subset $Y \subseteq \{y_1, \cdots, y_{n'}\}$, let $M(Y) = \bigcup_{y \in Y} M(y)$, i.e. $M(Y)$ is the set of all x nodes that are matched with nodes of Y.

Lemma 4 *If there is a generalized matching M' in G, then there is also a generalized matching M with the following property: For all edges $\{x_j, y_i\} \in M$, we have $m_1[j] \leq i$.*
M can be chosen such that $|M(y_i)| = |M'(y_i)|$ for $i = 1, \cdots, n'$. And hence, it is impossible to match $\{y_1, \cdots, y_i\}$ with more than β_i nodes[2].

Proof Let M' be a generalized matching in G, and suppose it does not have the desired property. Let i be minimal such that there is an edge $\{x_j, y_i\}$ with $k = m_1[j] > i$. Not all nodes x_ℓ with $m_1[\ell] = i$ can be matched with y_i in M', otherwise neither the queue nor the capacity of y_i would have been exhausted after iteration i in the first pass. So there is an edge $\{x_{j'}, y_{i'}\} \in M'$ with $i' \neq i$ and $m_1[j'] = i$ (see Figure 4). By the choice of i we have $i' > i$. When $x_{j'}$ was extracted in the first pass, x_j was also in the queue. So we conclude that x_j is also connected to $y_{i'}$. Thus we can swap the mates of x_j and $x_{j'}$ and obtain a new generalized matching M''. M'' does not violate the property for the nodes $y_1, \cdots, y_{i-1}$ and the number of violations at y_i is less than in M'. This shows that we can transform the matching until we eventually obtain a matching M with the desired property.
The last statement follows from the fact that $\beta_i = |\{x_j \mid m_1[j] \leq i\}|$. ∎

Lemma 5 *If the algorithm does not report failure in iteration i of the second pass, then for any generalized matching M in G, $|M(\{y_1, \cdots, y_i\})| \geq \alpha_i$.*

[2]We will refer to the β_i values in Section 5.

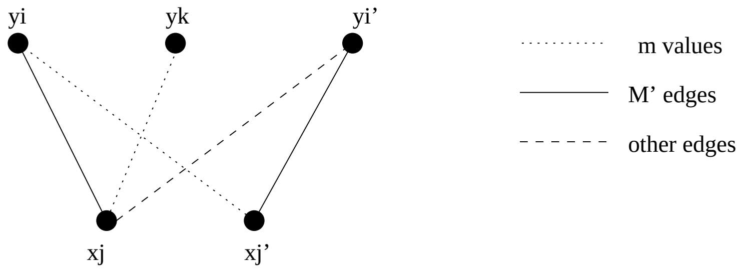

Figure 4: Illustration of the proof of Lemma 4

Proof By induction on i. For $i = 0$ and $i = 1$ the claim is easy to verify. So let us assume that it holds for $i' = 1, \cdots, i-1$ and prove it for i. If the body of the while-loop is not executed, we have $\alpha_i = \alpha_{i-1} + L_i$, and applying the induction for $i' = i-1$ immediately proves the claim. So suppose that nodes are extracted in the while-loop, which implies that no x that is extracted in iteration i is connected to y_{i+1}. Let i' be the maximum iteration before i such that a node connected to y_{i+1} is removed. If no such iteration exists, choose $i' = 0$. Let X denote the x nodes extracted in the iterations $i'+1, \ldots, i$, and let $Y = \{y_{i'+1}, \cdots, y_i\}$. A similar argument as in the proof of Lemma 3 shows that the neighbors of any node $x \in X$ are contained in Y. Thus $|M(Y)| \geq |X| = \alpha_i - \alpha_{i'}$. By the induction hypothesis for i', $|M(\{y_1, \cdots, y_i\})| \geq \alpha_{i'} + |M(Y)| = \alpha_i$. ∎

Lemma 6 *If the algorithm reports failure in the second pass, then there is no generalized matching in G.*

Proof Suppose that the algorithm reports failure in iteration i, although a generalized matching M exists. By Lemma 4, we can assume that for any edge $\{x_j, y_k\}$ in M we have $m_1[j] \leq k$. So $y_1, \cdots, y_i$ are matched only with nodes that have been inserted into P so far. From the previous lemma we can conclude that there must be at least $\alpha_{i-1} + L_i$ such nodes. But since the algorithm reports failure, this is not the case, a contradiction to the existence of M. ∎

Lemma 7 *If the algorithm does not report failure, it constructs a generalized matching in G.*

Proof If the second pass succeeds, we have a generalized matching in $G^{U=\infty}$, because we fulfil the lower capacity bounds of every y node (cf. the for-loop), and we match every x node with a neighbor on the y side (see the while-loop). So the only problem is the upper capacity bounds of the y nodes. The third pass takes care of these. Observe that this pass sweeps over the y nodes from right to left and distributes the excess mates of a node y_i only to y nodes with lower indices, so that it cannot increase the number of mates of a y node after it was processed. Furthermore, if x_j is matched with y_i at some point in time then $m_1[j] \leq i$. Since we have equality for at most U_i nodes, we can always select e excessive mates to distribute, if necessary. Suppose a node x_j is removed from y_i and matched with $y_{i'}$ in the third pass, where $i' = m_1[j]$. From the first pass, it is easy to see that $i' \in D_j$, i.e., x_j and $y_{i'}$ are connected. ∎

Implementing the algorithm in linear time (for normalized domains)

Now we discuss some implementation details of the algorithm. First we show how to implement it in time $O(n' + n \log n)$, and then we refine this implementation to obtain $O(n' + n)$ time. In the first variant we use a binary heap of size n to implement the priority queue P. Then the operations *Insert* and *ExtractMin* take time $O(\log n)$ and *MinPriority* runs in constant time. Before we run the algorithm we sort the x nodes according to their lower range endpoints, which takes time $O(n + n')$ because all domains are in $[1, n']$. This sorting allows us to determine efficiently the nodes that have to be inserted into P in each iteration of the first pass. Recall that the order by which the x nodes are extracted in the first pass determines the order by which they are inserted in the second pass. So the two passes can be implemented in time $O(n' + n \log n)$. The third pass takes linear time. To see this we notice that every x node changes its mate at most once. So we maintain for every y node two separate lists for the mates that it received in the second and the third pass respectively. Thus we can make sure that we process every x node only once.

In order to shave off the logarithmic factor in the running time, we have to find a faster implementation of the priority queue. As in [4] we simulate the priority queue by creating an instance of the offline-min problem [1, Chapter 4.8], which can be solved in linear time using a special union-find data structure [2]. (The offline-min algorithm needs a sorting of the x's according to their upper interval endpoints, which can be computed in time $O(n + n')$.)

Except for failure detection, the algorithm does not use the mapping f in the first pass at all. So the whole sequence σ of *Insert* and *ExtractMin* operations can be computed without knowing the results of the extractions. Checking if P is empty can be done by counting the number of insertions and comparing it with the number of extractions (which is equal to j). Failure detection can be done after the mapping f has been computed.

We give a brief description of the offline-min algorithm. When it is applied to a sequence σ it determines for every insertion the corresponding extraction. Let $E_1, \cdots, E_n$ be the extract operations in the order in which they occur in σ. The node x_j with minimum priority is the output of the first extraction E_k that follows its insertion. After deleting E_k from σ we process the x with next higher priority, and so on. We implement this with a union-find structure that maintains a partition $\mathcal{P}$ of $E_1, \cdots, E_n$. Every set in $\mathcal{P}$ has the form $\{E_h, E_{h+1}, \cdots, E_k\}$, where all extractions except for E_k have been deleted. Suppose we process x_j and let E_s be the first extraction in the original sequence after the insertion of x_j. Observe that E_s may have been deleted already. To determine the extraction E_k for x_j, we simply have to find the set S in $\mathcal{P}$ containing E_s. Deleting E_k amounts to uniting S with the set of E_{k+1}.

We cannot use the offline-min algorithm directly for the second pass, because we need to know the result of the *MinPriority* operation in the while-loop online. But a slight enhancement will do the job (see Figure 5). We initialize our data structures as above but we only create a set for the mandatory extractions that are made in the for-loop. When we find a tentative extraction E_k for x_j, we verify that E_k occurs in iteration $i \leq \overline{D}_j$. If not, then we know that x_j is extracted by the while-loop in iteration $\overline{D}_j$, and we do not delete E_k. We want to point out that we detect failure if there is a mandatory extraction for which no corresponding insertion is found. Assuming w.l.o.g. that $\sum_{i=1}^{n'} L_i \leq n$, the algorithm runs in time $O(n' + n)$.

4 Finding the SCCs of the oriented intersection graph

As described in Section 1, we construct an oriented graph $\vec{G}$ from the intersection graph and the generalized matching M (see Figure 6). The edges which are not within a strongly connected component of $\vec{G}$ describe inconsistent assignments of values to variables. Mehlhorn and Thiel [4] gave an algorithm that finds the SCCs of a simpler oriented intersection graph in $O(n + n')$ time. Their graph is simpler than $\vec{G}$ in that it does not contain the additional node s, which violates the convexity property of the graph because there is an edge from s to each node in $\{y_i : |M(y_i)| > L_i\}$, and this

$k \leftarrow 1$**(*** E_k will always be the next mandatory extraction ***)**
for $i = 1$ to n' **do**
  forall x_j with $m_1[j] = i$ **do**
    (* x's inserted in iteration i (before E_k) ***)**
    $s[j] \leftarrow k$
  endfor for $\ell = 1$ to L_i **do**
    create set $\{E_k\}$ labelled $\mathcal{L}_k^{(i,\ell)}$ **(*** for mandatory extraction ℓ in iter. i ***)**
    $k \leftarrow k+1$
  endfor
endfor
create set $\{E_k\}$ labelled with $\mathcal{L}_k^{(n'+1,0)}$ **(*** dummy extraction ***)**
forall x_j sorted in ascending order according to $\overline{D}$ **do**
  $S \leftarrow \text{find}(E_{s[j]})$; let $\mathcal{L}_k^{(i,\ell)}$ be its label
  if $i \leq \overline{D}_j$ **then**
    x_j is removed by extraction ℓ in the for-loop of iteration i
    unite S with the set S' containing E_{k+1} and
      label the union with the former label of S'
  else
    x_j is extracted by the while-loop in iteration $\overline{D}_j$
  endif
endfor

Figure 5: Enhanced offline-min algorithm for the second pass

set is not necessarily consecutive. We first use their algorithm to compute the SCCs of the graph $\vec{G}\backslash s$. We can do this despite the fact that now a y node may be matched with more than one x node by merging its neighbors into one x node (note that the neighbors of this merged x node still form an interval of the y-nodes). Let C_s denote the SCC of the node s in $\vec{G}$. An SCC of G which is different from C_s is also an SCC of $\vec{G}\backslash s$. C_s, however, may be composed of zero or more SCCs of $\vec{G}\backslash s$. We wish to find these SCCs and merge them to obtain C_s. Each of these SCCs has the property that it can reach s and can be reached from s. For each SCC C of $\vec{G}\backslash s$ we compute two flags, *reached_from_s*$[C]$ and *reaches_s*$[C]$, and merge all components for which both flags are set to true into C_s.

We will use the following notation. For each SCC C of $\vec{G}\backslash s$, let *min_y*$[C] = \min\{i | y_i \in C\}$ and *max_y*$[C] = \max\{i | y_i \in C\}$ be the lowest and highest indices of y nodes in C. Moreover, let *reaches_left*$[C] = \min\{\underline{D}_j | x_j \in C\}$ and *reaches_right*$[C] = \max\{\overline{D}_j | x_j \in C\}$ be the minimum and maximum indices of y nodes that can be reached from C.

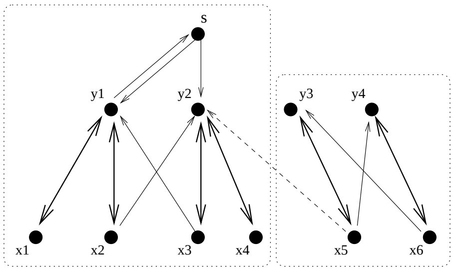

Figure 6: The oriented intersection graph for our running example. It has two SCCs: $\{s, x_1, x_2, x_3, x_4, y_1, y_2\}$ and $\{x_5, x_6, y_3, y_4\}$. The dashed edge is inconsistent.

The following lemma states that the y nodes reachable by an edge from C form a contiguous interval.

Lemma 8 *C can reach every y_i with $reaches\_left[C] \leq i \leq reaches\_right[C]$ by one edge from an x node in C.*

Proof
Case 1: $reaches\_left[C] \leq i \leq min\_y[C]$. Let $x_\ell \in C$ be such that $\underline{D}_\ell = reaches\_left[C]$ and let y_k be the node matched with x_ℓ. Then $\underline{D}_\ell \leq i \leq k$ and by convexity $i \in D_\ell$.
Case 2: $max\_y[C] \leq i \leq reaches\_right[C]$ is symmetric to Case 1.
Case 3: $min\_y[C] < i < max\_y[C]$. Since C is strongly connected, there is a path $(y_{i_1}, x_{j_1}) \circ \cdots \circ (x_{j_{k-1}}, y_{i_k})$ in C where $i_1 = min\_y[C]$ and $i_k = max\_y[C]$. Then there exists $1 \leq \kappa < k$ such that $i_\kappa < i \leq i_{\kappa+1}$. This implies that $\underline{D}_{j_\kappa} \leq i_\kappa < i \leq i_{\kappa+1} \leq \overline{D}_{j_\kappa}$, and by convexity $i \in D_{j_\kappa}$, i.e., y_i is a neighbor of x_{j_κ}. ∎

Now we will explain how to determine whether s can reach a component C. If this is the case then there is a path Q from s to some node y_i in C. And hence, if we delete the first edge of Q, we obtain a path P from a node $y_{i'}$ to y_i in $\vec{G} \backslash s$. In the lemma below, we state that this implies the existence of path P' from $y_{i'}$ to y_i that traverses the SCCs of $\vec{G} \setminus s$ in monotonous manner, either from left to right (cf. Condition (1)) or from right to left (cf. Condition (2)):

Lemma 9 *If there is a path from y_{i_1} to y_{i_k} in $\vec{G} \setminus s$ then there is a path $P = (y_{i_1}, x_{j_1}) \circ (x_{j_1}, y_{i_2}) \circ \cdots \circ (x_{j_{k-1}}, y_{i_k})$ in $\vec{G} \backslash s$ such that if we let C_κ denote the component of y_{i_κ} for each $1 \leq \kappa \leq k$, then one of the following holds:*

$$\forall_{1 \leq \kappa < k}\ min\_y[C_\kappa] \leq min\_y[C_{\kappa+1}] \quad \text{right-monotony} \tag{1}$$

$$\forall_{1 \leq \kappa < k}\ max\_y[C_\kappa] \geq max\_y[C_{\kappa+1}] \quad \text{left-monotony} \tag{2}$$

Proof If P visits more than one component, let κ be the smallest value such that $C_\kappa \neq C_{\kappa+1}$. Assume that Condition (1) holds for κ. If P does not fulfil Condition (1), then there must be three components C_1, C_2, C_3 which are visited consecutively by P in that order such that $min\_y[C_1] < min\_y[C_2]$ and $min\_y[C_3] < min\_y[C_2]$. We will show that there is a path which skips C_2 and goes directly from C_1 to C_3.
If $min\_y[C_3] < min\_y[C_1] < min\_y[C_2]$ then by Lemma 8, C_2 can reach C_1 by a single edge, which closes a cycle that visits both of them. This means that $C_1 = C_2$, in contradiction to the assumption that they are distinct.
We therefore know that $min\_y[C_1] < min\_y[C_3] < min\_y[C_2]$. Then by Lemma 8, C_1 can reach C_3 by one edge. So we can construct a path from y_{i_1} to y_{i_k} which visits the same components as P except that it goes from C_1 directly to C_3 without going through C_2. Since this decreases the number of transitions between components along the path, we can obtain a monotonous path by a finite number of such modifications.

If Condition (1) does not hold for the first transition in P between two distinct components $C_\kappa, C_{\kappa+1}$ then we show that Condition (2) must hold and then it is clear that we can similarly find a path that fulfils Condition (2) for all $1 \leq \kappa < k$.

Assume that neither condition holds. Then $min\_y[C_{\kappa+1}] < min\_y[C_\kappa]$ and $max\_y[C_\kappa] < max\_y[C_{\kappa+1}]$. By Lemma 8, $C_{\kappa+1}$ reaches back to C_κ, a contradiction. ∎

The algorithm in Figure 7 performs a left to right scan to mark all the components of $\vec{G}\backslash s$ that can be reached from s via a right-monotonous path. In the initialization phase it marks all components that can be reached from s by a single edge. During the scan it maintains the index *max_y_reached_from_s* which describes the rightmost node that can be reached from a scanned marked component by one edge. The running time is $O(n')$. A symmetric algorithm can be used to find left-monotonous paths.

Lemma 10 *The algorithm in Figure 7 sets reached_from_s$[C]$ to true iff s can reach C by a right-monotonous path.*

Proof It is easy to see that if a component becomes marked, then it can be reached from s. We will now prove the converse. Consider a path $P = (s, y_{i_1}) \circ \cdots \circ (y_{i_{k-1}}, x_{j_{k-1}}) \circ (x_{j_{k-1}}, y_{i_k})$ such that P without its first edge is right-monotonous. We will show the following claim by induction on the length of P: after the i_k-th iteration, the component C_k of y_{i_k} is marked "reached from s".
For $|P| = 1$: $P = (s, y_{i_1})$ and the component of y_{i_1} is marked before the scan.
For $|P| > 1$: By the induction hypothesis, *reached_from_s*$[C_{k-1}]$ is set to true

```
forall SCCs C of G⃗\s do
    reached_from_s[C] ← false
end
for i = 1 to n' do
    if (s, y_i) exists then reached_from_s[C[y_i]] ← true
endfor
(* Scan from left to right and find the maximum y's reached from s *)
max_y_reached_from_s ← 0
for i = 1 to n' do
    C ← C[y_i]
    (* Check whether C can be reached from s via a right-monotonous path *)
    if max_y_reached_from_s ≥ i then
        reached_from_s[C] ← true
    endif
    (* Advance max_y_reached_from_s if possible *)
    if reached_from_s[C] then
        max_y_reached_from_s ← max{max_y_reached_from_s, reaches_right[C]}
    endif
endfor
```

Figure 7: Algorithm for finding SCCs that can be reached from s via a right-monotonous path

after iteration $min\_y[C_{k-1}]$. [3] Since $x_{j_{k-1}}$ is in C_{k-1}, $reaches\_right[C_{k-1}] \geq i_k \geq min\_y[C_k]$. In iteration $min\_y[C_{k-1}]$, $max\_y\_reached\_from\_s$ is set to at least $min\_y[C_k]$. By Lemma 9, we know that iteration $min\_y[C_k]$ does not preceed iteration $min\_y[C_{k-1}]$. Since the index $max\_y\_reached\_from\_s$ can only increase, we get that in iteration $min\_y[C_k]$, the component of y_{i_k} will be marked as reached from s. ∎

In order to determine whether a component C can reach s, we can use the same approach: if we delete the last edge of a path in $\vec{G}$ from a node in C to s, we obtain a path in $\vec{G} \setminus s$. So we can mark all components that can reach s by two scans in time $O(n')$. The complete algorithm is given in Figure 8. Finally, we merge all components that can reach s and can be reached from s into the single component C_s.

5 Narrowing the Bounds of the x-ranges

Let S denote the set of all solutions of the constraint, which is defined in the introduction, and for $j = 1, \ldots, n$ let S_j be the projection of S onto the jth component. We will discuss how to compute the values $\overline{S}_1, \ldots, \overline{S}_n$ in time $O(n + n')$, a symmetric procedure can be used for the lower endpoints of the narrowed ranges.

Consider a node x_j and let C be its SCC in $\vec{G}$. By Lemma 2, a value $i \in D_j$ is contained in S_j iff $y_i \in C$. Thus $\overline{S}_j$ is the index of the rightmost y node in C that is connected to x_j. Suppose the y nodes $y_{i_1}, \ldots, y_{i_k}$ in C are sorted such that $i_1 < \cdots < i_k$. Then $\overline{S}_j = i_\kappa$ where $i_\kappa \leq \overline{D}_j < i_{\kappa+1}$ (and $i_{k+1} = n' + 1$). So let us further assume that the x nodes $x_{j_1}, \ldots, x_{j_l}$ in C are sorted such that $\overline{D}_{j_1} \leq \cdots \leq \overline{D}_{j_l}$. Then we can determine $\overline{S}_j$ for every x_j in C by merging the sorted sequences $(i_1, \ldots, i_k)$ and $(\overline{D}_{j_1}, \ldots, \overline{D}_{j_l})$ in time $O(k + l)$.

We need to say how these sorted sequences are constructed. For the y's this can be done with bucketsort. We have a bucket for each component, and for $i = 1, \ldots, n'$, we append y_i to the bucket corresponding to its component. This takes time $O(n')$ and constructs the y-sequences for all SCCs of $\vec{G}$. Since we have a global sorting of the x's according to their upper range endpoints, we can use the same approach to sort them in time $O(n)$.

For the first SCC of our example we would merge the y sequence $(1, 2)$ with the x upper bounds $(2_{x_1}, 3_{x_2}, 3_{x_3})$ and narrow the domains of x_2 and x_3 from $[1, 3]$ to $[1, 2]$.

[3] We may assume w.l.o.g. that if P visits a component C then it also visits the node $y_{min\_y[C]}$.

$SCCs \leftarrow$ Find_SCCs($\vec{G} \setminus s$)
forall $C \in SCCs$ **do**
  $reached\_from\_s[C] \leftarrow false$
  $reaches\_s[C] \leftarrow false$
end
for $i = 1$ to n' **do**
  if (s, y_i) exists **then** $reached\_from\_s[C[y_i]] \leftarrow$ true
  if (y_i, s) exists **then** $reaches\_s[C[y_i]] \leftarrow$ true
end
(* Scan from start to end and find the maximum y's connected to s ***)**
$max\_y\_reached\_from\_s \leftarrow 0$
$max\_y\_reaches\_s \leftarrow 0$
for $i = 1$ to n' **do**
  $C \leftarrow C[y_i]$
  (* Check whether C can be reached from s ***)**
  if $max\_y\_reached\_from\_s \geq i$ **then** $reached\_from\_s[C] \leftarrow$ true
  if $reached\_from\_s[C]$ **then**
    $max\_y\_reached\_from\_s \leftarrow \max\{max\_y\_reached\_from\_s, reaches\_right[C]\}$
  (* Check whether C can reach s ***)**
  if $reaches\_left[C] \leq max\_y\_reaches\_s$ **then** $reaches\_s[C] \leftarrow$ true
  if $reaches\_s[C]$ **then** $max\_y\_reaches\_s \leftarrow \max\{max\_y\_reaches\_s, i\}$
end for
(* Scan from end to start and find the minimum y's connected to s ***)**
$min\_y\_reached\_from\_s \leftarrow n' + 1$
$min\_y\_reaches\_s \leftarrow n' + 1$
for $i = n'$ to 1 **do**
  $C \leftarrow C[y_i]$
  (* Check whether C can be reached from s ***)**
  if $min\_y\_reached\_from\_s \leq i$ **then** $reached\_from\_s[C] \leftarrow$ true
  if $reached\_from\_s[C]$ **then**
    $min\_y\_reached\_from\_s \leftarrow \min\{min\_y\_reached\_from\_s, reaches\_left[C]\}$
  (* Check whether C can reach s ***)**
  if $reaches\_right[C] \geq min\_y\_reaches\_s$ **then** $reaches\_s[C] \leftarrow$ true
  if $reaches\_s[C]$ **then** $min\_y\_reaches\_s \leftarrow \min\{min\_y\_reaches\_s, i\}$
end for
(* Merge all components that are strongly connected through s ***)**
$C \leftarrow \{s\}$
foreach $C' \in SCCs$ **do**
  if $reaches\_s[C'] \wedge reached\_from\_s[C']\}$ **then** $C \leftarrow C \circ C'$
end for

Figure 8: Algorithm for finding SCCs of the oriented intersection graph

6 Narrowing the Bounds of the Count Variables

This section and the next deal with the projections of S onto its components $S_{n+1}, \cdots, S_{n+n'}$. We show how to compute lower and upper bounds for the values in S_{n+i} for each $i = 1, \cdots, n'$. In this section we show a very fast algorithm that narrows the domains of the count variables but does not achieve bound consistency. In the next section we show a different algorithm that achieves bound consistency for the count variables. It has the same linear-time asymptotic complexity but uses more elaborate data structures that can be expected to make it slower in practice.

For the rest of this section we consider a fixed graph G. We assume that the generalized matching algorithm has computed the α_i and β_i values for G and terminated without reporting failure. Now we show that in any generalized matching, the number of x nodes matched to each y_i are in a certain range that we can compute in linear time.

Lemma 11 *Let M be a generalized matching and for $i = 1, \cdots, n'$ let $\mu_i = |M(\{y_i\})|$. Then for all i*

$$\max\left(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \mu_j\right) \leq \mu_i \leq \min\left(U_i, n - \alpha_{i-1} - \sum_{j=i+1}^{n'} \mu_j\right) \quad (*)$$

Proof By the choice of the μ's we have $|M(\{y_1, \cdots, y_{i-1}\})| = n - \sum_{j=i}^{n'} \mu_j$. By Lemmas 4, 5 we know that $\alpha_{i-1} \leq |M(\{y_1, \cdots, y_{i-1}\})| \leq \beta_{i-1}$. Therefore $\alpha_{i-1} \leq n - \sum_{j=i}^{n'} \mu_j \leq \beta_{i-1}$ or $n - \beta_{i-1} \leq \sum_{j=i}^{n'} \mu_j \leq n - \alpha_{i-1}$. And hence $n - \beta_{i-1} - \sum_{j=i+1}^{n'} \mu_j \leq \mu_i \leq n - \alpha_{i-1} - \sum_{j=i+1}^{n'} \mu_j$. ∎

This motivates the following definition. We call a sequence $(\mu_s, \cdots, \mu_{n'})$ a *legal count choice* iff inequality (*) is satisfied for all i in $[s, n']$. The Lemma above implies that any generalized matching M in G induces the legal choice $(|M(y_1)|, \ldots, |M(y_{n'})|)$. This allows us to compute for any y_i a lower bound l_i and an upper bound u_i on $|M(y_i)|$. Unfortunately there exist examples where the bounds are not tight. Choosing $D_1 = [1, 3], D_2 = [2, 2]$ and $E_1 = E_2 = E_3 = [0, 1]$ is such an example. The lower endpoint of E_2 will not be narrowed to 1 by our algorithm.

Apart from these bounds the algorithm below determines two count choices. The algorithm maintains the invariant that in iteration i both $(\kappa_i, \ldots, \kappa_{n'})$ and $(\lambda_i, \ldots, \lambda_{n'})$ are legal count choices such that for any legal count choice

$(\mu_i, \ldots, \mu_{n'})$ we have $\sum_{j=i}^{n'} \kappa_j \leq \sum_{j=i}^{n'} \mu_j \leq \sum_{j=i}^{n'} \lambda_j$. As all sums that appear in the algorithm can be computed incrementally, the running time is $O(n')$.

for $i = n'$ to 1 **do**

$l_i \leftarrow \max\left(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \lambda_j\right)$; $u_i \leftarrow \min\left(U_i, n - \alpha_{i-1} - \sum_{j=i+1}^{n'} \kappa_j\right)$

$\kappa_i \leftarrow \max\left(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \kappa_j\right)$; $\lambda_i \leftarrow \min\left(U_i, n - \alpha_{i-1} - \sum_{j=i+1}^{n'} \lambda_j\right)$

endfor

We will now prove by induction on i that the algorithm computes the correct bounds for the count variables and that the invariant holds. For $i = n'$ Lemma 11 implies that $l_{n'}$ and $u_{n'}$ are lower and upper bounds. Since all sums are empty, we have $\kappa_{n'} = l_{n'}$ and $\lambda_{n'} = u_{n'}$, and hence, the claimed invariant holds.

We come to the induction step, we assume that our claim holds for $i+1$ and verify it for i. Looking at the left-hand side of inequality (*), we see that the minimum legal value for μ_i is obtained if $\sum_{j=i+1}^{n'} \mu_j$ is set to its largest possible value. Applying the invariant for $i+1$ this sum is maximized by the legal choice $(\lambda_{i+1}, \ldots, \lambda_{n'})$. So by Lemma 11, the algorithm computes a lower bound l_i. A similar argument holds for the upper bound u_i. By the definition of a legal choice, extending our two count choices by κ_i and λ_i respectively yields two legal choices again.

Fix a legal choice $(\mu_i, \ldots, \mu_{n'})$. What remains to prove are the two inequalities on the sum of the μ's.

$$\begin{aligned}
\sum_{j=i}^{n'} \mu_j &= \mu_i + \sum_{j=i+1}^{n'} \mu_j \overset{\text{L 11}}{\geq} \max(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \mu_j) + \sum_{j=i+1}^{n'} \mu_j \\
&= \max(L_i + \sum_{j=i+1}^{n'} \mu_j, n - \beta_{i-1}) \\
&\overset{\text{IH}}{\geq} \max(L_i + \sum_{j=i+1}^{n'} \kappa_j, n - \beta_{i-1}) \\
&= \max(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \kappa_j) + \sum_{j=i+1}^{n'} \kappa_j \\
&= \kappa_i + \sum_{j=i+1}^{n'} \kappa_j = \sum_{j=i}^{n'} \kappa_j
\end{aligned}$$

An analogous computation shows $\sum_{j=i}^{n'} \mu_j \leq \sum_{j=i}^{n'} \lambda_j$. So we have just shown the following lemma:

Lemma 12 *Suppose that $S \neq \emptyset$ and let $\ell_1, \cdots, \ell_{n'}$ and $u_1, \cdots, u_{n'}$ be the values computed by our algorithm. Then $\underline{S}_{n+i} \geq \ell_i$ and $\overline{S}_{n+i} \leq u_i$ holds for $i = 1, \cdots, n$.*

If we apply the algorithm to our running example, it computes the values listed below:

i	l_i	u_i	κ_i	λ_i	$\sum_{j=i}^{n'} \kappa_i$	$\sum_{j=i}^{n'} \lambda_i$
4	1	1	1	1	1	1
3	1	1	1	1	2	2
2	1	2	1	2	3	4
1	2	3	3	2	6	6

E_1, then, is narrowed from $[1, 3]$ to $[2, 3]$ and $E_2, \ldots, E_4$ remain as they were.

7 Bound Consistency for the Count Variables

In this section we show a linear-time algorithm that achieves bounds consistency for the count variables. The following lemma extends the result of Lemma 4.

Lemma 13 *Let $m_2[j]$ be the index of the y node that x_j was matched with in the second pass of the algorithm. If there is a generalized matching M' in G, then there is also a generalized matching M with the following property: For all edges $\{x_j, y_i\} \in M$, we have $m_1[j] \leq i \leq m_2[j]$ and for all $1 \leq i \leq n'$, $|M(y_i)| = |M'(y_i)|$.*

Proof By Lemma 4, we can assume that for all edges $\{x_j, y_i\} \in M'$, we have $m_1[j] \leq i$. Assume that M' does not fulfil the condition on the $m_2[j]$'s and let i be the minimum index such that there is an edge $\{x_j, y_i\} \in M'$ with $i > m_2[j] = k$. Since x_j is matched in M' with y_i, $\overline{D}(x_j) \geq i > k$ (see Figure 9). This implies that y_k was matched in the second pass with L_k nodes. In M', y_k is not matched with x_j so it must be matched with a node $x_{j'}$ with $m_2[j'] \neq k$. By the choice of i, we know that $k' = m_2[j'] > k$, and as M' fulfils Lemma 4 we know that $m_1[j'] \leq k$. Since $k = m_2[j]$ and $m_1[j'] \leq k \leq m_2[j']$, we get that both x_j and $x_{j'}$ were in P during the extractions of the kth iteration of the second pass. Since x_j was extracted first, we get that $\overline{D}(x_{j'}) \geq \overline{D}(x_j) \geq i$, hence there is an edge $(x_{j'}, y_i)$. We can therefore replace in M' the edges $(x_{j'}, y_k)$ and (x_j, y_i) by the edges $(x_{j'}, y_i)$ and (x_j, y_k). We get a new matching such that $|M(y_i)| = |M'(y_i)|$ for all $1 \leq i \leq n'$ and the matching mates of y_k still do not violate the claim. As for y_i, let d_i be the sum of the differences $i - m_2[\ell]$ over all violating matching mates x_ℓ of y_i. Since $k' > k$, d_i decreases and we can continue performing such substitutions until $d_i = 0$, which means that there are no violations at y_i. We can then advance to y_{i+1} and continue making such adjustments until we obtain the matching M. ∎

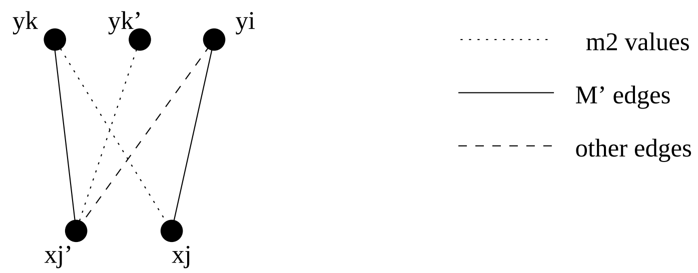

Figure 9: Illustration of the proof of Lemma 13

The algorithm in Figure 10 traverses the y nodes from right to left and determines for each y_i the minimum number of nodes that it must be matched with in a generalized matching M that fulfils the conditions of Lemma 13. It uses a priority queue P in which the x nodes are sorted according their m_1 values. For each y_i, it first inserts into P all nodes x_κ with $m_2[\kappa] = i$; these nodes are candidates for matching with y_i. By the time the algorithm finishes handling each $y_{i'}$, it extracts from P as many nodes as $y_{i'}$ can be matched with: $U_{i'}$ nodes if there are that many in P or all nodes in P if there are less. When the algorithm comes to handle y_i, then, $y_{i+1} \dots y_{n'}$ are matched with as many nodes as possible. The algorithm extracts L_i nodes from P, because this is the minimum number of nodes that must be matched with y_i. Then it extracts from P all nodes x_κ with $m_1[\kappa] = i$, because these nodes cannot be matched in a Lemma 13-type matching M with y nodes to the left of y_i. The total number of nodes extracted so far is the minimum number of nodes that must be matched with y_i so $MinL_i$ is set to it. The algorithm then proceeds to extract additional nodes from P, up to a total of U_i as described above, so that y_i is also matched with as many nodes as possible before the algorithm proceeds to handle y_{i-1}. In the following, *iteration* i will not refer to the ith iteration but rather to the iteration that handles y_i.

Lemma 14 *The algorithm in Figure 10 achieves bound consistency for the lower capacity bounds of the count variables.*

Proof Let M' be the matching generated by the algorithm in Figure 10. That is, $M' = \{(x_j, y_i) \mid x_j \text{ was extracted in iteration } i\}$. Assume that the claim does not hold and let i be maximal such that $MinL_i > L_i$ and there is a generalized matching M with $L_i \leq |M(y_i)| < MinL_i$. By Lemma 13 we can assume that for every edge $(x_j, y_i) \in M$, $m_1[j] \leq i \leq m_2[j]$.

Let N_i be the set of nodes that the algorithm extracted from P in the first two loops of iteration i. Since $MinL_i > L_i$, after the first L_i nodes were extracted, there was at least one node $x_\kappa \in P$ with $m_1[x_\kappa] = i$ and

```
P ← [] (* Priority queue sorted by m1[j] *)
for i = n' to 1 do
    (* Insert into P all mates of y_i in m2. *)
    forall x_h with m2[h] = i do P.Insert x_h
    ℓ ← 0
    for ℓ = 1 to L_i do
        P.ExtractMax
        ℓ ← ℓ + 1
    end for
    while P is not empty and P.MaxPriority = i do
        P.ExtractMax
        ℓ ← ℓ + 1
    end while
    MinL_i ← ℓ
    while P is not empty and ℓ < U_i do
        P.ExtractMax
        ℓ ← ℓ + 1
    end while
endfor
```

Figure 10: Narrowing algorithm for the lower bounds of the count variables.

this implies that all nodes that were extracted in iteration i have m_1-value i so they are not matched in M with any y node to the left of y_i. Let i' be the minimum index such that $i' > i$ and either P was empty at the end of iteration i' of the algorithm in Figure 10 or during iteration i' a node $x_{j'}$ with $m_1[j'] < i$ was extracted. If no such $i' \leq n'$ exists, let $i' = n' + 1$. Let x_κ be a node that was extracted after iteration i' and not later than iteration i. Then $m_1[\kappa] \geq i$ (if x_κ was not extracted in iteration i, this follows from the choice of i'). In addition, $m_2[\kappa] < i'$ because if we assume otherwise then x_κ was in P in iteration i', but was not extracted then. So P was not empty at the end of iteration i', which by the choice of i'' implies that some $x_{\kappa'}$ with $m_1[\kappa'] < i$ was extracted. So all nodes that remained in P after iteration i' (in particular x_κ) have m_1-value smaller than i. We get that x_κ can be matched in M only with one of $y_i \ldots y_{i'-1}$. In addition, since P was not empty after any of iterations $i+1 \ldots i'-1$, we get that for each $i < \ell < i'$, $|M'(y_\ell)| = U_\ell$.

In total, we get that $y_i \ldots y_{i'-1}$ must be matched in M with all nodes that they are matched with in M', whose number is at least $MinL_i + \sum_{\ell=i+1}^{i'-1} U_\ell$.

This implies that $|M(y_i)| \geq MinL_i$.

Finally, we need to show that for all i, there is a generalized matching M_i with $|M_i(y_i)| = MinL_i$. We construct M_i as follows. First put in M_i all the edges representing the matchings of iterations n' to $i+1$. For each such edge $(x_{j'}, y_{i'})$, we have by construction that $m_1[j'] \leq i' \leq m_2[j']$. In addition, the number of nodes matched with $y_{i'}$ is between $L_{i'}$ and $U_{i'}$: At least $L_{i'}$ because there are at least $L_{i'}$ nodes x_κ with $m_2[\kappa] = i'$ which were inserted into P in iteration i'. At most $U_{i'}$, because if any node was extracted in the second loop because its m_1 value is i', then the same is true for all of the nodes that were extracted in iteration i', and this, by construction, holds for at most $U_{i'}$ nodes.

y_i is matched in M_i with the $MinL_i$ nodes that were extracted in the first two loops of the algorithm. The rest of the nodes, $y_1 \dots y_{i-1}$, are matched with the remaining x nodes; those that remain in the queue after the $MinL_i$ extractions of iteration i and those that were not inserted to the queue up to iteration i. To match these nodes with $y_1 \dots y_{i-1}$, we first match the nodes in P with y_{i-1} and each node $x_{j'}$ that was not yet inserted into P is matched with $y_{m_2[j']}$. The nodes that were not inserted yet to P are exactly all the nodes with $m_2[j'] < i$, so they fulfil the $L_{i'}$ values of $y_1, \dots, y_{i-1}$. For a node $x_{j''}$ in P, $m_1[j''] \leq i-1 < m_2[j'']$. We can then apply the third pass of the generalized matching algorithm to shift nodes to the left such that the $U_{i'}$ bounds are fulfilled for $y_1, \dots, y_{i-1}$. ∎

Figure 11 shows an algorithm that achieves bound consistency for the upper bounds of the count variables. Again, it uses a priority queue P for the x nodes, sorted by m_1 values. In each iteration it matches the respective y-node with the minimum required number of x nodes. At the beginning of iteration i it inserts into P all nodes which were matched with y_i in the second pass of the generalized matching algorithm. P contains at this point all candidates for matching with y_i. Let $|P|$ be the number of nodes in P at this point in time. The algorithm compares $|P|$ and U_i and sets $MaxU_i$ to the smaller of them. We will show that since the nodes that were extracted before iteration i are the minimum number of nodes that must be matched with y nodes to the right of y_i and the nodes that were not inserted into P before iteration $i-1$ cannot be matched with it, the number of nodes in P is an upper bound on the number of nodes that y_i can be matched with. The algorithm then proceeds to extract from P the minimum number of nodes that must be matched with y_i in a Lemma 13 type matching; first L_i nodes and then all nodes x_j with $m_1[j] = i$. It can then advance to iteration $i-1$.

```
P ← [] (* Priority queue sorted by m_1[j] *)
for i = n' to 1 do
    (* Insert into P all mates of y_i in m_2. *)
    forall x_h with m_2[h] = i do P.Insert x_h
    MaxU_i ← min{U_i, |P|}
    for ℓ = 1 to L_i do
        P.ExtractMax
    end for
    while P is not empty and P.MaxPriority = i do
        P.ExtractMax
    end while
endfor
```

Figure 11: Narrowing algorithm for the upper bounds of the count variables.

Lemma 15 *The algorithm in Figure 11 achieves bound consistency for the upper capacity bounds of the count variables.*

Proof Let M be a generalized matching, by Lemma 13 we can assume that for every edge $(x_\kappa, y_\ell) \in M$, $m_1[\kappa] \leq \ell \leq m_2[\kappa]$. We fix some i with $1 \leq i \leq n'$ and we will prove $|M(y_i)| \leq MaxU_i$. We may assume that $MaxU_i < U_i$, otherwise there is nothing to show. Let M' be the partial matching generated by the algorithm in Figure 11 up to iteration i. That is, for $\ell = i+1, \ldots, n'$, $M'(y_\ell)$ is the set of all nodes extracted in iteration ℓ, and $M'(y_i)$ consists of all nodes that are in P when $MaxU_i$ is determined. Observe that any x node that has not been inserted into P up to iteration i cannot be matched in M with $y_i, \ldots, y_{n'}$ because of its m_2-value. Thus $M(y_i, \ldots, y_{n'}) \subseteq M'(y_i, \ldots, y_{n'})$.

Below we identify subsets of $y_{i+1} \ldots y_{n'}$ such that for any subset S the x nodes in $M'(S)$ must be matched with the y nodes in S in any Lemma 13-type matching. We call such a subset *a block* and define it to be a consecutive set of y nodes $y_{i_\ell} \ldots y_{i_r}$ such that (1) for every matching mate x_ℓ of y_{i_ℓ}, $m_1[\ell] = i_\ell$ and (2) i_r is minimal such that $i_r \geq i_\ell$ and y_{i_r+1} is matched with some node x_r with $m_1[r] < i_\ell$ (see Figure 12).

Let $B = \{y_{i_\ell} \ldots y_{i_r}\}$ be a block. By construction, for any x_j which is matched with $y_\kappa \in B$, $m_1[j] \geq i_\ell$. In addition, for every such x_j, $m_2[j] \leq i_r$; otherwise x_j was in P in iteration i_{r+1} and since x_r was extracted first, $m_1[j] \leq m_1[r] < i_\ell$, contradicting condition (1) in the definition of a block above. We get that $M'(B) \subseteq M(B)$.

We partition $y_{i+1} \ldots y_{n'}$ into two sets B_{in} and B_{out} where B_{in} contains the nodes that belong to some block and B_{out} contains the nodes that do not belong to any block. If a node y_κ for some $\kappa > i$ is matched in M' with more than L_κ nodes then it is in B_{in} because for any of its matching mates x_j, $m_1[j] = \kappa$. This implies that any $y_\kappa \in B_{\text{out}}$ is matched with exactly L_κ x nodes in M'.

From the arguments above we conclude that $M(y_i \cup B_{\text{out}} \cup B_{\text{in}}) \subseteq M'(y_i \cup B_{\text{out}} \cup B_{\text{in}})$ and $M'(B_{\text{in}}) \subseteq M(B_{\text{in}})$, which implies $M(y_i \cup B_{\text{out}}) \subseteq M'(y_i \cup B_{\text{out}})$. So we get that $|M(y_i \cup B_{\text{out}})| \leq MaxU_i + \sum_{y_\kappa \in B_{\text{out}}} L_\kappa$. Thus $|M(y_i)| \leq MaxU_i$.

It remains to show that for every i there is a generalized matching M_i with $|M_i(y_i)| = MaxU_i$. For a node $y_{i'}$ with $i' > i$, let $M_i(y_{i'})$ be the nodes that were extracted from P in iteration i'. If $|M_i(y_{i'})| > L_{i'}$ then for every node $x_{j'}$ that was extracted in iteration i', $m_1[j'] = i'$ so $|M_i(y_{i'})| \leq U_{i'}$. Otherwise, $|M_i(y_{i'})| = L_{i'}$. In either case the capacities are respected. $M_i(y_i)$ is set to the topmost $MaxU_i$ nodes in the queue at the time when $MaxU_i$ is computed. By construction, for each such node x_j, $m_1[j] \leq i \leq m_2[j]$. Finally, $y_1, \ldots y_{i-1}$ are matched with the remaining nodes in P and the x nodes that were not yet inserted into P as described in the proof of Lemma 14. ∎

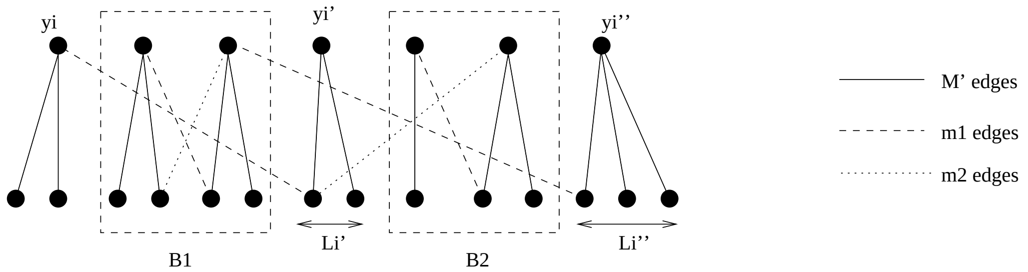

Figure 12: Example of the blocks in the proof of Lemma 15

Implementing the algorithm in linear time

As with the generalized matching algorithm, the algorithms in Figures 10 and 11 can be implemented in time $O(n' + n \log n)$ using a binary heap for the priority queue P. To shave off the logarithmic factor, we once again simulate the queue with offline-min type algorithms. In the lower bounds algorithm (Figure 10), the insertions and extractions of each iteration do not depend on the results of previous extractions from P; in iteration i the algorithm extracts $\max\{U_i, |P|\}$ nodes. This means that we can use the usual offline-min algorithm to determine the sequence of insertions and extractions and then use this sequence to find the $MinL_i$ values. For the upper bounds algorithm (Figure 11) we need to know the results of the extractions in the while loop in order to know how many nodes were extracted in each iteration. Note that this is similar to the situation of the second pass of the generalized matching algorithm (cf. Figure 2), and can be handled by the algorithm in Figure 13 which is analogous to the one in Figure 5 on Page 12.

```
k ← 1(* E_k will always be the next mandatory extraction *)
for i = n' to 1 do
    forall x_j with m_2[j] = i do
        (* x's inserted in iteration i (before E_k) *)
        s[j] ← k
    endfor
    for ℓ = 1 to L_i do
        create set {E_k} labelled 𝓛_k^(i,ℓ) (* for mandatory extraction ℓ in iter. i *)
        k ← k + 1
    endfor
endfor
create set {E_k} labelled with 𝓛_k^(n'+1,0) (* dummy extraction *)
forall x_j sorted in descending order according to m_1[j] do
    S ← find(E_s[j]); let 𝓛_k^(i,ℓ) be its label
    if i ≥ m_1[j] then
        x_j is removed by extraction ℓ in the for-loop of iteration i
        unite S with the set S' containing E_(k+1) and
            label the union with the former label of S'
    else
        x_j is extracted by the while-loop in iteration m_1[j]
    endif
endfor
```

Figure 13: Enhanced offline-min procedure for the algorithm in Figure 11

8 Conclusion

We have designed a propagation algorithm for the Global Cardinality Constraint that achieves bound consistency for the assignment variables. We wish to point out that there are two possible implementations for the algorithm. One runs in time $O((n'+n)\log n)$, uses very simple data structures and performs well in practice. The other one requires more elaborate data structures (for the offline-min computation) and achieves a running time of

$O(n + n')$ plus the time required to sort the assignment variables by the endpoints of their ranges. In some cases the latter is asymptotically better.

In addition, we present two linear-time algorithms that narrow the bounds of the count variables. The first is very simple but does not always achieve bound consistency. The second does achieve bound consistency but uses the more elaborate data structures that are required for the offline-min computations.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[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] 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.

[5] 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.

[6] J.-C. Régin. Arc-Consistency for Global Cardinality Constraints with Costs. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint P rogramming (CP 1999)*, volume 1713 of *LNCS*, pages 390–404, 1999.

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-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-2003-2-003	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-2003-1-015	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	D. Fotakis, R. Pagh, P. Sanders, P. Spirakis	Space Efficient Hash Tables with Worst Case Constant Access Time
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	Soon to be published
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
MPI-I-2001-2-003	S. Vorobyov	Experiments with Iterative Improvement Algorithms on Completely Unimodel Hypercubes
MPI-I-2001-2-002	P. Maier	A Set-Theoretic Framework for Assume-Guarantee Reasoning
MPI-I-2001-2-001	U. Waldmann	Superposition and Chaining for Totally Ordered Divisible Abelian Groups
MPI-I-2001-1-007	T. Polzin, S. Vahdati	Extending Reduction Techniques for the Steiner Tree Problem: A Combination of Alternative-and Bound-Based Approaches

MPI-I-2001-1-006	T. Polzin, S. Vahdati	Partitioning Techniques for the Steiner Problem
MPI-I-2001-1-005	T. Polzin, S. Vahdati	On Steiner Trees and Minimum Spanning Trees in Hypergraphs
MPI-I-2001-1-004	S. Hert, M. Hoffmann, L. Kettner, S. Pion, M. Seel	An Adaptable and Extensible Geometry Kernel
MPI-I-2001-1-003	M. Seel	Implementation of Planar Nef Polyhedra