Faster Algorithms for Bound-Consistency of the Sortedness and the Alldifferent Constraint

Kurt Mehlhorn and Sven Thiel

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

Abstract. We present narrowing algorithms for the sortedness and the alldifferent constraint which achieve bound-consistency. The algorithm for the sortedness constraint takes as input $2n$ intervals $X_1, \ldots, X_n$, $Y_1, \ldots, Y_n$ from a linearly ordered set D. Let $\mathcal{S}$ denote the set of all tuples $t \in X_1 \times \cdots \times X_n \times Y_1 \times \cdots \times Y_n$ such that the last n components of t are obtained by sorting the first n components. Our algorithm determines whether $\mathcal{S}$ is non-empty and if so reduces the intervals to bound-consistency. The running time of the algorithm is asymptotically the same as for sorting the interval endpoints. In problems where this is faster than $O(n \log n)$, this improves upon previous results.
The algorithm for the alldifferent constraint takes as input n integer intervals $Z_1, \ldots, Z_n$. Let $\mathcal{T}$ denote all tuples $t \in Z_1 \times \cdots \times Z_n$ where all components are pairwise different. The algorithm checks whether $\mathcal{T}$ is non-empty and if so reduces the ranges to bound-consistency. The running time is also asymptotically the same as for sorting the interval endpoints. When the constraint is for example a permutation constraint, i.e. $Z_i \subseteq [1; n]$ for all i, the running time is linear. This also improves upon previous results.

1 The Sortedness Constraint

1.1 Introduction

Let D be a non-empty linearly ordered set. An interval X in D consists of all elements of D which lie between two given elements a and b of D, i.e., $X = \{d \in D \; ; \; a \le d \le b\}$. For a non-empty interval X we use $\underline{X}$ and $\overline{X}$ to denote the smallest and largest element in X, respectively. The function *sort* maps any n-tuple over D to its sorted version, i.e., if $(d_1, \ldots, d_n) \in D^n$ then $sort(d_1, \ldots, d_n) = (e_1, \ldots, e_n)$ with $e_1 \le e_2 \le \ldots \le e_n$ and $e_i = d_{f(i)}$ for all i, $1 \le i \le n$, for some permutation f of $[1; n]$.
Let $X_1, \ldots, X_n, Y_1, \ldots, Y_n$ be $2n$ non-empty intervals in D which we consider fixed in this section. We use $\mathcal{S}$ to denote all $2n$-tuples $(d_1, \ldots, d_n, e_1, \ldots, e_n)$ with $d_i \in X_i$ and $e_i \in Y_i$ for all i and $(e_1, \ldots, e_n) = sort(d_1, \ldots, d_n)$. The *task of narrowing the sortedness constraint* is to decide whether $\mathcal{S}$ is non-empty and, if so, to compute the minimal and maximal element in each of its $2n$ components. Bleuzen-Guernalec and Colmerauer [BGC97,BGC00] gave an $O(n \log n)$

algorithm for narrowing the sortedness constraint. We relate the problem to matching theory and derive an alternative algorithm. The new algorithm has two advantages over the previous algorithms: it is simpler and its running time is $O(n)$ plus the time required to sort the interval endpoints of the x-ranges. In particular, if the interval endpoints are from a integer range of size $O(n^k)$ for some constant k the algorithm runs in linear time.

The last n components of any $2n$-tuple in $\mathcal{S}$ are sorted in increasing order and hence we may restrict our attention to *normalized* $2n$-tuples $(X_1, \ldots, X_n, Y_1, \ldots, Y_n)$ with $\underline{Y}_i \leq \underline{Y}_{i+1}$ and $\overline{Y}_i \leq \overline{Y}_{i+1}$ for all i with $1 \leq i < n$. Normalization can be achieved algorithmically by setting $\underline{Y}_{i+1}$ to $\max(\underline{Y}_i, \underline{Y}_{i+1})$ for i from 1 to $n-1$ and $\overline{Y}_{i-1}$ to $\min(\overline{Y}_i, \overline{Y}_{i-1})$ for i from n to 2. We assume from now on (assumption of normality) that our $2n$-tuple $(X_1, \ldots, X_n, Y_1, \ldots, Y_n)$ is normalized.

Example. We use the following running example in this section:

$$\begin{array}{lllll} X_1 = [1;16] & X_2 = [5;10] & X_3 = [7;9] & X_4 = [12;15] & X_5 = [1;13] \\ Y_1 = [2;3] & Y_2 = [6;7] & Y_3 = [8;11] & Y_4 = [13;16] & Y_5 = [14;18] \end{array}$$

1.2 A Connection to Matchings

We define a bipartite graph G which we call the *intersection graph.* The nodes of G are $\{x_i \ ; 1 \leq i \leq n\}$ and $\{y_j \ ; 1 \leq j \leq n\}$ and there is an edge $\{x_i, y_j\}$ if $X_i \cap Y_j \neq \emptyset$. Clearly, if $(d_1, \ldots, d_n, e_1, \ldots, e_n) \in \mathcal{S}$ and f is a permutation such that $d_{f(j)} = e_j$ for all j then the set $\big\{\{x_{f(j)}, y_j\} \ ; 1 \leq j \leq n\big\}$ is a perfect matching in G. The following lemma provides a partial converse and gives a first indication of the relevance of the intersection graph. The lemma was stated as property (4) in [BGC00].

Lemma 1 (Characterization of input ranges). *Fix a perfect matching $M = \big\{\{x_i, y_{g(i)}\} \ ; 1 \leq i \leq n\big\}$ in the intersection graph. For each i let d_i be an arbitrary element in $X_i \cap Y_{g(i)}$. Then there is a tuple in $\mathcal{S}$ whose i-th component is equal to d_i for all i with $1 \leq i \leq n$.*

Proof. Consider a tuple $t = (d_1, \ldots, d_n, e_1, \ldots, e_n)$ with $d_i \in X_i \cap Y_{g(i)}$ and $e_{g(i)} = d_i$ for all i. Thus $e_j \in Y_j$ for all j. If $e_1 \leq \ldots \leq e_n$ we have $t \in \mathcal{S}$ and we are done. So assume that there is a j with $e_j > e_{j+1}$. We know $e_j \in Y_j$ and $e_{j+1} \in Y_{j+1}$ and, by the assumption of normality, $\underline{Y}_j \leq \underline{Y}_{j+1}$ and $\overline{Y}_j \leq \overline{Y}_{j+1}$. Thus $e_j \in Y_{j+1}$ and $e_{j+1} \in Y_j$ and hence we may swap e_j and e_{j+1} to obtain the tuple $(d_1, \ldots, d_n, e'_1, \ldots, e'_n)$ with $e_i = e'_i$ for $i \notin \{j, j+1\}$, $e'_j = e_{j+1}$ and $e'_{j+1} = e_j$. We have again $e'_j \in Y_j$ for all j. Continuing in this way we can construct a tuple in $\mathcal{S}$. □

For a node v (either an x or a y) we use $N(v)$ to denote its set of neighbors in G and for a set V of nodes we use $N(V) = \cup_{v \in V} N(v)$ to denote the set of all neighbors of nodes in V. We use $I(v)$ to denote the set of indices of the nodes in $N(v)$. The set of neighbors $N(x_i)$ of each x_i forms an interval

$I(x_i) = [\min\{j\ ;\ X_i \cap Y_j \neq \emptyset\}\,;\max\{j\ ;\ X_i \cap Y_j \neq \emptyset\}]$ or simply I_i in the y's. Glover (see [Glo67] and [Law76, Section 6.6.6]) called graphs with this property convex and gave a simple matching algorithm for them: $f(1)$, $f(2)$, ..., $f(n)$ are defined in this order. Assume that $f(1)$, ..., $f(j-1)$ are already defined. Let $I = I(y_j) \setminus \{f(1), \ldots, f(j-1)\}$ and set $f(j)$ to i where $i \in I$ is such that $\overline{I}_i$ is minimal.

Example. In the following table we show how the algorithm would compute the function f for our example (in the second column the x's are sorted such that the interval endpoints of their ranges are increasing):

y_j	$N(y_j)$	$I(y_j) \setminus \{f(1), \ldots, f(j-1)\}$	$f(j)$
y_1	x_5, x_1	$5, 1$	5
y_2	x_3, x_2, x_5, x_1	$3, 2, 1$	3
y_3	x_3, x_2, x_5, x_1	$2, 1$	2
y_4	x_5, x_4, x_1	$4, 1$	4
y_5	x_4, x_1	1	1

Lemma 2 (Glover). *If the intersection graph has a perfect matching, the algorithm above constructs one.*

Proof. Assume that the intersection graph has a perfect matching π. We use induction on k to show that there is a perfect matching π_k which matches y_j with $x_{f(j)}$ for all $j \leq k$. The claim holds for $k = 0$ with $\pi_0 = \pi$. So assume $k > 0$. If π_{k-1} matches y_k with $x_{f(k)}$ we take $\pi_k = \pi_{k-1}$. Otherwise π_{k-1} matches y_k with x_i and $x_{f(k)}$ with y_l for some i and l. Then $l > k$ and by definition of $f(k)$ we have $\overline{X}_i \geq \overline{X}_{f(k)}$ and hence $y_l \in N(x_i)$. Thus we can switch the roles of $x_{f(k)}$ and x_i in π_{k-1} to obtain π_k, i.e. we match y_k with $x_{f(k)}$ and y_l with x_i. □

We are interested in the edges that belong to some perfect matching of the intersection graph. These edges form the *reduced* intersection graph. They are easy to characterize; this characterization was already used by Regin in his arc-consistency algorithm for the alldifferent constraint [Reg94].

Lemma 3. *Assume that M is a perfect matching in G. Let us construct the oriented intersection graph $\boldsymbol{G}$ by orienting all edges in G from their x-endpoint to their y-endpoint and adding the reverse edge for all edges in M. An edge (x_i, y_j) belongs to some perfect matching iff it belongs to a strongly connected component of $\boldsymbol{G}$.*

Proof. standard matching theory (see for example Section 7.6 in [MN99]) □

We next show how a perfect matching and the strongly connected components of the intersection graph can be computed. We start by sorting the x_i according to both their lower interval endpoints $\underline{X}_i$ and their upper interval endpoints $\overline{X}_i$, i.e. we compute the sorting permutations.

The probably most suggesting implementation of Glover's algorithm maintains a priority queue P. After iteration $j-1$, the queue P contains all

$i \in I(\{y_1, \dots, y_{j-1}\}) \setminus \{f(1), \dots, f(j-1)\}$ ordered according to their upper interval endpoint $\overline{X}_i$. In iteration j we first add all i to P with $\overline{Y}_{j-1} < \underline{X}_i \leq \overline{Y}_j$. We then select the i with smallest $\overline{X}_i$ from P. If P is empty or $\overline{X}_i < \underline{Y}_j$, we detect that there is no perfect matching. Otherwise we set $f(j) = i$. This implementation has complexity $O(n \log n)$.

But as we have the sorting of the x's according to both bounds of their ranges, we can implement the algorithm in linear time. We reduce the problem to an instance of the offline-min problem [AHU74, Chapter 4.8], which can be solved in linear time using the union-find data structure of Gabow and Tarjan [GT83]. We show that we can construct the sequence of *insert* and *extractmin* operations performed on the priority queue whithout knowing the results of the extract operations. The construction is iterative. We start with an empty sequence, in iteration j we append the following operations: *insert*(i) for all i with $\overline{Y}_{j-1} < \underline{X}_i \leq \overline{Y}_j$ and then *extractmin*$_j$. The offline-min data structure determines, if possible, for every insertion the corresponding extraction. If there is a matching in the intersection graph, the algorithm will run to completion and find the same matching as before. If there is no matching there are two ways to get stuck: The result of *extractmin*$_j$ is a value i with $\overline{X}_i < \underline{Y}_j$. Or the offline-min data structure detects an insert operation for which there is no corresponding extract operation. Since we have n insertions and n extractions in our sequence, this means that there is an extraction which is performed on an empty priority queue.

We next show how to compute the strongly connected components of the intersection graph. By construction, we have that for every j the edges $(x_{f(j)}, y_j)$ and the reverse edge $(y_j, x_{f(j)})$ belong to $\boldsymbol{G}$. Thus the two nodes always belong to the same strongly connected component and we can imagine that the nodes have been merged into a single node xy_j. The incoming edges are the edges of y_j and the outgoing edges are the edges of $x_{f(j)}$. It helps to visualize the graph with the nodes $xy_1, \dots, xy_n$ drawn from left to right in that order. There are many algorithms for computing strongly connected components, all based on depth-first search. We use the algorithm of [CM96] as a basis and adapt it to the special structure of our graph. We will show how to compute the components in time $O(n)$. This is not trivial since the intersection graph may have $\Omega(n^2)$ edges.

The algorithm is based on DFS. It maintains the strongly connected components of $\boldsymbol{G}_{\text{cur}}$, the graph consisting of all nodes and edges visited by DFS. A component is completed if the call to DFS is completed for all nodes in the component and uncompleted otherwise. The root of a component is the node in the component with the smallest DFS-number. We maintain two stacks. The stack S_1 contains the nodes in uncompleted components in order of increasing DFS-number. It was shown in [CM96] that each uncompleted component forms a contiguous segment in S_1 and that the roots of all uncompleted components lie on a single tree path. The second stack S_2 contains an item $\langle \mathit{root}, \mathit{rightmost}, \mathit{maxX} \rangle$ for every uncompleted component. The fields *root* and *rightmost* denote the indices of the root (also leftmost) node and the rightmost node of the component. The field *maxX* is the maximum upper interval endpoint of an x-node in the component, this determines the rightmost xy_j that can be reached from within the component. We still need to say how we scan the edges out of a node. We scan from left to right, i.e., the edge (xy_i, xy_j) is scanned before (xy_i, xy_k) iff $j < k$. We maintain the invariant that for a node xy in a component $\langle \mathit{root}, \mathit{rightmost}, \mathit{maxX}\rangle$ all edges (xy, xy_i) with $i \le \mathit{rightmost}$ have been scanned. Since the outgoing edges of a node xy form an interval and we scan them from left to right, we have that DFS visits the nodes in the order $xy_1, \ldots, xy_n$, provided that top-level calls to DFS are always performed on the unreached node xy with the smallest index. Thus tree edges are easy to recognize if we maintain the index *next* of the unreached node with minimal index.

The algorithm proceeds like this. If the stack S_2 of uncompleted components is empty, we start a new component by pushing a corresponding item on the stack. We also push xy_{next} on the stack S_1. This amounts to a top-level call of DFS. Otherwise we consider the topmost uncompleted component $\langle \mathit{root}, \mathit{rightmost}, \mathit{maxX}\rangle$ and determine whether it can reach xy_{next}. This is the case iff $\mathit{maxX} \ge \underline{Y}_{next}$. Assume first that it can reach xy_{next}. Then xy_{next} forms a new uncompleted component which extends the path of uncompleted components. In the algorithm of [CM96] we would push xy_{next} onto S_1 and the item $C = \langle xy_{next}, xy_{next}, \overline{X}_{f(next)}\rangle$ onto S_2. Then we would scan the outgoing edges from left to right. The leftward edges out of xy_{next} could cause merging of components: Consider the components $C_1, \ldots, C_k$ on S_2, where $C_k = C$ is the topmost component. Let j be minimal such that xy_{next} can reach a node in C_j, i.e. $\overline{Y}_{\mathit{rightmost}_j} \ge \underline{X}_{f(next)}$. Then the components $C_j, \ldots, C_k$ would be merged into a single component, which is represented by the item $C' = \langle \mathit{root}_j, \mathit{next}, \max(\mathit{maxX}_j, \ldots, \mathit{maxX}_k)\rangle$. In order to avoid unnecessary push and pop operations, our algorithm checks first whether components can be merged before pushing C onto S_2. If this is the case, it pops $C_{k-1}, \ldots, C_j$, computes C' and pushes it onto S_2. Otherwise it pushes C onto S_2. In either case we push xy_{next} onto S_1.

We come to the case where the topmost component cannot reach xy_{next}. Then we have explored all edges out of that component and hence can declare it completed. We pop all nodes between the root and the rightmost node from the stack S_1 of unfinished nodes and label them with their scc-number.

Example. We show a table of the states of our algorithm when it computes the strongly connected components for our example:

next	S_1	S_2	completed SCCs	action
1	1	$\langle 1,1,13\rangle$	-	start component
2	1,2	$\langle 1,1,13\rangle$, $\langle 2,2,9\rangle$	-	start component
3	1,2,3	$\langle 1,1,13\rangle$, $\langle 2,3,10\rangle$	-	merge
4	1	$\langle 1,1,13\rangle$	$\{xy_2, xy_3\}$	complete component
4	1,4	$\langle 1,1,13\rangle$, $\langle 4,4,15\rangle$	$\{xy_2, xy_3\}$	start component
5	1,4,5	$\langle 1,5,16\rangle$	$\{xy_2, xy_3\}$	merge
-	-	-	$\{xy_2, xy_3\}$, $\{xy_1, xy_4, xy_5\}$	complete component

Theorem 1. *The asymptotic time complexity of constructing a matching M and the strongly connected components of the intersection graph is $O(n)$ plus the time for sorting the endpoints of the x-ranges.*

Proof. by the discussion above. □

1.3 Output Ranges

For j, $1 \le j \le n$, we use T_j to denote the projection of $\mathcal{S}$ onto the y_j-coordinate. We show how to compute $\overline{T}_j$ for all j. Assume that the intersection graph has a perfect matching and that our algorithm computed the function f. For $j = 1, \ldots, n$ let $\tau_j = \max X_{f(j)} \cap Y_j$. We claim that $\overline{T}_j = \tau_j$ for all j. First we show that $\tau_1 \le \ldots \le \tau_n$ which implies $(\tau_{f^{-1}(1)}, \ldots, \tau_{f^{-1}(n)}, \tau_1, \ldots, \tau_n) \in \mathcal{S}$ and hence $\tau_j \le \overline{T}_j$. Assume $\tau_j < \tau_{j-1}$ for some j; then $\overline{X}_{f(j)} < \tau_{j-1} = \min(\overline{X}_{f(j-1)}, \overline{Y}_{j-1})$. Since $\overline{X}_{f(j)} < \overline{X}_{f(j-1)}$ the value $f(j)$ must have entered the priority queue in iteration j and not earlier. Thus $\overline{Y}_{j-1} < \underline{X}_{f(j)} \le \overline{X}_{f(j)} < \tau_{j-1} \le \overline{Y}_{j-1}$, a contradiction.

Now we prove by induction that $\overline{T}_j$ cannot be larger than τ_j. Assume we have already established $\tau_h = \overline{T}_h$ for all $h < j$. Imagine that we restrict Y_j to a single value $e \in [\tau_j; \overline{Y}_j]$ and set $\overline{Y}_h$ to $\min(e, \overline{Y}_h)$ for $h < j$ and $\underline{Y}_h$ to $\max(e, \underline{Y}_h)$ for $h > j$. Note that this change preserves normality and does not restrict $\overline{Y}_h$ below $\overline{T}_h$ for $h < j$. If we rerun our matching algorithm, it will construct the same matching as before, until it tries to match y_j. It will also extract $f(j)$ from the priority queue, but in case $e > \tau_j$ the algorithm will get stuck. Thus $\tau_j = \overline{T}_j$.

Example. In our example we get $(\overline{T}_1, \ldots, \overline{T}_5) = (3, 7, 10, 15, 16)$.

The symmetric procedure computes the lower bounds. We start with a function f' which is obtained in the following way: $f'(n), \ldots, f'(1)$ are defined in that order. When $f'(n), \ldots, f'(j+1)$ are already determined, we let $I = I(y_j) \setminus \{f'(n), \ldots, f'(j+1)\}$ and set $f'(j)$ to $i \in I$ such that $\underline{I}_i$ is maximal. Then $\underline{T}_j = \min X_{f'(j)} \cap Y_j$.

Note that we can compute the output ranges from any graph whose edge-set is a superset of the reduced intersection graph and a subset of the intersection graph. We can do it in linear time, provided that we have the sortings of the x's.

1.4 Input Ranges

Given Lemmas 1 and 3, the narrowing of the input intervals becomes easy. For i, $1 \le i \le n$, we use S_i to denote the projection of $\mathcal{S}$ onto the x_i-coordinate.

Lemma 4. *We have $S_i = X_i \cap \cup_{y_j \in N'(x_i)} Y_j$, where $N'(x_i)$ denote the set of neighbors of x_i in the reduced intersection graph. In particular, $\underline{S}_i = \min X_i \cap Y_{j_l}$ and $\overline{S}_i = \max X_i \cap Y_{j_h}$ where y_{j_l} and y_{j_h} are the minimal and maximal elelements in $N'(x_i)$, respectively.*

Proof. immediate. □

We now show how to compute the minimal neighbors of the x's, a symmetric procedure can find the maximal neighbors. Recall that each node xy in an scc C stands for a pair $\{x_{f(j)}, y_j\}$ of matched nodes. Assume that an scc C consists of the nodes $(x_{i_1}, \ldots, x_{i_k})$ and $(y_{j_1}, \ldots, y_{j_k})$ with $\underline{X}_{i_1}, \leq \ldots \leq \underline{X}_{i_k}$ and $j_1 < \ldots < j_k$. Because of the normalization of the y's we have $\overline{Y}_{j_1} \leq \ldots \leq \overline{Y}_{j_k}$. We can determine the minimal neighbor of every x in C by merging the sequences; observe that the minimal neighbor of x_{i_s} is y_{j_t} iff $\overline{Y}_{j_{t-1}} < \underline{X}_{i_s} \leq \overline{Y}_{j_t}$. The two sortings of the nodes can be computed as follows. The sorting of the y's is already determined by the scc-algorithm. The sorting of the x's is obtained by going through the sorted list of all x's and partitioning it according to scc-number. This bucket-sort step requires linear time and gives us the sorting of the x-nodes for all components. This proves that we can narrow the input ranges in time $O(n)$.

Example. If we split up the xy-nodes in our example, we get the two components $C_1 = \{y_2, x_3, y_3, x_2\}$ and $C_2 = \{y_1, x_5, y_4, x_4, y_5, x_5\}$. By merging the sequences $(\underline{X}_2, \underline{X}_3) = (5, 7)$ and $(\overline{Y}_2, \overline{Y}_3) = (7, 11)$, the algorithm discovers that the minimal neighbor of x_2 and x_3 in the reduced intersection graph is y_2 and hence $(\underline{S}_2, \underline{S}_3) = (6, 7)$. When the algorithm processes C_2, it finds out that y_1 is the minimal neighbor of x_5 and x_1 and that y_4 is the minimal neighbor of x_4. Thus it can compute $(\underline{S}_5, \underline{S}_1, \underline{S}_4) = (2, 2, 13)$.

1.5 Summary of the Full Algorithm

The full algorithm is as follows:

1. Sort the x-ranges according to their lower and their upper endpoints.
2. Normalize the y-ranges.
3. Perform a down sweep to compute the matching M_0 and the upper bounds of the y-ranges and an up sweep to compute a matching M_1 and the lower bounds of the y-ranges.
4. Compute the strongly connected components.
5. Reduce the x-ranges.

Except for the first step, all steps take linear time. Thus the complexity of the whole algorithm is asymptotically the same as for sorting the lower and upper endpoints of the x-ranges. This is $O(n \log n)$ in general, but is $O(n)$ if the interval endpoints are drawn from a range of size $O(n^k)$ for some constant k. As Bleuzen-Guernalec and Colmerauer have stated in [BGC00], every narrowing algorithm for the sortedness constraint which achieves bound-consistency can be used for sorting n elements of the set D in time $O(n)$ plus the running time of the algorithm. Thus the complexity of our algorithm is asymptotically optimal in all models of sorting.

1.6 Implementation

We have a stand-alone implementation of the algorithm and we have also incorporated it in the constraint programming system MOZ [Moz]. The implementation can be obtained from the second author.

2 Alldifferent Constraint

2.1 Introduction

Let $X_1, \ldots, X_n$ be n non-empty intervals in the integers. We use $\mathcal{S}$ to denote the set of all n-tuples $(d_1, \ldots, d_n)$ in $X_1 \times \cdots \times X_n$ such that $d_i \neq d_j$ for all $i < j$. The *task of narrowing the alldifferent constraint* is to decide whether $\mathcal{S}$ is non-empty and, if so, to compute the minimal and maximal element in each of its n components. Puget [Pug98] gave an $O(n \log n)$ algorithm for this task. The running time of our algorithm is $O(n)$ plus the time required for sorting the interval endpoints. In particular, if the endpoints are from a range of size $O(n^k)$ for some constant k, the algorithm runs in linear time. This is for example the case when $X_1, \ldots, X_n$ encode a permutation, i.e. $X_i \subseteq [1; n]$ for all i, or in the alldifferent constraints used in [Pug98] to model the n-queens problem.

Example. In this section we use the following running example:

$$X_1 = [3; 4], X_2 = [7; 7], X_3 = [2; 5], X_4 = [2; 7], X_5 = [1; 3], X_6 = [3; 4]$$

2.2 A Connection to Matchings

As before we reduce the problem to determining the matchings in a bipartite graph G. Let $l = \min_{1 \leq i \leq n} \underline{X}_i$ and $h = \max_{1 \leq i \leq n} \overline{X}_i$. We assume that $m = h - l + 1 \geq n$, otherwise we know that $\mathcal{S}$ is empty. The nodes of G are $\{x_i \; ; \; 1 \leq i \leq n\}$ and $\{y_j \; ; \; l \leq j \leq h\}$ and there is an edge $\{x_i, y_j\}$ if $j \in X_i$. We have the following one-to-one correspondence:

Lemma 5. *Every Matching* $M = \big\{\{x_i, y_{g(i)}\} \; ; \; 1 \leq i \leq n\big\}$ *in* G *corresponds to the tuple* $(g(1), \ldots, g(n))$ *in* $\mathcal{S}$ *and vice versa.*

Proof. by the definition of a matching. □

It is clear that G is convex. We use a slightly modified version of the algorithm in section 1.2 to compute a matching M in G. We encode the matching by a function $f : [l; h] \to [1; n] \cup \{\mathit{free}\}$ which maps every y-node to its mate on the x-side or indicates that that the node has no mate. We compute $f(l), f(l+1), \ldots, f(h)$ in that order. Assume that $f(l), \ldots, f(j-1)$ are already defined. Let $I = I(y_j) \setminus \{f(l), \ldots, f(j-1)\}$ denote the set of all unmatched neighbors of y_j. If I is empty we set $f(j)$ to *free*, otherwise we set $f(j)$ to i such that $\overline{X}_i$ is minimal.

Example. In our example we get the following function f:

j	1	2	3	4	5	6	7
$f(j)$	5	3	1	6	4	*free*	2

Lemma 6. *If there is a matching of cardinality* n *in* G*, the algorithm above constructs one.*

Proof. Consider the graph G' which is obtained from G by adding the nodes $x_{n+1}, \ldots, x_m$ to the x-side and connecting them to all nodes on the y-side. Clearly G' has a perfect matching. Thus the algorithm in section 1.2 will construct a mapping f' encoding it. Assume that we have modified both algorithms such that in case of multiple choices for $f(j)$ or $f'(j)$ the smallest index i (with maximal $\overline{X}_i$) is chosen. Then $f(j) = f'(j)$ if $f'(j) \leq n$, and $f(j) = \mathit{free}$ otherwise. □

The matching constructed by this algorithm has an interesting property which we will use later:

Lemma 7. *Let $\{x_i, y_j\}$ be an edge in the matching M constructed by the algorithm above. Then any $y_{j'}$ with $j' < j$ and $j' \in X_i$ is matched in M.*

Proof. Assume $y_{j'}$ is free. At the time when the algorithm determined $f(j')$, the node x_i was not matched since y_j had not been processed. This contradicts $f(j') = \mathit{free}$. □

How do we implement the algorithm? We have to take care of the fact that we do not have a bound on m, i.e. the number of nodes on the y-side. We want an algorithm whose time complexity does not depend on m. Let us first look at a priority queue implementation: We sort the x-ranges according to their lower interval endpoints. After iteration $j-1$, our priority queue P contains all $i \in I(\{y_1, \ldots, y_{j-1}\} \setminus \{f(1), \ldots, f(j-1)\}$ ordered according to their upper interval endpoint $\overline{X}_i$. In iteration j we first add all i to P with $\overline{Y}_{j-1} < \underline{X}_i \leq \overline{Y}_j$. If P is empty, the node y_j becomes a free node. And so will all his successors until the next insertion into P. Since we know the index i_0 of the unmatched x-node with the smallest interval endpoint, we can advance directly to iteration $j' = \underline{X}_{i_0}$. If P is non-empty, we select the i with the smallest $\overline{X}_i$ from P and check whether $\overline{X}_i < \underline{Y}_j$. If so, we detect that there is no matching of cardinality n, otherwise we set $f(j)$ to i.
The sequence of *insert* and *extractmin* operations can be computed in advance. When we construct that sequence we can also determine the free nodes. By counting the number of insertions and extractions, we know when the priority queue would become empty. This means that if we know the sorting of the x-ranges according to lower and upper endpoint, we can compute the matching M and the intervals of free nodes in time $O(n)$ using an offline-min data structure.

We are interested in the edges that belong to some matching in G where all x-nodes are matched. Therefore we construct the oriented graph $\boldsymbol{G}$. We orient all edges in G from their x-endpoint to their y-endpoint and add the reverse edge for all edges in M. The following lemma was already used by Regin in [Reg94], it characterizes the edges we are looking for:

Lemma 8. *An edge (x_i, y_j) belongs to some matching of cardinality n in G iff it belongs to a strongly connected component of $\boldsymbol{G}$ or lies on a path to a free node.*

Proof. standard matching theory (see for example Section 7.6 in [MN99]) □

The computation of the strongly connected components can be carried out as earlier because every single free node forms a component of its own. Thus we only have to consider the n matched nodes on the y-side.

Now we show how to mark all matched y-nodes that can reach a free node in time $O(n)$. First we want to put down a few facts about the strongly connected components of the graph $\boldsymbol{G}$. Let C be a component of $\boldsymbol{G}$ and let *root* and *rightmost* denote the minimal and maximal index of a y-node in C. We define $I(C)$ to be the interval $[root; rightmost]$.

1. Let $j \in I(C)$. Then there is a path from y_{root} to y_j. If $y_j \in C$ there is nothing to show. Otherwise consider a path $y_{root} = y_{j_1}, x_{i_1}, y_{j_2}, x_{i_2}, \dots, y_{j_k} = y_{rightmost}$ in C from the root to the rightmost node. Since $j_1 < j < j_k$ there must be a κ with $1 \le \kappa < k$ such that $j_\kappa < j < j_{\kappa+1}$. We have $X_{i_\kappa} \supseteq [j_\kappa; j_{\kappa+1}] \ni j$.
2. Let C be a non-trivial component, i.e. C does not consist of a single free node. Then any y_j with $j \in I(C)$ is a matched node. Assume that y_j is free. Considering a path from $y_{rightmost}$ to y_{root}, one can show similarly as in fact 1) that there is a node $y_{j'} \in C$ matched to some node $x_{i'}$ with $j' > j$ and $j \in X_{i'}$. This is a contradiction to Lemma 7.
3. For two different strongly connected components C and C' exactly one of the following 3 statements holds:

$$\text{i)}\quad I(C) \cap I(C') = \emptyset \qquad \text{ii)}\quad I(C) \subset I(C') \qquad \text{iii)}\quad I(C') \subset I(C)$$

   This follows directly from the fact 1). If statement iii) holds, we say that C' is *nested* in C. And we say that C' is *directly nested* in C, if there is no strongly connected component C'' different from C and C' such that $I(C') \subset I(C'') \subset I(C)$.
4. Let C and C' be two components such that C' is nested in C. Let x_i be a node in C' and let (x_i, y_j) be an edge in $\boldsymbol{G}$. We claim that y_j lies in a component nested in C. Assume otherwise. By fact 1) there is a path from the root of C to x_i. Thus if $y_j \in C$ then $C = C'$, a contradiction. If $j \notin I(C)$, i.e. y_j lies to the left or to the right of C, then there is also an edge from x_i to the root or to the rightmost node of C since the set of neighbors of x_i forms an interval. Again we can conclude $C = C'$ and derive a contradiction. We say that the edges of C' cannot escape from C. They can only lie within C' or between C' and an other component C'' nested in C.

Consider the top-level components $C_1, \dots, C_k$ of $\boldsymbol{G}$, i.e. all components which are not nested in an other component. This also includes free nodes, which form top-level components of their own by fact 2). We know that the intervals $I(C_1), \dots I(C_k)$ are a partition of the set of indices of the y-nodes. Thus we can assume that the components are numbered such that $\overline{I}(C_i) < \underline{I}(C_j)$ for all $i < j$.

Imagine that we shrink each top-level component and its nested components to a single node keeping only the edges between different top-level components.

Then we get an acyclic graph $\boldsymbol{G}_\mathrm{s}$. To be precise, the nodes of $\boldsymbol{G}_\mathrm{s}$ are $C_1, \ldots, C_k$ and there is an edge (C_i, C_j) iff there are $u \in C_i$ and $v \in C_j$ such that the edge (u, v) is in $\boldsymbol{G}$ and $i \neq j$. We call a node F of $\boldsymbol{G}_\mathrm{s}$ *free* iff F consists of a single free node of $\boldsymbol{G}$. Let us consider a path in $\boldsymbol{G}$ from a node y_j in a top-level component C to a free node y_f. This path cannot visit a node in a nested component, because edges from nested components cannot escape from their enclosing top-level component by fact 4). Thus the path corresponds to a path in $\boldsymbol{G}_\mathrm{s}$ from C to the free node $F = \{y_f\}$.
Clearly, the converse is also true. If we have a path in $\boldsymbol{G}_\mathrm{s}$ from a node C to a free node $F = \{y_f\}$, we can find a path in $\boldsymbol{G}$ from any node $u \in C$ to y_f. In order to find all nodes of $\boldsymbol{G}_\mathrm{s}$ that can reach a free node we exploit the following property of $\boldsymbol{G}_\mathrm{s}$:

Lemma 9. *Assume that the nodes of $\boldsymbol{G}_\mathrm{s}$ are numbered as described above and that there is a path from C to C' in $\boldsymbol{G}_\mathrm{s}$. Then there is also a monotone path from C to C', which means a path $C = C_{i_1}, C_{i_2}, \ldots, C_{i_k} = C'$ with $i_1 < \ldots < i_k$ or $i_1 > \ldots > i_k$.*

Proof. Consider any path $C = C_{l_1}, C_{l_2}, \ldots, C_{l_k} = C'$ from C to C' in $\boldsymbol{G}_\mathrm{s}$ and assume w.l.o.g. that $l_1 < l_2$. If the path is not monotone then there is a κ with $1 < \kappa < k$ such that $l_{\kappa-1} < l_\kappa > l_{\kappa+1}$. We distinguish two cases:

- $l_{\kappa-1} < l_{\kappa+1} < l_\kappa$:
  Since $\boldsymbol{G}_\mathrm{s}$ contains the edge $(C_{l_{\kappa-1}}, C_{l_\kappa})$ there must be an edge $(x_{i_{\kappa-1}}, y_{j_\kappa})$ in $\boldsymbol{G}$ with $x_{i_{\kappa-1}} \in C_{l_{\kappa-1}}$ and $y_{j_\kappa} \in C_{l_\kappa}$. As $x_{i_{\kappa-1}}$ can reach a node in $C_{l_{\kappa-1}}$ (its mate) and a node in C_{l_κ}, we have that $I(x_{i_{\kappa-1}}) \supset I(C_{l_{\kappa+1}})$. Thus the edge $(C_{l_{\kappa-1}}, C_{l_{\kappa+1}})$ is in $\boldsymbol{G}_\mathrm{s}$ and we can shorten the path.
- $l_{\kappa+1} \leq l_{\kappa-1} < l_\kappa$:
  Since $\boldsymbol{G}_\mathrm{s}$ contains the edge $(C_{l_\kappa}, C_{l_{\kappa+1}})$ there must be an edge $(x_{i_\kappa}, y_{j_{\kappa+1}})$ in $\boldsymbol{G}$ with $x_{i_\kappa} \in C_{l_\kappa}$ and $y_{j_{\kappa+1}} \in C_{l_{\kappa+1}}$. As the neighbors of x_{i_κ} form an interval in the y-nodes, we have that there is an edge from x_{i_κ} to the rightmost node of $C_{l_{\kappa-1}}$, and hence the edge $(C_{l_\kappa}, C_{l_{\kappa-1}})$ is in $\boldsymbol{G}_\mathrm{s}$. This contradicts the fact that this graph is acyclic.

This proves that we can shorten the path until it becomes monotone. □

The statement of Lemma 7 implies that no free node (either in $\boldsymbol{G}$ or $\boldsymbol{G}_\mathrm{s}$) has an incoming edge from the right. And hence, we only have to consider monotone paths from left to right in our search for nodes that can reach free nodes. Now it is easy to design an algorithm that marks all matched y-nodes of $\boldsymbol{G}$ which can reach a free y. We know by the facts 2) and 4) that these nodes can only reside in non-trivial top-level components. We can easily modify the algorithm which computes the strongly connected components such that it generates a list L of these components, for the stack of uncompleted components represents the nesting relation. Whenever a top-level component becomes completed, we append the corresponding item $\langle \mathit{root}, \mathit{rightmost}, \mathit{maxX}, \mathit{nodes} \rangle$ to L, where *nodes* is a list of the indices of all y-nodes in the component. The overhead of this is only a constant factor.

After we have finished the computation of the components we perform a sweep over the non-trivial top-level components from right to left. We maintain the index $j^\star$ of the leftmost node that we have seen so far and that is either free or marked. We know that all y-nodes outside the non-trivial components (and their nested components) are free. So when the sweep moves from a non-trivial component C' to its immediate non-trivial successor C to the left, we advance $j^\star$ to $\overline{I}(C)+1$ if there is a gap between the rightmost node of C and the root of C', i.e. $\overline{I}(C) < \underline{I}(C')-1$. With the aid of the value $maxX$, we can determine whether the node $y_{j^\star}$ can be reached from C. If so, we mark all y-nodes in the component and advance $j^\star$ to the index of the root of C. The complexity of the sweep is linear in the number of matched y-nodes in top-level components. Thus the marking can be done in $O(n)$ time.

Example. In our example, we have one nested component $\{y_3, x_1, y_4, x_6\}$ and three top-level components $\{y_1, x_5\}$, $\{y_2, x_3, y_5, x_4\}$ and $\{y_7, x_2\}$. The sweep from right to left will first mark y_2 and y_5 because the $maxX$-value for their component is 7 and hence it can reach the free node y_6. Then it will also mark y_1 since its component can reach y_2.

2.3 Narrowing of the Ranges

Let S_i denote the projection of $\mathcal{S}$ onto the x_i-coodinate for $i = 1, \ldots, n$. Because of Lemma 7, we do not have to consider free nodes when we determine the lower bounds $\underline{S}_1, \ldots, \underline{S}_n$, and hence we can do it in the same way as the input ranges for the sortedness-constraint (cf. section 1.4). Since no x-node in a nested component can reach a free node, the computation of the upper bounds does not change for these nodes either.

In order to compute $\overline{S}_i$ for a node x_i in a top-level component we must determine two things:

1. We must compute the maximal neighbor y_{s_i} of x_i which belongs to the same strongly connected component as x_i.
2. We have to find the maximal neighbor y_{t_i} of x_i that is either free or marked. (If x_i has no such neighbor, we make sure that $t_i \leq s_i$.)

We have $\overline{S}_i = \max(s_i, t_i)$ by the Lemmas 5 and 8. The neighbor y_{s_i} can be found as in section 1.4. The computation of y_{t_i} is similar. First we generate the sequence $(U_1, \ldots, U_k)$ where $U_1, \ldots, U_k$ are non-empty intervals that form a partition of the set of unmarked matched y-nodes with the property $\overline{U}_{j-1} < \underline{U}_j - 1$ for $1 < j \leq k$. Since every interval contains a matched node, we have $k < n$. The sequence is easily constructed. Assume that our matching algorithm has computed the sequence of matched nodes sorted by ascending index. We step through this sequence, sort out the marked nodes and generate the intervals. In linear time we can generate a sorting $(i_1, \ldots, i_l)$ of the ranges of the x-nodes in top-level components such that $\overline{X}_{i_1}, \leq \ldots \leq \overline{X}_{i_l}$.

We determine $t_{i_1}, \ldots, t_{i_l}$ by merging the sequence $(\overline{X}_{i_1}, \ldots \overline{X}_{i_l})$ and the sequence $(\underline{U}_1, \overline{U}_1, \ldots, \underline{U}_k, \overline{U}_k)$. When an $\overline{X}_i$ does not lie within a U-interval, we set $t_i = \overline{X}_i$, because then the maximal neighbor of x_i is free or marked. Otherwise $\overline{X}_i \in U_j$ for some j, and we set $t_i = \underline{U}_j - 1$. Note that y_{t_i} is either the maximal free or marked neighbor of x_i or we have $t_i < \underline{X}_i \leq s_i$.

Example. We want to look now at the x-nodes which belong to top-level components. The sorted sequence of the upper interval endpoints of their ranges is $(\overline{X}_5, \overline{X}_3, \overline{X}_4, \overline{X}_2) = (3, 5, 7, 7)$. In our example the unmarked matched y-nodes are partitioned in two intervals $(U_1, U_2) = ([3; 4], [7; 7])$. Thus the merging step produces $(t_5, t_3, t_4, t_2) = (2, 5, 6, 6)$. The indices of the maximal neighbors of the x-nodes in their component are $(s_5, s_3, s_4, s_2) = (1, 5, 5, 7)$. And hence the narrowed upper bounds are $(\overline{S}_5, \overline{S}_3, \overline{S}_4, \overline{S}_2) = (2, 5, 6, 7)$.

2.4 Summary of the Full Algorithm

The full algorithm looks like this:

1. Sort the ranges according to their upper and lower endpoints.
2. Perform a sweep to compute the initial matching M.
3. Compute the strongly connected components of $\boldsymbol{G}$.
4. Mark all matched y-nodes that can reach a free node.
5. Narrow the ranges

Except for the first step, all steps take linear time, and hence the complexity of the algorithm is asymptotically the same as for sorting endpoints of the ranges. If we have a permutation constraint, the narrowing can be done in linear time. In this case there are no free nodes and the forth step can be left out.

2.5 Implementation

We have not implemented the algorithm yet, but we expect that it will also show good performance in pratice.

3 Conclusion

We have presented narrowing algorithms for the alldifferent and the sortedness constraint which achieve bound-consistency. Our algorithms are competitive with the best previously known algorithms. Under some circumstances our algorithm have a better asymptotic running time. For example, we can narrow instances of the alldifferent constraint in linear time when the variables encode a permutation.

Bleuzen-Guernalec and Colmerauer [BGC97] have already noticed that a bound-consistency narrowing algorithm for the sortedness constraint can be used for narrowing permutation constraints. We feel that translating both constraints to matching problems in bipartite graphs has made the relationship more obvious, because the matching problems for both constraints are the same. The matching problems originating from general instances of the alldifferent constraint are more difficult, because one has to cope with free nodes. Thus we think that matching theory has not only provided some efficient algorithms but also some deeper insight into the structure of the sortedness and the alldifferent constraint.

References

AHU74. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974. 309

BGC97. N. Bleuzen-Guernalec and A. Colmerauer. Narrowing a 2n-block of sortings in \$O(n log n)\$. *Lecture Notes in Computer Science,* 1330:2-16, 1997. 306, 318

BGC00. N. Bleuzen-Guernalec and A. Colmerauer. Optimal narrowing of a block of sortings in optimal time. *Constraints : An international Journal,* 5(1-2):85-118, 2000. 306, 307, 312

CM96. Joseph Cheriyan and Kurt Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica,* 15(5):521-549, 1996. 309, 310

Glo67. F. Glover. Maximum matchings in a convex bipartite graph. *Naval Res. Logist. Quart.,* 14:313-316, 1967. 308

GT83. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *ACM Symposium on Theory of Computing (STOC '83),* pages 246-251. ACM Press, 1983. 309

Law76. Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, New York;Chicago;San Francisco, 1976. 308

MN99. Kurt Mehlhorn and Stefan Naher. *LED A: a platform for combinatorial and geometric computing.* Cambridge University Press, Cambridge, November 1999. 308, 315

Moz. The Mozart Programming System, *http://www.MOZ-oz.org.* 312

Pug98. Jean-Fran\c{c}ois Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98),* pages 359-366, Menio Park, July 26-30 1998. AAAI Press. 313

Reg94. J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proc. 12th Conf. American Assoc. Artificial Intelligence,* volume 1, pages 362-367. Amer. Assoc. Artificial Intelligence, 1994. 308, 314