Propagation Algorithms for the Alldifferent Constraint

Basileios Anastasatos

Universität des Saarlandes
B.Anastasatos@Gmail.com

Abstract. The `alldifferent` constraint on a subset of the variables requires that all variables are pairwise different. Probably because it emerges in a wide variety of Constraint Satisfation Problems (CSP), `alldifferent` is the most studied constraint. With the passing of the years, special algorithms have been developed to handle it efficiently. The present article considers three of them. For each algorithm, its goal is defined and the mathematical background behind the algorithm is introduced. Then its nuts and bolts are explained. Finally, the algorithms are compared and appropriate usage of each of them is suggested.

1 Introduction

1.1 Constraint Satisfaction Problems

Definition 1. *A* constraint satisfaction problem *(CSP) is a triple* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$*, where* $\mathcal{X}$ *is a sequence of* n *variables* $x_1, x_2, \ldots, x_n$*,* $\mathcal{D}$ *is a sequence of* n *finite domains* $D_1, D_2, \ldots, D_n$*, where* D_i *is the set of possible values for variable* x_i*, and* $\mathcal{C}$ *is a finite set of constraints between variables.*

Definition 2. *A* constraint C *on the subsequence of variables* $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$*, is a subset of the Cartesian product* $D_{i_1} \times D_{i_2} \times \ldots \times D_{i_m}$*.*

Definition 3. *A solution to a CSP is an assignment to all variables, i.e. a n-tuple* $(d_1, d_2, \ldots, d_n)$ *so that all constraints are satisfied, i.e. for every constraint* C *in* $\mathcal{C}$ *on the variables* $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$*,* $d_{i_1}, d_{i_2}, \ldots, d_{i_m} \in C$ *holds.*

Specifying (some or all) solutions to a given CSP (or determining the absence of them) is the aim of constraint programming.

1.2 The Constraint Alldifferent

Definition 4. *The constraint* `alldifferent` *on the subsequence of variables* $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$ *states that all variables are pairwise different. Formally:*

$$\texttt{alldifferent}\,(x_{i_1}, \ldots, x_{i_m}) := \{(d_{i_1}, \ldots, d_{i_m}) \mid d_{i_l} \in D_{i_l}, j \neq k \Rightarrow d_{i_j} \neq d_{i_k}\} \tag{1}$$

`alldifferent` occurs naturally in a wide variety of problems, which range from puzzles, like the n-queens problem, to assignment problems, to time-tabling, frequency allocation, and, generally speaking, graph colouring problems.

It is built-in in virtually every contraint programming system, for two reasons:

1. The first benefit `alldifferent` offers as built-in, is its syntactic elegance. We can formulate the requirement that m variables are pairwise different either by using `alldifferent` and an argument list of m elements, or by a conjuction of $m\,(m-1)\,/2$ inequalities. The latter is difficult to read, counterintuitive, and error-prone. On the other hand, expressing the same thing with `alldifferent` is readable, elegant and easy to change.
2. Of course, were `alldifferent` merely syntactic sugar, it would not be worth mentioning. The second and most important reason for having `alldifferent` built-in in a system is that it allows us to employ specialized algorithms to handle the constraint much more efficiently than the general-purpose algorithms do.

Unfortunately, the early constraint programming systems did not exploit the extra information provided by `alldifferent` in order to handle it efficiently, but instead treated it internally either as a conjunction of inequalities, or as a usual m-ary constraint solved with the general-purpose algorithms.

However, in the last decade, many special algorithms have been developed, which explore the mathematical structure of `alldifferent` in a variety of ways, achieving different kinds —stronger or weaker— of local consistency, and invariably much faster than in the early stages of constraint programming. Some aspects of this progress form the subject of the present paper.

1.3 Constraint Propagation

To each constraint C is assigned a *propagator*, i.e. a process that tries to shrink the domains of the constraint variables, by removing values that are certainly used in no solution. To this end, the propagator exploits the locally available information, i.e. the information provided by the constraint itself.

On the other hand, when a propagator determines that a value of some domain is excluded from all solutions, it communicates this information to the other propagators that share the same variable, which take into account the removal of the value (i.e. the deletions are *propagated*). This externally provided information may reactivate a propagator that, having exploited all the locally availabe information to shrink the domains, had attained local consistency and could not proceed further.

However, some time a fixpoint of global stability is attained, i.e. all constraints of the CSP become *locally consistent*, and the propagation cycle reaches an end. If at this point the goal of determining (some of) the solutions has still to be reached (which is mathematically possible), the system splits the original problem into at least two CSPs, so that the union of their solution sets is exactly the solution set of the original CSP. This is done by dividing a domain D_i into k disjoint subsets D_{i_1}, D_{i_2}, …, D_{i_k} with $k \geq 2$, the union of which equals D_i and by considering the problems $(X, (D_1, \ldots, D_{i-1}, D_{i_1}, D_{i+1} \ldots, D_m), C)$, …, $(X, (D_1, \ldots, D_{i-1}, D_{i_k}, D_{i+1} \ldots, D_m), C)$ instead of the original one. Repeating of this procedure (constraint propagation, followed by splitting) creates a search tree that has the original problem at its root and the solved (or failed) problems at its leaves.

Theoretically the search for solutions through a search tree would suffice to determine all solutions, without having to interleave splitting with constraint propagation. However, the complexity of this method is exponential, which renders it useless for practical purposes. Constraint propagation seeks to *prune* large portions of the search tree *beforehand*, thus speeding up the solution process. Gains in speed are of course possible, only if pruning portions of the search tree takes less time than searching these portions for solutions, hence the need for efficient propagation algorithms!

2 Strong Form of Local Consistency

2.1 Hyper-arc Consistency

Definition 5. *An m-ary constraint C[1] on the variables x_{i_1}, x_{i_2}, …, x_{i_m}, is called* hyper-arc consistent *iff for every variable x_{i_k} and for every value d of the domain D_{i_k} of x_{i_k}, there exist $d_{i_1} \in D_{i_1}, d_{i_2} \in D_{i_2}$, …, $d_{i_{(k-1)}} \in D_{i_{(k-1)}}$, $d_{i_{(k+1)}} \in D_{i_{(k+1)}}$, …, $d_{i_m} \in D_{i_m}$, such that $\left(d_{i_1}, d_{i_2}, \ldots, d_{i_{k-1}}, d, d_{i_{k+1}}, d_{i_m}\right)$, is in C, i.e. iff all values of the domains of the variables constrained by the constraint are used in some permissible tuple of the constraint at the position corresponding to the variable.*

Definition 6. *The hyper-arc consistency is called simply* arc consistency *if the considered constraint is binary.*

Note 1. The (somewhat counterintuitive) terms hyper-arc resp. arc consistency stem from the graph theory. A hyper-arc beginning at the vertex v_1, passing through the vertices v_2, …, v_{n-1} and ending at the vertex v_n is the graph equivalent of the n-tuple $(v_1, v_2, \ldots, v_n)$. An arc from v_1 to v_2 is the graph equivalent of the ordered pair (v_1, v_2).

Hyper-arc consistency is the "real" local consistency. A constraint has a solution iff it can be made hyper-arc consistent. Hyper-arc consistency corresponds to the best possible pruning by exploiting only the local information provided by the constraint. We are going to see that the notion of local consistency can be relaxed.

[1] Not necessarily the `alldifferent`.

2.2 Early Approaches

As noted above, the early constraint programming systems treated `alldifferent` internally either as a conjunction of inequalities and used known algorithms to make each inequality arc-consistent, or applied general-purpose algorithms for achieving hyper-arc consistency.

Binary Decomposition. The constraint `alldifferent` $(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$, is replaced by $m\,(m-1)\,/2$ binary constraints $x_{i_j} \neq x_{i_k}\ \forall j < k$. Then every binary constraint is made arc-consistent. The algorithm works as follows: as soon as the domain of a variable is reduced to one value, i.e. as soon as a variable is assigned a value, this value is removed from the domains of all other variables.

This approach has two important drawbacks:

1. From one m-ary `alldifferent` constraint $O\left(m^2\right)$ binary constraints are produced.
2. Arc consistency of all binary constraints does not imply hyper-arc consistency of the original m-ary constraint.

   Example 1. The constraint `alldifferent` (x_1, x_2, x_3) with domains $D_1 = D_2 = D_3 = \{1,2\}$, is not hyper-arc consistent (actually it is not even consistent.) However the binary `alldifferent` constraints $x_1 \neq x_2$, $x_2 \neq x_3$, and $x_1 \neq x_3$ are all arc-consistent, so no domain can be shrinked during the propagation phase.

General-purpose Algorithms. The second solution was to use some general-purpose algorithm for achieving hyper-arc consistency of an arbitrary m-ary constraint. Such algorithms do their task, suffer nevertheless from a very high complexity, e.g. the algorithm by Mohr and Masini (1988) has cost $d!/\,(d-m)!$ for m variables and domains size d.

3 Régin's Algorithm

The first special algorithm for achieving local consistency of `alldifferent` was created by Régin (1994). It achieves hyper-arc consistency and makes use of results of the field of graph theory called matching theory to dramatically reduce the cost.

3.1 Matching Theory

Definition 7. *Given a graph $G = (V, E)$, a subset M of the edges E is called a* matching *iff no two edges share a vertex, i.e. iff the degree of the vertices in the graph (V, M) is at most one.*

Definition 8. *Given a matching M, an edge $e \in E$ is called* matching *if $e \in M$, and* free *otherwise.*

Definition 9. *Given a matching M, a vertex $v \in V$ is called* matched *if it is incident to some edge $e \in M$, and* free *otherwise, i.e. v is matched $\Leftrightarrow deg\,(v) = 1$ and v is free $\Leftrightarrow deg\,(v) = 0$.*

Definition 10. *A matching M is said to* cover *a set of vertices X iff all vertices in X are matched.*

Definition 11. *Given a matching M, an* alternating path *(resp.* circuit*) of the graph G is a (simple) path (resp. circuit) whose edges are alternately matching and free.*

Definition 12. *Given a graph $G = (V, E)$, the* maximum matching problem *asks for a matching $M \subseteq E$ of maximum cardinality, i.e. one that contains as many matching edges as possible, or, equivalently, leaves as few free vertices as possible.*

3.2 Application of Matching Theory to Régin's Algorithm

The relevance of the maximum matching problem to our problem of making `alldifferent` hyperarc-consistent lies on three facts:

1. For an m-ary constraint C on the variables $X_C = x_{i_1}, x_{i_2}, \ldots, x_{i_m}$, we can represent the information that each x_{i_j}, takes its values in D_{i_j}, by a special bipartite graph $\left(X_C \bigcup \left(\bigcup_{j=1}^{m} D_{i_j}\right), E\right)$, where $e = \{x_{i_j}, d\} \in E$ iff $d \in D_{i_j}$. In other words, the set of the vertices consists of two disjoint sets X_C and $\bigcup_{j=1}^{m} D_{i_j}$, X_C being the set of the variables constrained by C, and $\bigcup D_{i_j}$ being the union of the domains of the variables in X_C. Furthermore, an edge joins a variable x_{i_j} and a value d iff d belongs to the domain of x_{i_j}, D_{i_j}. This bipartite graph is called the *value graph* of X_C and can clearly be constructed in linear time.
2. `alldifferent` $(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$, has a solution iff a maximum matching M of the value graph G of $X_C = x_{i_1}, x_{i_2}, \ldots, x_{i_m}$, has size m. Furthermore, there exist efficient algorithms that, given V, determine a maximum matching M in $O\left(\sqrt{(|X_C|)} \cdot |E|\right)$.
3. Given a maximum matching M of G with size $|M| = m$, efficient algorithms exist that can make `alldifferent` hyper-arc consistent.

Proof. The first fact, namely that the value graph G of X_C encodes all possible assignments to the variables in X_C, as well as that G can be constructed efficiently, should be clear.

The second fact is also clear: by construction of G, a matching M of size m covers X_C and is also a maximum matching. Furthermore, to each such matching $M = \{\{x_{i_1}, d_{i_1}\}, \{x_{i_2}, d_{i_2}\}, \ldots, \{x_{i_m}, d_{i_m}\}\}$, corresponds a solution $(d_{i_1}, d_{i_2}, \ldots, d_{i_m})$, of `alldifferent` (X_C) and, conversely, to each solution of `alldifferent` (X_C) corresponds a matching M covering X_C.

Furthermore, `alldifferent` (X_C) is hyper-arc consistent iff every edge e of the value graph of X_C belongs to some matching M that covers X_C. The proof follows directly from the definition of hyper-arc consistency and the second fact.

The third fact needs more explanation. The second fact implies that we can make `alldifferent` (X_C) hyper-arc consistent, iff its value graph G possesses a matching M of size $|X_C| = m$ (otherwise `alldifferent` (X_C) would be inconsistent) and if we remove every value d in some D_{i_j}, whose corresponding edge $e = \{x_{i_j}, d\}$ belongs to no matching of size m in G. Fortunately, we do not have to compute *all* maximum matchings, for if we know just *one* such arbitrary matching M, then we can efficiently compute if an edge of G belongs to *some* matching of size m.

Theorem 1. *An edge e belongs to some maximum matching $\mathbb{N}$ of size m, iff, for a given arbitrary maximum matching M of size m, it either belongs to M or to an M-*alternating path of even length that begins at a M-free vertex*, or to an M-*alternating cycle of even length.

So, iff an edge $e = \{x_{i_j}, d\}$ belongs to no such alternating path or circuit, then we delete d from D_{i_j}. But how can we discover these alternating paths and circuits?

From the value graph $G = (V, E)$ and M we construct a *directed* graph $G' = (V', E')$ in the following way: $e' = \left(x_{i_j}, d\right) \in E'$ iff $e = \{x_{i_j}, d\} \in M$, and $e' = \left(d, x_{i_j}\right) \in E'$ iff $e = \{x_{i_j}, d\} \in E \backslash M$. In other words, we direct all M-matching edges from the variables to the values and all M-free edges from the values to the variables.

Theorem 2. *1. Every directed circuit of G' has even length (because G' is bipartite by construction) and corresponds to an even alternating cycle of G and vice versa.*

2. Every directed (simple) path of G' that begins at a free vertex is either even, or it is odd and can be extended at its end to an even path. This even path corresponds to an even alternating path of G that begins at a free vertex, and vice versa.

It follows that we can find the paths and circuits of G we were looking for above, by determining the strongly connected components in G', which correspond to directed circuits in G', as well as the directed paths in G' beginning at a free vertex. The first task takes $O\left(m + |E'|\right)$ time, the second one $O\left(|E'|\right)$ time.

Remark 1. As M covers X_C, all M-free vertices are not in X_C, but in $\bigcup_{x_{i_j} \in X_C} D_{i_j}$.

The complete algorithm works as follows:

1. Given is $C =$ `alldifferent` (X_C), with $X_C = x_{i_1}, x_{i_2}, \ldots, x_{i_m}$, and x_{i_j} taking values from D_{i_j}.
2. Construct the value graph $G = \left(X_C \bigcup_{x_{i_j} \in X_C} \left(\bigcup D_{i_j}\right), E\right)$.

3. Compute some maximum matching M of G.
4. If $|M| < |X_C| = m$ then *fail*.
5. Else construct directed graph G' as explained.
6. Mark all edges of G' that correspond to an edge of G that belongs to M as "consistent".
7. Find all strongly connected components (SCC) in G' and mark the edges connecting the vertices of each SSC as "consistent".
8. Find all directed paths that begin at a free vertex and mark their edges as "consistent".
9. For each edge e' in G' that has not been marked as "consistent", find the corresponding edge $e = \{x_{i_j}, d\}$ in G and remove d from D_{i_j}.

The complexity of the algorithm is that of finding a maximum matching, i.e. $O\left(\sqrt{(|X_C|)} \cdot |E|\right)$. For $|E| \sim |X_C|^2$ the complexity is $O\left(|X_C|^{5/2}\right)$.

3.3 Incremental Property of Régin's Algorithm

As mentioned above, in the course of a propagation cycle, values are removed not only because they are locally inconsistent, but also because they were found inconsistent to some other constraint. This means, after removing some value we also have to update our value graph and, possibly, our maximum matching M.

The naive and time-consuming way of reconstructing the matching from scratch can fortunately be avoided, as there exists an effiecient algorithm for computing a matching covering X_C from a matching of cardinality $|M - k|$, where k is the number of the edges removed from M. The complexity of this algorithm is $O\left(\sqrt{k} \cdot m\right)$.

The complexity of the whole propagation cycle is $O\left(m^2 \cdot d^2\right)$ for m variables with domains of average size d.

4 Weaker Forms of Local Consistency

Remark 2. For the rest of the article, it is assumed that all domains are subsets of a linearly ordered set Ω and that for every two elements l, u of that set, $l \leq u$ implies $|\{x \text{ such that } l \leq x \leq u\}| \leq \infty$, i.e. between any two elements of Ω there are only finitely many elements. We write $[l, u]$ for the interval $\{x \text{ such that } l \leq x \leq u\}$.

Remark 3. In practice, Ω is always the set of the whole numbers $\mathbb{Z}$, as the algorithms presented take for granted e.g. that $|\,[l, u]\,| = u - l + 1$, or that every element has an inverse, so that $l \leq u \Leftrightarrow -l \geq -u$.

4.1 Leconte's Algorithm

This algorithm was created by Leconte (1996). It derives from the celebrated Hall's Marriage Theorem of combinatorics and achieves the so-called "range consistency".

Range Consistency. Range consistency is a relaxation of hyper-arc consistency.

Definition 13. *An m-ary constraint C on the variables x_{i_1}, x_{i_2}, ..., x_{i_m}, is called* range consistent *iff for every variable x_{i_k} and for every value d of the domain D_{i_k} of x_{i_k}, $\exists\ d_{i_1} \in [\min(D_{i_1}), \max(D_{i_1})]$, $d_{i_2} \in [\min(D_{i_2}), \max(D_{i_2})]$, ..., $d_{i_{k-1}} \in \left[\min\left(D_{i_{k-1}}\right), \max\left(D_{i_{k-1}}\right)\right]$, $d_{i_{k+1}} \in \left[\min\left(D_{i_{k+1}}\right), \max\left(D_{i_{k+1}}\right)\right]$, ..., $d_{i_m} \in [\min(D_{i_m}), \max(D_{i_m})]$, such that $\left(d_{i_1}, \ldots, d_{i_{k-1}}, d, d_{i_{k+1}}, \ldots, d_{i_m}\right)$ is in C, i.e. iff all values of the domains of the variables constrained by the constraint were used in some permissible tuple of the constraint at the position corresponding to the variable, had we filled the gaps of the other domains by replacing them by the minimal intervals containing them.*

Hyper-arc consistency implies range consistency, while the converse is not true (range consistency does not "see the holes").

Hall's Theorem and its Relation to Range Consistency.

Theorem 3. *The constraint* `alldifferent` $(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$ *has a solution iff for every subset $K \subseteq (x_{i_1}, x_{i_2}, \ldots, x_{i_m}) : |K| \leq |D_K|$ holds, where D_K is the union of the domains of the variables in K.*

Theorem 4. *The constraint* `alldifferent` $(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$ *is range consistent iff no domain D_{i_j} is empty and for every subset $K \subseteq (x_{i_1}, x_{i_2}, \ldots, x_{i_m})$: $|\,[\min(D_K), \max(D_K)]\,| = |K|$ implies $D_{i_j} \bigcap [\min(D_K), \max(D_K)] = \emptyset\ \forall x_{i_j} \notin K$.*

In other words, if the minimal interval containing the union of the domains of a subset of the constrainted variables equals the cardinality of this subset, then no value in this interval should be contained also in the domain of a variable not in K.

Definition 14. *A subset K for which the condition of the theorem holds, is called a* Hall set*.*

Nuts and Bolts of Leconte's Algorithm. It can be proven that the time complexity of every algorithm achieving range consistency of the `alldifferent` constraint is at least $O\left(m^2\right)$.

The idea of Leconte's algorithm exploits the above mentioned theorem: find all Hall sets K and remove all values in D_K from the domains of all variables not in D_K.

A naive algorithm would work as follows: with u ranging over all maximum domain values and l over all minimum domain values, it would check if the interval $[l, u]$ corresponds to a Hall set, by counting the number of variables whose domains fall in $[l, u]$. For each detected Hall set K it would then remove the values in $[l, u]$ from the domain of every variable x not in K.

This sums up to three nested loops, one for u, one for l, and one for x, each one of them having $O(m)$ steps, yielding a total complexity of $O\left(m^3\right)$. Furthermore, if an $[l, u]$ is detected, such that the number of variables whose domains are contained in $[l, u]$ is greater than $|\,[l, u]\,|$, then the constraint is inconsistent and the algorithm terminates immediately.

We can reduce the complexity to the optimal $O\left(m^2\right)$ if we find a way to update the domains not in $O(m)$, as the naive algorithm does, but in $O(1)$. To this end we first sort the domains twice, once in descending order of their maximums and once in descending order of their minimums and save the two orderings. The sorting takes $O(m \log m)$ time, which is less than $O\left(m^2\right)$.

Then u loops over all maximum domain values in decreasing order and l loops over all minimum domain values in decreasing order. When a Hall set is detected, the current position i in the array `descMin`, which contains the domains sorted in descending minimum order, is saved and $[l, u]$ is removed on the fly from the domains of the variables that follow in the loop, but not from the domains of the variables already checked. The removal of $[l, u]$ on the fly does not increase the complexity of the algorithm.

If a new Hall set is detected, then the current $[l, u]$ contains the old $[l, u]$ that corresponded to the previously detected Hall set, because l is descending and u constant. The new current position in the array `descMin` is saved, overwriting the previously saved one.

Once the loop of l is over, we loop once more from the beginning of `descMin` till the saved position i and remove $[\min(\texttt{descMin}[i]), \max(\texttt{descMin}[i])]$ from all domains falling in that interval. This loop has complexity $O(m)$, so the two loops together (the last one and the loop of l) have a complexity of $O(2 \cdot m) = O(m)$. Considering also the outer loop of u we obtain a total complexity of $O\left(m^2\right)$ for the whole algorithm.

Incremental Property of Leconte's Algorithm. Like Régin's algorithm, Leconte's algorithm can work incrementally during a propagation cycle. The algorithm's cost during a complete propagation cycle is $O\left(m^2 \cdot d\right)$, where d is the average domain size.

4.2 Puget's algorithm

This algorithm appeared two years after Leconte's algorith (1998) and also exploits Hall's Theorem to detect inconsistent values. It achieves a third kind of local consistency, called "bounds consistency".

Bounds Consistency. Bounds consistency is a relaxation of range consistency (which was in turn a relaxation of hyper-arc consistency.)

Definition 15. *An m-ary constraint C on the variables x_{i_1}, x_{i_2}, …, x_{i_m}: is called* bounds consistent *iff for every variable x_{i_k} and for both bounds of its domain D_{i_k} i.e. for $d \in \{\min(D_{i_k}), \max(D_{i_k})\}$, $\exists\ d_{i_1} \in [\min(D_{i_1}), \max(D_{i_1})]$,*

$d_{i_2} \in [\min(D_{i_2}), \max(D_{i_2})]$, …, $d_{i_{k-1}} \in \left[\min\left(D_{i_{k-1}}\right), \max\left(D_{i_{k-1}}\right)\right]$, $d_{i_{k+1}} \in \left[\min\left(D_{i_{k+1}}\right), \max\left(D_{i_{k+1}}\right)\right]$, …, $d_{i_m} \in [\min(D_{i_m}), \max(D_{i_m})]$, such that the tuple $\left(d_{i_1}, d_{i_2}, \ldots, d_{i_{k-1}}, d, d_{i_{k+1}}, \ldots, d_{i_m}\right)$ is in C, i.e. iff the bounds of the domains of all variables constrained by the constraint were used in some permissible tuple of the constraint at the position corresponding to the variable, had we filled the gaps of the domains by replacing them by the minimal intervals containing them.

Clearly range consistency implies bounds consistency, while the converse does not hold (for bounds consistency considers only the bounds of each domain, instead of its whole range, as range consistency does).

Application of Hall's Theorem to Bounds Consistency.

Theorem 5. *The constraint* `alldifferent` $(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$ *is bounds consistent iff no domain D_{i_j} is empty and for each interval $|I| >= |D_I|$ (where $D_I := \bigcup D_{i_j}$ and $D_{i_j} \subseteq I$) and if $|I| = |D_I|$ implies $D_{i_j} \not\subseteq I \Rightarrow \max\left(D_{i_j}\right), \min\left(D_{i_j}\right) \notin I$. In other words, if the number of all variables whose domains fall in an interval I equals the cardinality of I (which implies that any assignment of these variables will use all the values in I, thus forbidding their use by the other variables), then no bounds of the domains of the other variables are in I.*

Definition 16. *An interval I satisfying $|I| = |D_I|$ is called a* Hall interval.

Nuts and Bolts of Puget's Algorithm.

Remark 4. Puget's algorithm updates only the lower bounds of the domains. To also update the upper bounds, we inverse the domains by multiplying them by -1, rerun the algorithm and then restore the original form of the domains by multiplying them once more by -1.

A naive algorithm would work as follows: with u ranging over all maximum domain values and l over all minimum domain values, it would check if the interval $[l, u]$ is a Hall interval, by counting the number of variables whose domains fall in $[l, u]$. For each detected Hall interval $[l, u]$ it would then remove the values in $[l, u]$ from the domain of every variable x whose domain is not in $[l, u]$.

The complexity of the two loops for l and u is $O\left(m^2\right)$ and the complexity of each one of the inner loops for detecting Hall intervals resp. for updating the domains is $O(m)$, so the overall complexity is $O\left(m^3\right)$. Furthermore, if an $[l, u]$ is detected, such that the number of variables whose domains are contained in $[l, u,]$ is greater than $|\,[l, u]\,|$, then the constraint is inconsistent and the algorithm terminates immediately.

It can be proven that given m variables, the number of Hall intervals can be at least m^2. So every algorithm that loops over all Hall intervals is bound to have complexity of at least $O\left(m^2 \cdot \text{time to update the domains}\right)$.

Fortunately it is not necessary to consider all Hall intervals. For if $l \leq l'$ and $[l, u]$ and $[l', u]$ are both Hall intervals, then every value we delete by considering

$[l', u]$ would be deleted anyway by considering $[l, u]$. So for given u it suffices to detect the *left-maximal* Hall interval that has u as its right bound and is unique, provided it exists.

The improved algorithm works as follows: with u ranging over all maximum domain values, l ranges over all minimum domain values to find the left-maximal Hall interval with u as its right bound. Then we range over all domains to update them. The complexity of each of the outer loops and of the two consecutives inner loops is $O(m)$, so the overall complexity is reduced to $O\left(m^2\right)$.

We can further improve the complexity if we can detect left-maximal Hall intervals and update the domains in logarithmic time instead of linear one. Both tasks are possible if we first sort the variables twice, one in ascending order of their maximums and once in ascending order of their minimums (and, of course, store the two orderings), consider each variable once in ascending order of their maximums, and use balanced binary trees both to keep special counters used to detect Hall intervals, as well as to update the domains.

Updating of the counters and the domains is done in a lazy way along a path leading from the root of the tree to a leaf. When we discover a Hall interval $[l, u]$ after having considered the first i variables in ascending order of their maximums, the only candidates for updating their domains can be found in the yet unconsidered variables, which have a maximum greater of equal to the considered ones, because only these variables may have a minimum $\geq l$ and a maximum $> u$. Now, as the variables are kept in a balanced tree in ascending order of their minimums, it suffices to update a frontier that consists of the internal nodes whose children's minimum is at least l and the leaf holding the first variable whose minimum is at least l, by setting the minimum to at least $u + 1$. When it is time to consider a variable, we travel from the root to it and set its minimum to be the greatest minimum found on the path we walked on. Updating of the counters is done similarly.

As for balanced trees with m leaves all paths have length at most $\log m$, the overall complexity of the algorithm is reduced to $O(m \log m)$. As the cost of sorting the variables at the beginning is again $O(m \log m)$, it does not affect the complexity of the algorithm.

Further Improvements. The algorithm was further improved by López et al. (2003.) Mehlhorn et al. (2000) devised another algorithm based on matching theory. Both algorithms have linear complexity plus the cost of sorting the variables at the beginning. In special cases, this cost can be linear.

5 Comparisons

For non-ordered domains, Régin's algorithm clearly outperforms the early solutions of binary decomposition and general-purpose algorithms, as it has the pruning performance of the second one at a speed of the first one. As Leconte's and Puget's algorithms rely on arithmetical properties of the integers, they are not applicable to arbitrary domains.

However, if the domains are subsets of the integers, then Leconte's algorithm achieves local consistency faster than Régin's and Puget's faster than Leconte's. Because of the relaxed notions of local consistency they achieve, their pruning performance is inferior to that of Régin's algorithm, but nevertheless their overall performance is better. It should be expected that the pruning performance of the two last algorithms will equal that of Régin's algorithm if the domains represent intervals, i.e. if they have no gaps at all or at least only a few.

Régin's and Leconte's algorithms share the benefit of working incrementaly during a complete propagation cycle (Leconte's algorithm being d times faster than Régin's, where d is the average domain size), which is not the case with Puget's algorithm. However, the latter still outperforms Leconte's algorithm.

The improved algorithm by López-Ortiz et al. as well as the algorithm by Mehlhorn et at. are even faster than the one by Puget.

It seems that the savings achieved by pruning the search tree as well as possible are inferior to the time required for this better pruning.

References

Van Hoeve, W.: The Alldifferent Constraint: a Systematic Overview. Submitted manuscript, January 2005.

Mohr, R., Masini, G.: Good Old Discrete Relaxation. In European Conference on Artificial Intelligence (ECAI), pp. 651-656.

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

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

Puget, J.-F.: A fast algorithm for the bound consistency of alldiff constraints. In Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI / IAAI), pp. 359-366. AAAI Press / The MIT Press.

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

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